
Enterprise Application Management with PATROL®

David A. Spuler

To join a Prentice Hall PTR Internet mailing list, point to
<http://www.prehnhall.com/register>



Prentice Hall PTR
Upper Saddle River, NJ 07458
<http://www.prenhall.com>

Library of Congress Cataloging-in-Publication Data

Spuler, David A.: Enterprise Application Management with PATROL® XXXXXXX

/ David A. Spuler

p. cm,

Includes bibliographical references and index.

ISBN 0-13-XXXXXX-X

1. XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXX

XXXXXXX

XXXXXXXXXX

CIP

Editorial/production supervision: *Name*

Cover design: *Name*

Cover photo: *Name*

Cover design director: *Jerry Votta*

Manufacturing manager: *Alexis R. Heydt*

Marketing manager: *Name*

Acquisitions editor: *Name*



©1997 Prentice Hall PTR

Prentice-Hall, Inc.

A Simon & Schuster Company

Upper Saddle River, New Jersey 07458

Prentice Hall books are widely used by corporations and government agencies for training, marketing, and resale.

The publisher offers discounts on this book when ordered in bulk quantities.

For more information, contact Corporate Sales Department, Phone: 800-382-3419;

Fax: 201-236-7141; E-mail: corpsales@prenhall.com

Or write: Prentice Hall PTR, Corp. Sales Dept., One Lake Street, Upper Saddle River, NJ 07458.

Product names mentioned herein are the trademarks or registered trademarks of their respective owners.

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-XXXXXX-X

Prentice-Hall International (UK) Limited, *London*

Prentice-Hall of Australia Pty. Limited, *Sydney*

Prentice-Hall Canada Inc., *Toronto*

Prentice-Hall Hispanoamericana, S.A., *Mexico*

Prentice-Hall of India Private Limited, *New Delhi*

Prentice-Hall of Japan, Inc., *Tokyo*

Simon & Schuster Asia Pte. Ltd., *Singapore*

Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

To Anita, who is my whole life.



Preface

Application management is a state-of-the-art management concern. It is about bringing the art of management up to the high-level application layer and managing at that level instead of at the hardware or network infrastructure layer. Over time, Information Technology (IT) management has gone through various levels of sophistication from network management and systems management to the higher level of application management. By leveraging this higher level, many IT organizations are reaping the returns of a more structured approach to managing their applications.

This book is about application management as a discipline and also about PATROL as an existing solution. I have separated the more theoretical aspects of the discipline from the practical details of the PATROL product. For example, two chapters discuss architecture, one from a mostly theoretical point of view, and one covering PATROL architecture in detail. Certainly, other products offer application management functionality, but my belief is that PATROL is the best solution available today. This book is very much a book about PATROL, and I do not attempt to cover any other products that play in the application management arena.

This work showcases how the various pieces of PATROL fit together to allow you to achieve the advantages of advanced application management through PATROL. While offering a high level of flexibility, PATROL is also a large product suite with many components, and this book demonstrates how each piece can be used to complement others.

PATROL can be a very rewarding product to learn. Extending its capabilities is a quick and easy task that can show quick returns for your efforts. The open architecture allows PATROL to easily be applied to places where it is needed and adds value. Although the broad set of off-the-shelf products allow quick solutions for the most common IT needs, the customizability of PATROL is where it truly shines. Taking PATROL that little bit further can provide

tremendous payoff by automatically detecting and correcting the site or machine-specific problems that beset you through the working day.

Who This Book Is For

This book addresses a number of different readership roles. The discussion of the theoretical aspects of application management as a management discipline should be interesting to all industry readers.

IT Personnel. Application management is a discipline that should be of interest to anyone involved with the management of software applications in a corporate environment.

CIOs and IT Managers. As the manager of an IT organization, you know only too well the pain of managing diverse applications. This book examines the implementation of application management to achieve relief through a more proactive approach. There is also information for defining the advantages of application management to help justify it to senior executives who may be less familiar with the technical details.

IT Architects. This book shows what the goal should be and allows IT architects and planners to best decide how to plug PATROL into the management scheme to get the best value. Application management is a discipline that affects the entire IT organization. Full leveraging of the advantages of application management involves careful planning, implementation, and often development efforts that consider application management during the development of in-house business applications.

Administrators and Implementers. Administrators and implementers will find the discussion of the deployment or implementation aspects of PATROL useful. I have covered the theoretical aspects of application management implementation as a general problem, and also the practical aspects of PATROL implementation. There is coverage of issues such as configuration, distribution, pager/email notification, SNMP trap notification, and integrations with other software components.

Developers. Developers and advanced IT administrators will find the various discussions of how to write a Knowledge Module and how to integrate with PATROL useful material. In-house application developers will find the discussion of how to instrument the application to make it more management-friendly a thought-provoking discussion. Developers of commercial applications or development tools should consider adding application management capabilities for their associated benefits to their applications or toolsets.

Operators. Operators can benefit from a deeper understanding of the PATROL product and the overall value of their place in application management procedures. However, this book

is not really tutorial in nature and assumes that the details of PATROL are learned from another source such as the documentation or a training course.

Analysts. The topic of application management has been a hot area of research by technology industry analysts. Many of their papers have offered ideas and are cited in the bibliography. This book should give back some interesting viewpoints on the theory and practice of application management.

Instructors. An innovative course on application management would be highly useful for students as they enter the workplace.

Application management is a topic that touches many job roles in the IT organizations. This book offers something for everyone to whom application management is important.

Trademarks

This book mentions many names or symbols that are trademarks or registered trademarks. The trademark credits listed below were gathered from various sources. All product, trade, service, or company names or service marks not shown below are trademarks, service marks, registered trademarks, or registered service marks of their respective holders.

UNIX is a registered trademark licensed exclusively through X/Open Company Ltd. PostScript is a registered trademark of Adobe Systems, Inc. AT&T is a registered trademark of American Telephone & Telegraph. Apple, AppleTalk, Mac, Macintosh, and Power Macintosh are trademarks or registered trademarks of Apple Computer, Inc. NetWare, Novell, NetWare 3, NetWare 4, and NLM are trademarks of Novell, Inc. Compaq is a registered trademark of Compaq Computer Corporation. DEC, Alpha AXP, VAX, DIGITAL, ULTRIX and VMS are trademarks or registered trademarks of Digital Equipment Corporation. Hewlett-Packard, and HP are registered trademarks of Hewlett-Packard Company. Intel is a registered trademark, and 286, 386, and Pentium are trademarks of Intel Corporation. AIX, IBM, OS/2, DB/2, CICS, IMS, AS/400, OpenEdition, Presentation Manager, RS/6000, System/390, NetView, OS/390, OS/400, SP, PowerPC, Presentation Manager, RISC System/6000, and VisualAge are trademarks or registered trademarks of International Business Machines Corporation. Microsoft, Microsoft Access, MS, Windows, Visual Basic, MS-DOS, Visual C++, Win32, Win32s, and Windows NT are trademarks or registered trademarks of Microsoft Corporation. Motorola is a registered trademark of Motorola Corporation. NCR is a registered trademark of NCR Corporation. NEC is a registered trademark of NEC Corporation. Oracle is a registered trademark of Oracle Corporation and the Oracle product names are trademarks or registered trademarks of Oracle Corporation. Motif, OSF, DCE, and OSF/Motif are registered trademarks of Open Software Foundation, Inc. SCO is a registered trademark of Santa Cruz Operations, Inc. Silicon Graphics is a registered trademark of Silicon Graphics, Inc. NFS, Network File System, Solaris, Java, SunOS and Sun-Soft are trademarks or registered trademarks of Sun Microsystems, Inc. SYBASE is a registered trademark of Sybase, Inc. Tivoli Management Environment and TME 10 are trademarks or registered trademarks of Tivoli Systems, an IBM Company. SAS is a registered trademark of SAS Institute, Inc. MIPS is a registered trademark of MIPS Computer Systems, Inc. X Window System and X-Windows are trademarks of The Massachusetts Institute of Technology.

BMC Software, the BMC Software logos, Application Service Assurance or ASA[™], BEST/1[®], I₂O Application Management Toolkit[™], NetTUNE[®] PRO, OPERTUNE[®], PATROL[®], PATROL Agent[™], PATROL Agent[™] for I₂O, PATROL Agent[™] for OS/390, PATROL Agent[™] for OpenVMS, PATROL Agent[™] for OS/2, PATROL Agent[™] for Unix, PATROL Agent[™] for Windows NT, PATROL Alarm Manager[™], PATROL CGI Application Server[™], PATROL CGI Server[™], PATROL Console[™], PATROL Console[™] for Windows NT-Developer, PATROL Console[™] for Windows NT-Operator, PATROL Console[™] for Windows NT-Developer/E, PATROL Console[™] for Windows NT-Operator/E, PATROL Console[™] for Unix-Developer, PATROL Console[™] for Unix-Operator, PATROL DB-Admin Knowledge Module[™], PATROL DB-Change Manager[™], PATROL DB-Reorg[™], PATROL DB-Stats[™], PATROL DB-

Voyager™, PATROL® Development Bundle, PATROL® Event Manager Console, PEM Console, PATROL® Intelligent Autonomous Agent, IAA, PATROL Agent, PATROL® Integration Module, PATROL IM™, PATROL IP/Optimizer™, PATROL IP/O™, PATROL Knowledge Module™, PATROL KM™, PATROL Knowledge Module™ for ADABAS C, PATROL Knowledge Module™ for ADABAS D, PATROL Knowledge Module™ for ADABAS SQL Server, PATROL Knowledge Module™ for AS/400, PATROL Knowledge Module™ for Baan Applications, PATROL Knowledge Module™ for CA-OpenIngres and CA-Ingres, PATROL Knowledge Module™ for CGI Applications, PATROL Knowledge Module™ for CGI Server, PATROL Knowledge Module™ for Compaq Insight Manager, PATROL Knowledge Module™ for DB2 Universal Database, PATROL Knowledge Module™ for DCE, PATROL Knowledge Module™ for Informix, PATROL Knowledge Module™ for Internet Servers, PATROL Knowledge Module™ for Lotus Notes, PATROL Knowledge Module™ for Microsoft SQL Server, PATROL Knowledge Module™ for MQSeries, PATROL Knowledge Module™ for Networked Applications, PATROL Knowledge Module™ for NetWare, PATROL Knowledge Module™ for OmniGuard/Intruder Alert, PATROL Knowledge Module™ for OpenVMS, PATROL Knowledge Module™ for Oracle, PATROL Knowledge Module™ for Oracle Applications, PATROL Knowledge Module™ for Oracle OS/390, PATROL Knowledge Module™ for Oracle Parallel Server, PATROL Knowledge Module™ for Oracle Symmetric Replication, PATROL Knowledge Module™ for OS/2, PATROL Knowledge Module™ for OS/390, PATROL Knowledge Module™ for PeopleSoft Applications, PATROL Knowledge Module™ for PeopleSoft HRMS, PATROL Knowledge Module™ for PROGRESS, PATROL Knowledge Module™ for Rdb, PATROL Knowledge Module™ for RS/6000 SP, PATROL Knowledge Module™ for R/3, PATROL Knowledge Module™ for Sybase Replication Server, PATROL Knowledge Module™ for Sybase SQL Server, PATROL Knowledge Module™ for Transaction Server for Open Systems, PATROL Knowledge Module™ for TUXEDO, PATROL Knowledge Module™ for Unify VISION, PATROL Knowledge Module™ for Unix, PATROL Knowledge Module™ for Windows NT, PATROL® Knowledge Module Toolkit, PATROLNET™, PATROL Pathfinder™, PATROL® Recovery Manager, PATROL RMGR™, PATROL SQL-BackTrack Knowledge Module™, PATROL SQL-BackTrack KM, PATROL SQL-Explorer™, PATROL® SNMP Toolkit, PATROL® Toolkit for Console Integration, PATROLVIEW®, PATROLVIEW® for Cabletron SPECTRUM, PATROLVIEW® for Unicenter TNG, PATROLVIEW® for HP OpenView Network Node Manager, PATROLVIEW® for HP OpenView IT/Operations, PATROLVIEW® for Remedy AR System, PATROLVIEW® for Solstice Domain Manager, PATROLVIEW® for TME 10 NetView, PATROLVIEW PLUS® for Tivoli, PATROLWATCH™, PATROLWATCH™ for Web Browsers, PATROLWATCH™ for Windows, PATROL Web Analyzer™, SQL-BackTrack™, Authorized PATROL Service Provider, APSP™, BMC Software, Inc., BMC Software, Certified for PATROL, Designed for PATROL, PATROL® Developer Network or PDN™, PATROL® History Loader Knowledge Module, PHL™ KM, PATROLMARK™, PATROLMARK™ Solutions Catalog, PATROL® Script Language, PSL, and all other BMC Software product or service names are trademarks or registered trademarks of BMC Software, Inc. in the USA and in other select countries.

Disclaimer

The author and publisher have used their best efforts in developing this book and associated software. The information and software in this book distribution are provided on an “as is” basis and the author and publisher hereby disclaim all warranties, express or implied, including those of merchantability and fitness for any particular purpose. The author and publisher make no warranties or representations regarding the accuracy or completeness of the information. In no event shall the author or publisher be liable for any special, indirect, incidental, or consequential damages, lost profits, lost data, or any indirect damages, arising out of or in connection with the use or performance of this information or software, even if the author or publisher have been informed of the possibility thereof.

Permissions

Parts of this work used with permission from BMC Software, Inc.

Acknowledgments

Thanking everyone who has contributed to my PATROL experience would be impossible to do without missing someone. Rather than attempt this, I would just like to thank everyone involved with PATROL in Houston, Austin, Melbourne, Frankfurt, Paris, and London. Special thanks to Mark Sterin for his technical review and also to Kirill Tatarinov, Brinkley Morse, Wayne Morris, Subhash Agrawal and Tim Shen for their support.

Thanks to the staff at Prentice Hall for their professional assistance and guidance, especially Jeffrey Pepper as executive editor. Thanks also to Mary Sudul as production editor and Christa Carroll as editor for their top quality work.

This is the first book on PATROL. In fact, I believe it is the first book on application management as a discipline. I am pleased to bring you a book on the topic that has filled my life for the last several years and I trust you will enjoy it.

Foreword

Application management as a discipline in Distributed Systems Management basically started in the late eighties and early nineties. At that time, the only management available for enterprise IT was network management with products such as CABLETRON Spectrum, HP OpenView, and Sun NetManager dominating the market. These products were focused mainly on management of the infrastructure and hardware. However, this was also the time when distributed computing with large distributed databases and applications started to come to market and bring with them ever increasing complexity. Customers started to realize that management of the networking hardware and infrastructure was about 20% of the problem, and management of distributed software components such as the databases and applications was what was really needed. From that need, products such as PATROL were created.

Martin Picard conceived the PATROL product concept in early '91. If you look at the industry trade press around that time, you would stumble across some exciting stories about how exactly the PATROL idea was conceived during Martin's unprecedented sailing journey from San Francisco to Sydney (Fisherman caught an idea. *Information Week*). In reality, with his earlier involvement with Oracle as Technical Director of SQL Net, Martin was uniquely positioned to understand the evolving need for managing large scale distributed Oracle databases. PATROL was initially thought of as a distributed database monitoring system.

I joined forces with Martin in the very early stages of the product design. As we were brainstorming the product concepts and ideas, we realized that the RDBMS was truly only one part of the puzzle. The application was delivering the end user benefits. So in reality the applications were the most critical components to be managed. Of course, all the underlying components such as the RDBMS, Operating System, hardware platform, and the network had to be monitored and managed as well. The concept of generic Application Management and PATROL was born. It was an exciting time.

The first version of PATROL shipped in early '93. A number of major enterprises were doing initial deployments of the product. By the end of '93, we at Patrol Software realized a need for a broader distribution channel to reach all the customers that PATROL was destined to help. The best option appeared to be a merger with BMC Software, Inc., which was completed in January of 1994. Since that time, PATROL has shined and evolved into a best of breed solution for Enterprise Application Management. PATROL is still the flagship product of BMC, leading the company to new highs as the core of BMC's Application Service Assurance strategy.

I see Application Management today as a discipline within the broad distributed systems management arena, including areas such as performance, event, alert, and problem management. But more importantly, Application Management deals with the operational aspects of applications, ensuring applications are up and running and doing what the customers expect them to do. As the distributed market has evolved and expanded, PATROL has broadened into a fully heterogeneous solution for management of any application a customer may have deployed or developed themselves. PATROL provides the foundation platform for Application Management in any environment, for everything from Oracle Financials and SAP R/3 to in-house applications.

Looking ahead, applications will continue to evolve and the market will become more and more complex. IT organizations will continue the trend of deploying distributed applications in heterogeneous environments. Customers will buy applications for different parts of their business, wherever they choose. The same trend happens right now with ERP applications, where we see cases of multiple ERP solutions within the same site. A tremendous need exists for a solution that will tie this all together and help customers manage it. So the fundamental concept of PATROL and distributed application management is rising to the next level—to more complex environments, more involvement with the Internet, and more moving parts.

The key that PATROL brings to help resolve this complexity is the core architectural decision that we made early on in the product design—the separation of the knowledge from the infrastructure. This makes the introduction and management of new knowledge easy. This design point remains the strongest point of the product, and this is what is going to ensure that PATROL stays in line with the new complexities emerging in the IT world. The patented truly scalable PATROL architecture is still recognized as the most flexible and robust in the enterprise management world.

I am pleased to see this dynamic market brought to life in this book by one of the original PATROL developers. David Spuler's important contributions to PATROL include reworking PATROL Script Language into an advanced language for PATROL 2.0 and delivering the scalable 3.0 architecture. Over the years, Dave has written an extraordinary number of articles, white papers, and publications about PATROL and Application Management in general—the best pieces I have seen on this topic. This is a very exciting book that I am sure you will enjoy.

Kirill Tatarinov,
Vice President, Strategic Planning and Research,
BMC Software, Inc.

Table of Contents

Foreword

Preface

PART I. Application Management Theory

- Ch. 1. Introducing Application Management
- Ch. 2. Application Management Architecture
- Ch. 3. Availability Management
- Ch. 4. Problem and Event Management
- Ch. 5. Performance Management
- Ch. 6. Service Levels and Capacity Planning
- Ch. 7. Application Instrumentation Standards
- Ch. 8. Future Trends in Application Management

PART II. Application Management Practice

- Ch. 9. Introducing PATROL
- Ch. 10. PATROL Architecture Overview
- Ch. 11. Case Study - BMC's Service Assurance Center
- Ch. 12. Business Application Management
- Ch. 13. Database Management
- Ch. 14. Implementing PATROL
- Ch. 15. In-House Applications
- Ch. 16. Application Instrumentation
- Ch. 17. Integration with PATROL
- Ch. 18. Building Console Integrations
- Ch. 19. SNMP Integration

PART III. Knowledge Module Development

- Ch. 20. KM Development Overview
- Ch. 21. KM Efficiency
- Ch. 22. PSL Performance Measurement and Tuning
- Ch. 23. KM Debugging Techniques
- Ch. 24. Common KM Development Pitfalls
- Ch. 25. KM Portability
- Ch. 26. KM Compatibility
- Ch. 27. Advanced KM Architectures
- Ch. 28. KM Environment Variables
- Ch. 29. Nested Instance Hierarchies
- Ch. 30. PSL Libraries
- Ch. 31. SNMP KM Development Features
- Ch. 32. Advanced PSL String Operations

Bibliography

Index

Application Management Theory

Introducing Application Management

The promise of application management is to bring information systems management to a new level of effectiveness. This offers immediate benefits for management throughout all the layers of the application stack for vital business applications. Application management aims squarely at the mission-critical business applications, to improve their availability and performance. In a typical corporation, the applications that have become mission-critical include:

- Enterprise Resource Planning (ERP) applications
- Web servers and online internet/intranet applications
- Groupware and collaborative applications
- Email, calendars, and messaging
- In-house developed business applications

Application management brings systems management focus to bear on these applications as a whole. Underneath this application-level focus is analysis and management of all the lower-level technology layers such as servers, databases, middleware, and transaction processing. Application management aims to focus on the top levels of an ERP application like SAP R/3, rather than the lower-level fact that the ERP application actually runs using Oracle on Windows NT or Solaris. This fact is important to achieving higher-level management, but the aim is to merge lower-level management information into a higher-level management view.

1.1 Application Management Defined

Application management is a relatively new discipline that raises the level of management to the application layer. This is a trend driven by the rise in importance of distributed applications,

ERP applications, and internet applications. These are all becoming 365x24x7 online applications. Application management is currently being adopted by an increasing number of corporate information technology departments to achieve superior operational efficiency and higher availability. By bringing the level of management techniques closer to the overall business application, rather than focusing on the underlying technologies, the returns from application management are there for those who adopt it. Application management is the focusing of the traditional management techniques toward the application. This creates a new IT management discipline that is similar to traditional disciplines, but also has a number of new facets.

Application management is the optimization of application availability to end users. This involves ensuring that the application stays up, monitoring the application for failures or performance problems, and tuning the application for adequate end-user performance. The bottom line is that the application is what is critical to the business process, and management at that level is the best way to ensure its availability.

The key term when considering application management is proactive management. The adoption of application management leads to a more proactive management approach, and reduces the level of reactive, fire-fighting, emergency action by IT personnel. This proactive approach frees personnel from a stressful water-treading situation to one where their talents can be used to bring further advances through intelligent planning and re-architecting for a better overall situation.

A number of other forms of “management” are known to corporate IT departments. They go by terms such as “systems management” or “network management.” Historically, the issues of managing the machines and their operating systems have been central to business operations. The network, too, is of central importance, and network management has arisen as a part of business life. Network management involves monitoring all network components and troubleshooting problems with network availability and performance. A great many modern software tools are related to network management. Application management takes these ideas and raises them to the top level as seen by the business users rather than the administrators.

1.1.1 Origins of Application Management

Application management has arrived as a modern IT idiom rather recently. It was a nascent concept in the early ‘90s and has since become a clear trend as major vendors stepped in. The idea of application management came about because of a number of trends in information technology that have raised the importance of the application software. The increasing reliability of machine hardware, operating systems software, and network components has made systems and network management less of an issue. Or perhaps the extra years of experience that the industry has in the management of systems and network issues has made these issues less problematic. Now the recent prominence of the application has brought with it management problems.

Application management is not really a revolutionary idea. More accurate would be to call it an evolutionary idea whose time has come. It is an application (forgive the pun) of the same traditional IT management improvement ideas to the next level of IT software: the application.

After all, “network management” and “systems management” were once new terms, but have become standard IT requirements. The discipline of network management has many commercial tools to help achieve it. The SNMP standard was a watershed in the extent to which it brought the area of network management onto a stable and consistent platform. All this, and yet “network management” was once a new term with unclear implications.

So application management is management applied to applications. The long-term implications for the discipline are once again unclear. With which of the various standards will the application vendors have their applications instrumented? SNMP was also once just one of many standards. However, one thing is clear: the goal of application management is to bring the advantages of coherent management to critical business applications.

The scope of application management is clearly limited to applications for corporations or organizations. Personal software is generally not so complex that management is required. It has no need to run 24 hours a day. Reliability is not as critical, because if an application crashes during use, it can be restarted easily. Because of their importance and complexity, corporate applications require management.

Corporations require applications to manage the huge volumes of data generated by their business operations. This need in turn raises the importance of the application, and the management of the application becomes central to smooth business operations. Businesses require application management because of the mission-critical nature of the application. Availability is an increasingly critical aspect of an application. More and more businesses are offering applications that are available online 24x7 to end-user customers and to business suppliers and dealers. This trend also raises the importance of the application.

Application management arises because of the inherent complexity of business applications. No need to “manage” applications would exist if they were simple beasts. But business applications are behemoths in the modern corporate world. Downed applications cannot be as easily restarted as personal software because they involve masses of data and numerous client machines on the network. Tuning a poorly performing application is no simple matter because the application itself is made up of dozens of components involving databases, files, networks, and processes. This sort of pain made the issue of managing applications so important that “application management” ultimately became an IT discipline in its own right.

1.1.2 Defining Application Management

The term “application management” has become overused in recent years. A variety of definitions come from industry analysts and vendors. The key differences in these definitions revolve around the “static” and “dynamic” tasks in making an application work.

The “static” issues are themes such as application distribution, deployment, and inventory management. Some definitions combine these disciplines with dynamic management under the umbrella term of “application management.” Although these issues are important, they fall into the more general aspect of systems management in terms of managing the software issues of a

system. These static issues are not our main discussion here and I'd say they are not all that useful in offering quick fixes towards improving the management situation for applications.

The other side is the “dynamic” aspects of managing an application. This refers to taking an application that is running and quickly making it perform better. This is real life for a database or application administrator and the rewards are immediate. Application management as defined here is the general issue of optimizing application availability, which involves specific issues such as application monitoring, problem correction, and performance tuning. In other words, when we refer to “application management” here, it means all the dynamic aspects of a running application, across the entire enterprise.

1.2 Application Management Challenges

A number of obstacles typically stand in the way of effective application management. Sometimes these problems leave management resources at the mercy of the users who are the first to discover the problems. Without a good solution, management personnel can be drawn into a totally reactive mode of fighting fires as they arise. True management is impossible unless these issues are handled. Some of the clear categories of management challenges include:

- Heterogeneity and diversity of the environment: many varieties of software and hardware
- Complexity of the software and hardware that are in place
- Limited IT resources from budget constraints
- Limited IT skills from hiring or staff retention issues
- Increasing demands for application availability from business requirements

All of these issues tend to bring about a management situation of reactive response to problems at the various application levels. The cost of managing the applications increases because of the complexity of the overall environment. Taking the time, or rather, *finding* the time, to plan a clear solution to these issues is the only way to achieve a more proactive and relaxed situation.

The heterogeneity of modern information systems requires the management of many diverse applications. This involves both commercial off-the-shelf products and in-house developed applications. This situation arises in an organization from a variety of unavoidable events. Standardization of hardware and software acquisition is not common. Different personnel are responsible for hardware and software requirements for independent departments and make different choices. Differing requirements and commercial situations at different times lead to inconsistent purchasing decisions. Legacy systems just never seem to be discarded and sit alongside the modern open systems projects. Company mergers and acquisitions throw together entirely different networks of hardware and software, and usually expect a single information systems organization to handle both. In any event, corporate information managers are usually stuck with the task of managing a mishmash of different hardware, operating systems, databases, and applications.

Overcoming the challenges of this diversity of information infrastructure resources is a requirement of application management. True application management must cover all of these bases. This includes everything from the hardware and network through to databases, middle-ware, and the applications. Achieving 24x7 monitoring at the application level requires consistent handling of all the layers that build up the infrastructure for a software application. This consistency has to be achieved by the application management tool in spite of an almost total lack of standardization of management interfaces to all of the database and application software components.

Diversity is just one challenge; complexity is another. Not only does a variety of system components exist, but each component also is quite complex in its own right. This situation applies to both hardware and software. All of these components have their own little quirks to manage and specific points of failure. Furthermore, standards compliance is not really much help even where it does exist. The only solution seems to be maintenance of a staff of personnel with varying skills to cover all the components. Typically, this means a very shallow coverage of most items with some depth in the areas where the current team has strengths. With the increasing budgetary pressures for information technology departments, the likelihood of both a broad and a deep skill level is not high. And keeping the desired level for any length of time is difficult in the competitive employment market. Corporate managers must make do with the resources and skill levels at hand or pay premiums for specific skills if their budgets permit.

Overcoming the issues of both diversity and complexity is a key tenet of the role of application management in an organization. Application management aims to serve as an abstraction layer that simplifies the management of diverse software packages. The impact of heterogeneity is reduced by a consistent management interface across all different packages. Complexity is also reduced because of the simplified models that are used to represent the objects and management tasks. Application management encapsulates the complexity of an application into a simpler model of reality. Complexity is replaced with simplicity in true application management.

The distinction between complexity and heterogeneity is sometimes difficult to make. However, a good example appears in the vendor-specific application management solutions. The vendors of individual databases and applications are well aware of the need for application-level management and are beginning to offer base-level features in this area. This may solve complexity but it does not solve heterogeneity. Each vendor may reduce the complexity of its own database or application, but not that of its competitors. Manageability is a competitive differentiation point and no coming together has occurred to attempt to reach a more consistent standard. The problems of managing diverse and heterogeneous applications and databases still remain for the IT departments making use of these localized solutions.

Automation through application management is a necessary solution because of the variety of pressures on corporate information systems departments. Budgetary constraints are so commonplace they are hardly necessary to mention. The competitive nature of the modern business market pressures IT departments to maintain efficiency and productivity levels whilst still keeping down costs so as not to affect the quarterly profitability announcements to Wall Street.

These shrinking dollars have to be split between the demands of hardware, software, and the ever-demanding software experts. The complexity of applications leads to lengthier training cycles in the various types of software, which in turn lead to a scarcity of staff with particular skills. Staff do not have the time to attend the multitude of vendor-specific training programs. And those that return from such programs may suddenly discover their own worth in the competitive hiring market. All of this constraint and yet those applications still have to stay running round the clock because they are the lifeblood of the business.

1.3 Competitive Advantage through Superior IT Service

Despite all the pressures, the corporate IT department is in the proud position of centrally supporting the business. The applications are key to the corporate bottom line, and suddenly IT is as much about revenues as it is technology. It is competing via computing. The problem of managing the availability of the applications is central to the business, and the personnel that perform this function are key players who help achieve corporate goals.

Better management of applications is a competitive advantage. The applications that are available the most lead to more satisfied customers and fewer delays in business processes. Online customer transactions and point-of-sale applications must stay live for the business to continue. Order entry and inventory databases have to be available to have a streamlined business process for moving merchandise from material suppliers through to product distributors.

Data requirements are also growing, and the management of this data and the databases that contain it is also a business issue. Retail companies want to store detailed transaction data to analyze trends in customer buying patterns. Individual purchases and user transactions mean even more detail that must be stored as data in the database. The gathering of this data and the availability of the business reports arising from such data warehouses is a competitive advantage, allowing more informed management decisions.

Delivering an improved level of manageability for applications is a key method whereby the desired competitive advantage occurs. Application management improves competitiveness in both the short and long term. The key short-term benefit is increased availability from reduced unplanned downtime. This occurs via a paradigm shift from a reactive management style to a proactive mode around early detection, notification, and correction. The longer-term benefits arise through greater alignment between IT and business goals. This occurs through the higher-level focus on the business application as a whole rather than the underlying technical infrastructure.

1.4 Centralized Management Paradigm

Centralization of management issues rises as the clear solution to the tug-of-war between scarce human resources and the expanding numbers of computing components. This centralized management area is the client/server equivalent of the “glass house.” Even if the actual hardware is not centralized, the management issues can be.

The issue of centralization arises in open systems through the distributed nature of the components of the applications. The centralized mainframe architectures of the glass house are replaced with dispersed technology, and usually also dispersed management personnel. Centralization aims to bring some sanity to this situation by offering a coherent view. In some cases, this centralized view is a complementary addition to decentralized management departments, and in some cases it can serve to bring them together, at least for the first-level help desk.

Centralization offers many advantages for application management personnel. Instead of running ragged to the various distributed components, the management issues are brought back to a central point of control. This approach allows focused work from one physical place within the enterprise while still addressing problems throughout the network, hardware, and software infrastructure. Although the physical need to go directly to the site still exists for hardware and network failures, the majority of software failures can be addressed from a remote site. No longer do the databases and the Database Administrators (DBAs) need to have so close a tie. The heavy hardware running the databases can be housed in secure, climate-controlled environments and accessed by database and system administrators from their desktops.

Ideally, centralized application management brings everything back to a single console. The support issues of dealing with a problem are handled through a central service desk. The first level of handling a problem can be centralized fully. The operator can handle problems from the full range of hardware and software failures, and forward escalated problems to more specialized administration staff as appropriate.

Centralization need not remove a decentralized administrative organization. Indeed, the typical organization has varying departments, and their administrators may well be distributed throughout a site, a country, and across country boundaries. Centralized management can augment this scheme by providing a safety net for the management of these resources. For example, it can help catch problems that occur outside normal business hours. This net can be added centrally for relatively little extra cost when compared to the number of applications that may benefit. The issues of application management are shared between the local administrators who handle the majority of issues and the centralized help desk that deals with unexpected and urgent events.

1.5 Business Application Layers

The typical business application is a very complex beast. It is built up out of a number of layers of infrastructure and application logic layers. Anyone will understand that when a network wire is cut, the client/server application is disconnected from its users and therefore disabled. However, this layer is just one of the many layers that can fail. The network is just the most familiar because of the historical development of networks and network management software. The application itself is vulnerable to failures of any of its logical layers, and true application management requires the monitoring of the availability of all the various layers:

- Hardware
- Network
- Operating system
- Database
- Middleware—messaging, transaction processing
- Applications—ERP, email, web, in-house applications

The machine hardware and network are the lowest physical layer. The network itself consists of hardware components and some software-like capabilities in routers, bridges, and hubs. The management of the network layer is well understood and covered by various network management tools.

The operating system is the software layer on top of the computer hardware. This provides all the services to physical resources such as CPU, memory, and disk storage. These basic resources are similar across all operating systems such as UNIX, Windows NT, VMS, and MVS. System monitoring needs to monitor critical resources such as CPU usage, memory thrashing, and the availability of disk resources. Clearly, the crashing of a system is a critical event that needs to be detected by monitoring software. All these resources are critical to the functioning of the database and application layers.

The database layer provides advanced data storage, organization, and searching capabilities. This is a hugely advanced area of software technology. Databases have grown in leaps and bounds in terms of the sophistication and scalability of data processing.

The middleware layer includes a variety of middleware components. Some provide network transport models to move data between client/server components. Other middleware types provide interfaces between the different database and application components. Transaction processing systems can also be loosely considered a component of the middleware layer in that they interface between the clients sending transactions and the databases executing them.

The application layer is the topmost level of functionality for business requirements. This is where the business processes are encapsulated in software form. The logic of the application is the closest to the model of the business requirements of the organization. The data is typically stored in one or more databases, and the application logic understands what the meaning of this data is, and how it is processed.

An effective application management organization requires coverage of all the layers. By monitoring both the availability and the performance of each layer, any downtime or performance bottlenecks can be detected and corrected.

1.6 Application-Level Focus

A clear goal of application management is to bring an application-level focus to the management of corporate IT resources. This focus at the high level of business applications is valuable in achieving both the financial and technology requirements of the company. The financial benefits arise from the alignment of IT with the financial requirements of the business. Technologically,

application management brings a more holistic approach to dealing with all possible sources of problems rather than only focusing on the underlying system hardware and operating systems.

The technology advantages arise not only because of the corporate reliance on the application but also because of the complexity of the application itself. This complexity leads to an increasing proportion of application failures that arise within the application itself. The increasing reliability of the underlying hardware and network infrastructure means less frequent failures at that level. Instead, the software failures in the database or the application become more common. For example, the failures such as unavailable tablespace extents or deadlocked internal transactions are more software-oriented failures. Although they may be loosely coupled with the filesystem and the underlying performance of the hardware, they are not likely to show up as low-level failures. Monitoring the database or application directly at the software level becomes necessary. Although this must still occur in concert with low-level infrastructure monitoring, the higher levels of application management are requirements to achieve proactive notification and resolution of software-level problems. The application software has become a layer in its own right, and this forces the management disciplines to evolve.

The financial and business impacts of the prominence of the application are also key. The application level focus of application management is important to financial health of the business because of the closer attention to the underlying business process. The company executives are not likely to really understand how the failure of a router or a CPU affects their business. But they will understand if a business application ceases to be available, and they are losing real, live customers with every passing minute. Bridging this separation between the business principles that require IT resources and the technical aspects of supporting these IT resources is the goal of application management. This involves mapping back the low-level technical aspects of the IT infrastructure to an overall view of the health of the business itself.

Consider the case of online banking transactions. Events need to come from the perspective of a business focus rather than a physical infrastructure focus. Alerts should be initiated based on the number of customers whose business transactions fail. Performance alerts need to be tied to real customer queue lengths rather than CPU usage percentages. These are high-level business process measurement and alerting issues.

Achieving this goal requires the use of a logical view of all the physical components that make up the system infrastructure. Which business application or applications does a particular network or CPU component affect, anyway? The business view of the infrastructure should answer this question.

This form of logical view is the state-of-the-art of modern application management. Objects from the various lower-level components are grouped into a logical application view of the business processes that underlie the real problems of a corporation.

1.7 Return On Investment

Although it may appear superficially to be technology oriented, Return On Investment (ROI) is the bottom-line need for any application management project. Business advantage is the only

reason to even consider application management. Why assign precious resources, if not to get a significant return? Without clear and relatively short-term ROI, the resources have no point.

So what is the ROI for application management? Clearly the single greatest advantage from application management is reduced downtime of critical applications, and thereby less interruption to business processes. No lost orders or slowed business data processing occur because of these outages. By preventing downtime, application management does not add revenue per se, but prevents any loss of it. The extent to which this prevention is applicable depends on the industry that the application is supporting. Table 1-1 shows an estimate of the hourly costs of an outage for various industries.

Table 1-1 Hourly outage costs

Business Cost	Average Hourly Cost Impacts
Credit Card Sales Authorization	\$2.6 million
Brokerage Operations	\$6.45 million
Catalog Sales	\$90,000
Infomercial 800 Promotions	\$199,500
Pay Per View Services	\$150,250
Telephone Ticket Sales	\$69,000
Airline Reservation Centers	\$89,500

Source: Disaster Impact Research Report, Contingency Planning Research, Inc., March 1995.

A second key advantage of application management is the cost-effective use of existing skills. The skills of application management personnel can be fully leveraged to more complicated problems because the simple problems can be corrected more easily. Indeed, staff can more quickly learn to manage applications for which they may not have had past experience or training. The value of a tool to an entire group can be more than the value of adding a new employee. The learning curve and training costs are reduced, and staff can more quickly achieve more rounded value to the company. The value of centralized application management is to place a consistent homogeneous representation around the various heterogeneous components of the enterprise. This simple model is usually graphical but can also be as simple as a centralized list of all critical events in an event management viewer. This homogeneous view of all IT

resources is a critical advantage, given the modern difficulty of finding personnel who are deeply skilled in all the various software packages and technologies. Through standard notification sequences and, in some cases, automated correction of problems, the first level of support can be more effective and make less use of the time of the more specialized administrators. The time of the advanced DBA or System Administrator (SA) can then be put to better use on more business-critical tasks such as designing future enhancements to the network infrastructure or business applications.

1.7.1 Intangible Benefits

Both of these benefits of reduced downtime costs and personnel effectiveness are relatively tangible values from incorporating application management. A variety of intangible benefits arise that cannot be seen as easily by top-level analysis of financial numbers. Perhaps one of the most difficult reasons to justify, and yet one of the most powerful advantages, is the value of being ahead of the curve. Incorporating an application-level focus into IT management brings the technology and financial groups into closer alignment. The chasm between the technology and the business operations is narrowed. This can be only advantageous to the company, and yet to measure this value is impossible. Both groups start to see the value of the business and its relationship to the applications. A more singular direction of progress becomes clear for all areas of business operations support. This closer relationship and better understanding can only lead to a fuller understanding of the need for and value of the IT department by the top levels of the company. This garners support for the IT department and opens the way to a better understanding of how applying resources in this area can improve the overall company bottom line.

Even the value of decreased downtime (discussed above) has many extra intangible benefits. The value of striving toward the 100% availability goal is not just that no orders are lost. This is certainly the clearest financial advantage, but others permeate the entire business. Confidence is one such advantage. Customers, suppliers, and distributors come to see the quality of the information processes. Customers have rapid and uninterrupted service, and perceive the company as a reliable place to service their needs without any unpleasant waiting or surprises. Suppliers and distributors have fast access to accurate data, because the departments with which they interact have a reliable infrastructure to gather this information.

This smooth ride of the most critical corporate resource, information, can lead to other overall business advantages. Time to market of products can be improved. New business projects, particularly those relying on technology such as new remote offices, can be delivered earlier and without as much drama about the technology aspects. This possibility makes the corporation more competitive as it opens its doors to the customers. Such advances in business process are possible with a more flexible and efficient IT department. Although they may not get much of the credit, a lot of the competitive advantage rides on the IT personnel who deliver the applications and maintain their high availability.

Reductions in downtime also affect the IT department in a very positive way. Fewer emergencies and unpleasant late-night sessions occur or 3 a.m. beepers go off. By adding application

management, proactive resolution of a lot of problems occurs that would otherwise have escalated into problematic events. This further frees personnel to even more proactive tasks, such as planning for future problems as the business grows and brainstorming solutions to existing difficult problems. Talented people deserve to spend the time using those talents on such problems rather than on repetitive emergency maintenance. They can then attend to the information requirements of the growing parts of the business, and spend less time on maintaining the systems that should already be working. The IT department should be free to strive toward advanced goals that showcase its corporate value clearly, rather than a problem-oriented organization whose only recognition is from negative occurrences such as downtime. You sometimes hear that the best IT department is one that you never have to call. Although this statement may be true for problem management, the IT department can also play a prominent role in advancing areas of new business to higher profitability.

1.8 Heightened Asset Control and Resource Utilization

Another clear benefit from application management is the increased control over both hardware and software assets. The more focused measurement that comes with application management offers the opportunity to more tightly control these resources. This control brings with it several advantages in terms of resource utilization and productivity.

Software assets are the primary focus of application management. With the application-level focus of application management comes application-level measurement. This is highly valuable information about the usage patterns of the application. Application software license usage can be measured against the actual user patterns. Having a printed graph displaying the actual number of users over the last week is valuable information in negotiating license agreements with application or database vendors. Since these numbers come directly from monitoring the database or application at its own level, through its own APIs and interfaces, the numbers are undeniable.

These usage measurements are valuable not only for vendor discussions. The information is also useful for observing patterns in application usage that might offer information on how to reduce the load, leading to further productivity and again ultimately to fewer licenses. For example, an application management tool might observe that the license usage peaks only at one time during the week, and analysis of the postmortem information might reveal that the reason is the combination of a data warehouse integrity application and a reporting application. By separating these activities, such as by timing them slightly differently, the peak can be smoothed, and the overall requirement for access to the database or application is reduced.

Application management also brings with it opportunities for greater hardware resource control. The measurements associated with application management are sources of useful information to determine the best usage of hardware resources. Clearly, concurrent access to servers can be measured exactly as already discussed for application software user measurement. Access to a hardware server can be analyzed over time to determine patterns in server access

such that the peaks can be spread out to gain better utilization of hardware and operating system resources.

Performance analysis also helps in overall hardware utilization. The analysis of the performance of a database or application can show how to better utilize existing hardware and still improve performance. Database access and cache hit rates can show that one particular tablespace is the highest cause of performance problems, and partitioning it differently can lead to better hardware usage, and ultimately reduced long-term hardware costs.

1.9 Leveraging Existing Investments

An issue related to Return On Investment (ROI) is making the best use of money already spent. This applies to both people and technology. Leveraging people investments is all about ensuring that their capabilities are expanded by a new application management investment. Leveraging existing technology infrastructure is all about ensuring good ROI on past projects. Throwing away the investment in network management infrastructure to add application management would not make financial sense. Fortunately, this doesn't make technical sense either, since network and application management projects can usually complement one another.

Like any technology investment, an application management project should make good use of existing infrastructure. You should not need to discard any existing hardware, operating systems, databases, or applications to add application management. Adding application management to the existing technologies makes far more financial sense. For example, which would be cheaper: to replace one set of database servers with another brand because of advanced application management, or to add application management to the existing database servers? This answer is obvious. The full cost is huge for replacing an entire set of database servers, redeploying the new ones, revising any applications or business processes based on these databases, and so on. The addition of application management, on the other hand, is a smooth deployment that is outside the existing databases and does not affect them, and could even be rolled back without damage to the databases if some unforeseen impact occurred during the project. Changing the infrastructure levels is a radical move, whereas adding application management is a low-risk addition to the environment.

In-house developed applications are also a very important form of existing investment. The management of these applications is critical to an overall application management success story. Leveraging these types of investment is also necessary, and some form of application management needs to be carefully considered. In particular, an intrusive form of application instrumentation to achieve application management is not making use of the existing investment. An intrusive solution requires a full revision of the application itself, from development and testing through to redeployment on production servers. A better solution is to add non-intrusive application management, which does not require an application development revision, and typically does not even require the application to be brought down.

All of the above discussion assumes the presence of an existing infrastructure that needs manageability in the most cost-effective manner. The other side of the coin is that revision of

this infrastructure is always an ongoing process. New hardware, databases, and applications are constantly required to improve the overall business performance. These might take the form of projects to revise existing infrastructure or projects to expand the information infrastructure into new business areas. Two other major application revision catalysts are Year 2000 and European Monetary Union upgrades. In any case, these new projects are also a form of existing investment to which application management can be added. Application management can be added to the rollout of the new databases and applications. This possibility is an important consideration to make at the start of the project when the application management can be merged into existing deployment plans. In-house developed applications, if caught early enough in the development process, can have intrusive instrumentation added, although this may increase the overall development costs. A reasonable consideration for application management of in-house applications is often to use a non-intrusive method for the first release of the application and move to an intrusive approach as development resources permit.

The rollout of a new IT project thus has a key component in terms of ensuring the manageability of the new infrastructure. This management must occur at all levels, from the network, hardware, operating systems, databases and application layers. The topmost level of the application layer gives the closest view of the impact on the business concerns. The lower-level failures can be mapped into an application-focused view of the enterprise. In this way, the investment in the rollout of a new project is maximized by ensuring adequate management of the applications that are the ultimate reason for the rollout. The overall return from a new project investment is ensured by avoiding the rollout of an unmanageable application.

1.10 Removing Existing Resource Drains

The opposite to leveraging existing investments is eliminating the costs from ongoing redundant investments. The best example where application management applies is the elimination of ongoing maintenance and support for inconsistent and redundant monitoring and management efforts at the various application layers. Although leveraging existing investments in hardware, networks, databases, and applications makes sense, does leveraging the ongoing cost of supporting this in a heterogeneous manner still make sense? In an organization without formalized application management, different approaches to network, server, database, and application management will typically exist. These are really a spread-out form of multi-layer application management that causes too many development and procedural efforts to be duplicated, and keeps many staff in a reactive mode of fighting their own localized problems with localized development extensions. A centralized formal program of application management taking into account all layers takes a lot of pressure off all these layers, and frees personnel resources for more productive tasks. This centralizes the efforts to develop and customize an application management solution for the full enterprise and its many technology layers.

Formalized application management is a disciplined approach to centralizing the management of these various layers in the application stack. The aim is to centralize all problem management into a consolidated service desk, thereby at least alleviating the first level of support for

particular administrative groups. In practice, this often alleviates second or third levels of support, because by examining all levels of the application as a whole, the real cause of the problem can be identified. This eliminates a lot of redundant work by all groups when a downtime problem arises, and avoids unproductive finger pointing. Or rather, it starts pointing the finger in the right direction. The development of the overall program of application management can be focused on solving the overall issues to properly identify and resolve problems.

A clear solution to this duplication effort is to establish an application management group. By focusing the efforts on application management as a whole, the redundant development efforts can be avoided and the overall solution can move forward. Whether through a full in-house developed application management solution or the customization of an off-the-shelf solution, the importance is in the focus on the whole problem and not the parts. In practice, once the whole problem is seen in all its glory, so many parts to the overall problem exist that an off-the-shelf solution becomes the most cost-effective. The cost of support and maintenance of such a solution is immense, and it is not really the core business need for the organization, anyway. An off-the-shelf solution becomes cost-effective because the overall burden of development, maintenance, and support that is carried by the vendor is divided across the many IT departments that adopt the off-the-shelf product.

A key part of formalizing this process is to make sure that you do leverage existing IT creations. A distinction exists between the creations from the work, which are worth keeping, and the actual work itself, which you want to reduce or eliminate. The aim is to eliminate future costs of maintenance and support, but still retain the gems that were built as part of this cost in the past. For example, powerful administrative commands may be already written in script form (e.g., shell script, Perl, SQL, TCL, VB, etc.). The aim of introducing a formal application management program is to take the pressure off future maintenance, while still merging these existing specialized development assets into the overall solutions. Therefore, a key point of an application management solution is that, although it must be general enough to take over the maintenance of the layers, it must still be extensible enough to permit inclusion of already existing work.

1.11 Acting on Application Management

Organizations that act on application management can gain superior returns from their technology investments. The first step is to raise application management onto the corporate agenda, if this is not already the case. One plausible initial step is to justify and create a group responsible for application management throughout the enterprise. This group can make an assessment about the current state of affairs and what is needed to move onto the next level.

The advantages of application management can be achieved through a number of avenues. Application management can be added to an existing software infrastructure to get the higher-level benefits from managing at the application level. Application management can also be added at the start of the rollout of a new IT project to ensure that the newly rolled out applications will not become a management nightmare.

Fortunately, application management is already more than just a nice theory. Mature application management tools offer these benefits in practical ways. These can be deployed to get the benefits of application management in the short-term. In the long-term, other issues exist, such as the involvement of the in-house application development staff in order to instrument their applications for advanced management.

Application Management Architecture

The architecture is the technical means by which the desired goal is achieved. Successful application management requires an architecture that achieves the goal of centralized management. Clearly the realization of this goal involves not only the architecture of the application management solution, but also the underlying architecture of the application itself. An intricate relationship exists between the two architectures, and one cannot be designed without the other.

The architecture of a client/server application is a massive topic that is a familiar problem for corporate information systems architects. Although it is really the domain of an entirely separate and much larger book, we can examine some of the common issues. For example, decisions are required regarding server hardware, client hardware, networking connectivity hardware, server software, client software, database software, application software, middleware software, network protocols, and so on. These options are affected by the type of project, be it a real-time transaction processing system, a groupware implementation, or a web-based application. All of these various components must be tied together into an architecture that satisfies business goals whilst retaining the capability for effective management. The application is not going to be long-standing if its availability cannot be guaranteed by rapid resolution of failures at the various levels.

2.1 Fundamental Architectural Alternatives

The basic goal of any application management solution is to centralize the management issues. This can be achieved in numerous ways involving different architectural components and the communications mechanisms between these components. Some of these components have become so well understood that they have their own names (e.g. agents, consoles, etc.), but first

let us take a step back and look at the underlying problem. This problem is the centralization of the management issues for a number of distributed machines and the software that they are executing. Really, the solutions clearly abide into two basic classifications: management from the outside, and management from the inside.

Management from the outside involves no changes to the managed server. No new management software component needs to be placed on the server, and no changes made to the server. How then is remote management to occur? Well, if truth be told, it does require something on the server. After all, without a network daemon on the machine, how can the machine communicate with the outside world? However, these things are not components designed for application management. Application management from the outside involves using that which is already present. The centralized application management component must connect to the remote machine using the existing daemons. For example, the telnet and ftp client executables could be used to connect to the managed server. One can imagine a centralized management “console” that repeatedly uses telnet to connect to the machine and then run commands to determine its status (see Figure 2-1).

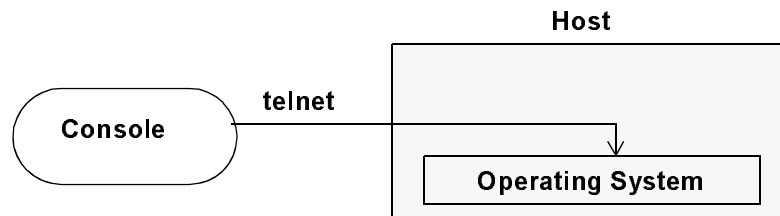


Figure 2-1 Console using telnet

However, certain disadvantages of this setup soon become obvious. The performance in terms of network traffic is poor, because of the bottleneck at the management console once this console is connecting to multiple remote machines via telnet. Also, the capabilities of the management tool are quite limited, because use must be made of the existing executables or commands. If no command determines the required information, such as kernel-level access, this data collection cannot be done via telnet. This limitation leads inexorably to the need to place some special commands on the server, and this leads into the second category.

Management from the inside refers to placing additional application management components on the remote managed server. The advantages of this approach quickly eliminate the idea of outside management, except in very rare cases. The sophistication of the management solution is greatly improved by the flexibility to control, in a sense locally, what is occurring on the

remotely managed server. The simplest form of this management from the inside is a small command (see Figure 2-2) that is used to augment the existing commands.

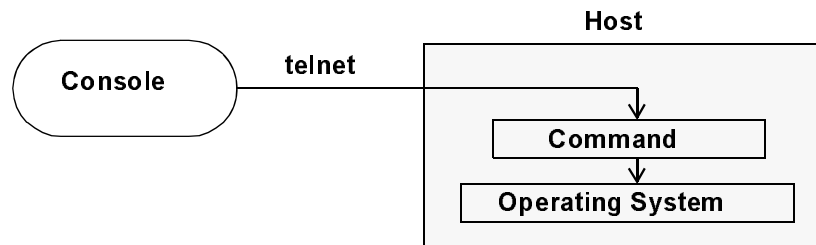


Figure 2-2 Console using telnet plus an additional command

A console could still connect via telnet or some other method, just to run this command. However, this approach still retains the disadvantages related to network traffic that cause scalability problems. These lead to the need for a self-contained management component that minimizes network traffic. The solution is what is known as an “agent.” This is a self-contained daemon process that handles both the connection aspects of the console-agent connection, and also augments the natural set of management commands on the remote machine. Some of the main differences in characteristics of an agent compared to a simple command include:

- Continuous execution—an agent runs as a daemon, whereas a command is ad hoc and user-initiated.
- Network methods—an agent is typically accessed using its own network support.

This agent becomes the most common architectural component for application management. Different types of agents are available, as we will further discuss. The autonomous agent is the most useful of such architectures but requires the agent to act in certain ways.

2.2 SNMP Architecture

The Simple Network Management Protocol (SNMP) and data model present an interesting architectural example. SNMP is characterized by having an entity on the managed node or device, called the “SNMP agent.” Although in practice, an intricate architecture of master agent and subordinate agents may exist, in concept, it is a single agent entity. The client side is the SNMP manager console. Therefore, SNMP appears to be the typical client-server monitoring architecture.

However, SNMP arose out of the need to monitor hardware devices, particularly network devices such as routers, hubs, and bridges. Because these components could not support sophisticated processing, the SNMP agents on these devices were of necessity quite simple. The capability to support low-level hardware devices effectively limited the model. Although several revisions of the SNMP standard have occurred, and are still occurring, the basic limitation of SNMP arises from its early beginnings in the lack of sophistication on the server side.

The SNMP architecture typically involves a form of polling architecture where the “smart” SNMP manager polls the “dumb” SNMP agent using a “get” operation via its Management Information Base (MIB) description. This is a “pull” model that involves a continuous stream of unnecessary data in order to achieve real-time monitoring. Although the SNMP trap does exist to give a form of non-polling event-driven or “push” capability, this is not enough upon which to hang an advanced event-driven architecture.

Perhaps future revisions of SNMP will offer more advanced forms of event-driven non-polling architecture such as publish-subscribe models. Time will tell whether the SNMP architecture evolves to meet the new challenges, or becomes a checkbox item for support of older hardware and software.

2.3 Web Architecture

The architecture of web browser and server is also of the typical client-server form. Indeed, the recent advent of the “Web-Based Enterprise Management” (WBEM) initiative is an advanced application management standard based upon this architecture—see “Web-Based Enterprise Management (WBEM)” on page 79. The web browser forms the ultimate console because of its free-form capabilities to support any display, and the widespread appeal of a familiar interface. However, the web server lacks some of the features that a better architecture requires.

In its current form, the web server itself is not particularly flexible. However, it is certainly capable of independent action via execution of server-side programs through CGI or other interfaces. In theory, therefore, a server-side application can perform monitoring and then inform the web browser in an event-driven manner. However, in practice, this style of web algorithm has not been widely used for management.

The problem is clearly that the original basic web architecture is a polling architecture. Users of web browsers have roamed the internet, connecting to servers to pull down the material. This model is the well-known polling or “pull” model. The advent of “push” web technology may well change that, but this is a relatively new technology at the time of writing. A “push” architecture is exactly what an event-driven monitoring architecture requires for effective implementation. Between the activity on “push” technology and the WBEM initiative, the implementation of web-based monitoring and management seems certain to evolve into fully autonomous agents.

2.4 Autonomous Agents

The autonomous agent is the ultimate in self-contained management components. Through autonomy comes the power to achieve more advanced monitoring architectures for application management. To achieve autonomy, you have three key architectural requirements:

- Local control
- Local execution
- Local storage

The agent must have a self-contained program to direct its actions, must be capable of execution of these actions locally without any remote control, and must store these results locally.

Autonomy of the agent is achieved via these three capabilities: local commands, local execution, and local result storage. Autonomy is actually unrelated to the network protocol that is used to communicate between console and agent. The agent can be autonomous under any protocol used for communication, provided that the three criteria for autonomy are met.

An autonomous agent can run in a lights-off management environment. No actual need for a console's presence exists, although in this case, it is typically connected to some form of notification through pager, email, or SNMP trap sending. The console is an option, and of course, a commonly used option, to give a continued graphical view to the user. However, the user can shut down and start up the console according to whim because the monitoring, corrections, and notifications are not dependent on that single component. This flexibility is a key advantage of a truly autonomous agent.

Another key advantage of an autonomous agent is the possibility of an event-driven rather than a polling monitoring architecture. Unlike the polling model of the SNMP manager and the "dumb" SNMP agent, an autonomous agent can raise events in real-time through its own actions. You do not need to waste time inefficiently polling the monitoring values from some console or client.

Scalability is another aspect of an autonomous agent architecture. By pushing most of the decision logic further down the chain, the flow of information and events to the top level is reduced. This reduction allows you to view the overall situation without having a scalability problem from a flood of data values.

2.5 Console-Centric Architectures

The opposite style of console-agent architecture to autonomous agents is a console-centric architecture. Under this style of architecture, the console connection is a requirement for the agent's management actions. The console becomes a central point of failure in this form of architecture. This style of architecture also inevitably leads to a polling or "pull" model and the inefficiencies of unnecessary data transmission.

Only one of the three requirements for autonomy needs to be absent for a console-centric architecture to result. Without local control, the console is required to tell the agent what to execute. Without local connection, the agent lacks scheduling autonomy, not knowing when to run any command, and again a console is required. Without local storage, the agent can collect data, but this data is effectively lost because it can never be viewed or gathered at a later point by a viewing or reporting tool. A console is needed to capture, monitor, or view this data.

While any one requirement would lose the autonomous architecture, a typical situation is that all three are missing at once. A console-centric architecture involves a console sending the commands to the agent across the network, controlling when they are executed, and receiving the results of the commands immediately from the agent via the network (see Figure 2-3). The

main scalability problem that arises is from the network traffic from both downloaded commands and the returned command results.

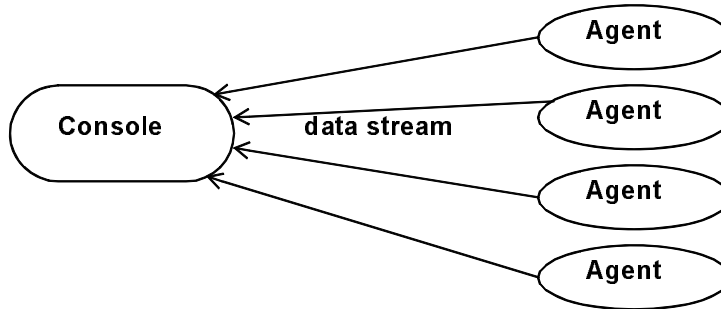


Figure 2-3 Console-centric agent bottleneck

Although this network traffic may be bearable for a single agent or a small number of agents, a clear limit exists. A console connecting to dozens of agents becomes a network traffic bottleneck. With all its limits, the main reason for console-centric architectures is that they are easier to create, because an autonomous agent is a more complex architecture to design.

2.6 Example: PATROL 3.0 Autonomous Agent Architecture

The history of PATROL is a classic showcase of the difference between console-centric and autonomous agent architectures. PATROL version 2.0 was a console-centric architecture. The release in 1995 of PATROL 3.0 introduced the Intelligent Autonomous Agent (IAA).

The versions of PATROL up to and including the 2.x series had a console-centric architecture. The agent would not start until a console downloaded a Knowledge Module, and then the agent would start pumping back gathered parameter values. Experience showed that this approach created scalability problems at agent counts below a hundred, the limit depending on factors such as the number of Knowledge Modules and the console machine capacity. Network traffic was too high, and the 2.0 console received too many data points from the agents and could not process them quickly enough, leading to console performance degradation. Hence, the PATROL 2.0 console was limited to significantly less than a hundred agents by its console-centric architecture.

The PATROL 3.0 redesign set out to resolve the scalability problem that arose from the console-centric 2.0 architecture. The basic requirements were local control, local execution, and local storage. By agent-side placement of Knowledge Modules and the PATROL Script Language (PSL) commands, the first requirement of local control or “intelligence” was met. The agent also executed these PSL commands without requiring any external impetus, instead relying on its internal scheduling mechanisms, achieving local execution. The results of the PSL commands were stored on the local disk where the agent was running, achieving local storage. In this way, the PATROL 3.0 agent was capable of continued monitoring, and even corrective

actions, even in the absence of any external control mechanism. The resulting PATROL autonomous agent was a major innovation and was awarded a U.S. patent.

The solution was not only to go to a fully autonomous agent (see Figure 2-4), but also to make significant changes to the protocol that was used when a console is connected, to avoid the network and processing bottleneck at the console. Network traffic was reduced to almost nil by local storage of the PSL commands (avoiding the cost of downloading commands), and by changing the agents to only send parameter data values and object statuses on an as-needed basis. This second point, the change in the handling of data sent to a console, solved both the network traffic problem and also the console performance degradation problem, because the console had less processing work to do.

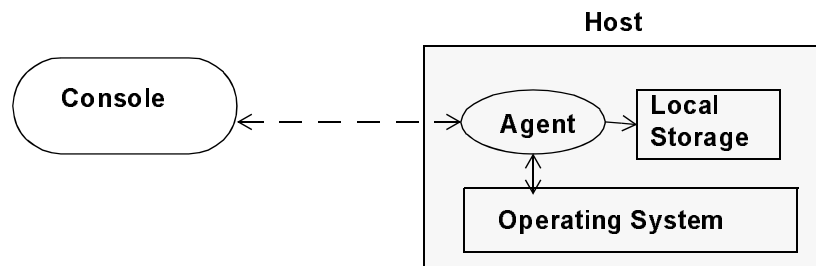


Figure 2-4 Autonomous agent architecture

This method of sending object statuses on an as-needed basis is a non-trivial algorithm. Clearly, when the agent is running autonomously without any console, no network traffic occurs. But how is a console to see the agent's status if it does not receive values from the agents while it is connected? The solution is that clearly it must send some statuses, but only when necessary. The cases where a status must be sent occur in a small minority of situations, leading to a massive reduction in traffic. These cases occur when:

- An alert occurs due to an exception, such as when a parameter exceeds a threshold value.
- An alert is canceled, such as when a problem parameter is corrected by a recovery action.
- An object is directly visible on the console screen, in which case the console must register interest in receiving "real-time" status updates for that object.

These cases form the minority because they involve problematic conditions or user activity. Too many alerts are unlikely to occur together, and the user cannot have so many windows open that it creates a network scaling problem. In fact, even in these rare cases, the scalability problem is not likely to arise, because the actual number of state changes is far, far less than the repeated parameter value sending that caused the problem in PATROL 2.0. The problem in 2.0 was compounded by the fact that all states were sent, even if the state had not changed, and all collected data values were sent. PATROL 3.0 never sends data values unless they are explicitly requested by the console, such as a user viewing a parameter graph.

2.7 Agentless Monitoring

The extreme opposite of an autonomous agent architecture is to avoid an agent completely. This is a full case of management from the “outside” without touching the monitored machine. No agent process is running permanently on the remote machine. This situation is the complete antithesis to the autonomous agent architecture.

Somewhere in the middle of the two architectural extremes is the case of introducing a command that is not an agent on the remote machine. This means adding an executable to the monitored machine, but not one that is an agent process. Instead, this command is run as needed on an ad hoc or scheduled basis. In any case, this architecture differs only marginally from the agentless architecture in that part of the access method uses a remotely stored command. It is still fundamentally an agentless architecture.

An agentless monitoring architecture represents an extreme version of a console-centric architecture. It has the same disadvantages, plus a few of its own. The agentless solution suffers from the same scalability problems because it imposes a pull or polling model rather than an asynchronous event-driven architecture. Network traffic also prevents scalability because the messages back to the central console become both a traffic and processing bottleneck. The console is also a central point of failure with no capability to perform automated local corrective actions. These are all the architectural disadvantages shared with a console-centric architecture. However, additional disadvantages are incurred by the absence of an agent.

The lack of agent on the remote machine forces all processing to occur on the console machine. This further exacerbates network traffic problems because of the need to send all raw data across the network. No capability exists to perform any local processing of machine data, as exists in a console-centric agent architecture. This lack also leads to a fat client architecture where the console is required to perform all the processing.

2.8 Application Management Agents—Not Web Agents

Unfortunately, two distinct classes of software have chosen to use the term “agent.” Agents exist in both the distributed systems management arena and the web technology arena. Both of these agents have quite different faculties, but are often confused. Even the advent of WBEM does not really narrow the gap, because WBEM uses systems management type agents rather than web-style agents.

An agent in terms of application management refers to a remote daemon capable of local actions affecting the machine on which it resides. An application management agent does not typically involve itself in the analysis of data from remote machines via network activity. This class of agent uses the network mainly for the communication with its console, for notification methods, and for integration with other clients from console-like components.

An agent in the area of web or Internet technology is far less localized than one used for application management. Such agents often comprise more exotic creatures such as spiders, crawlers, or robots. For example, these agents can be used to scour the web for information to build a personalized newspaper report each morning, or crawl across the web searching for

information on a particular topic, or simply to report on web pages that have been updated. These agents are very different from systems management agents in that they involve a more outward-focused approach via the network.

Cooperating or inter-communicating agents are another area where web agent technology issues arise. These issues occur, for example, when indexing data on the web, where one agent could index the information, and the other agents could gather it. Significant research in internet technology has arisen surrounding how to get agents to talk to one another for various purposes.

This area of agent cooperation is one where marginal overlap exists with the areas of systems management agents. These agents can benefit from sharing information from multiple agents on different machines across the network, such as for the monitoring of a distributed application such as SAP R/3. Although a very useful feature for distributed application management, this is a relatively uninteresting issue from a theoretical point of view. These agents are the same agent, and these agents do not have difficulty speaking the same language. The feat is of software engineering rather than research. More interesting is the sharing of management data between dissimilar agents. For example, one agent could gather some of its data from another type of agent. Indeed, this idea is achieved in a limited manner via access to the SNMP agent's data from other agents. However, the more general aspects of standards for sharing application management data are still an area of activity.

Roaming agents are another hot button in web agent research. These are agents that do not run locally and access the web over the network, but actually relocate their execution to the remote machine. This approach allows significantly more scalable distributed architectures for the solution of CPU-intensive problems such as text indexing. Unfortunately, the issue of roaming agents is not relevant to systems management agents, because the assumption in this case is that any interesting machines already have an agent upon them. They have no reason to roam the agent, because an agent would already be there. The power to perform sophisticated distributed algorithms is still there because many web agents are available to share the processing if needed. However, no complexity exists in the issue of "roaming" or "relocation," because these aspects are simply not needed. All systems management agents are static and assumed to be present on any managed server.

2.9 Proxy Architectures

A proxy architecture refers to an architecture in which one entity fronts for another. This is a kind of hybrid application management architecture. This is halfway between management from the inside and from the outside. In fact, a proxy architecture can be either.

In an "outside" proxy architecture, nothing is placed on the monitored machine. The monitoring and managing software resides on another machine completely. Access for measurement and administrative commands relies only upon existing network daemons. For example, such an application management architecture might well use telnet or the HTTP protocol, and rely on the remote machine's telnet daemon or HTTP server.

In an “inside” proxy architecture, you have a specific proxy subagent. Under this architecture, components exist on both the monitored machine and on the proxy machine. However, the component on the remote machine is typically not as fully featured as the local proxy component. If it were as fully featured, you would not need the proxy machine. Under this architecture, the remote component can perform commands or measure values, but will typically transmit these values back to the proxy agent. This is a trade-off between commands on the remote machine versus network traffic. It is purely a design decision as to how much of the processing of monitoring data is to occur on the remote subagent or on the local proxy machine.

The proxy architecture serves as an indicator that some middle ground exists between autonomous agent architectures and console-centric architectures. Sometimes the issues are not so clear-cut. In some cases, a proxy architecture is a practical alternative that has some characteristics of both approaches. It is like an autonomous agent, since the proxying agent is still autonomous. However, a console-centric aspect also exists since the proxy agent can lose connection with the remote machine, as in a console-centric architecture.

Some cases for the use of a proxy architecture are valid. One is clearly when no fully autonomous agent is available for a machine type. Developing a smaller subagent is clearly simpler than developing a fully functional autonomous agent. Another reason arises when the cost of processing on the monitored machine is so high that a cheaper approach is to send data across the wire and perform calculations elsewhere. This is a trade-off between server CPU time and network traffic, and can sometimes favor such a proxy implementation of an application management solution.

2.10 Polling versus Event-Driven Designs

A key concept that keeps arising in all these various architectures is that of a polling versus a non-polling architecture. Polling refers to one software component repeatedly asking or polling another component for a value. Generally, polling is an undesirable architectural component because it involves regular checking of a value even if it has not changed. This leads to wasted resources and scalability limitations.

The telephone is a good example of the efficiency of an event-driven architecture. Let us consider the case of an emergency call handling bureau (e.g., 911 calls in the U.S.). If the telephone had a polling architecture, they would have to call everyone repeatedly to check whether they had a problem. Fortunately, the telephone has a non-polling architecture, and when a problem arises, the initiating person’s telephone can send an “event” to tell the 911 call handler’s telephone to ring.

Other names for this type of architecture are “event-driven” or “interrupt-driven.” Another terminology used is that a polling architecture is a “pull” architecture, and an event-driven architecture has a “push” model.

The disadvantages of a polling architecture are not purely performance. Another problem is that the period of polling limits the level of immediacy of the information. If polling occurs only every 10 minutes, a problem may not be detected for 9 minutes and 59 seconds. Thus on

average, most data is at least 5 minutes old. An event-driven architecture does not suffer this lack of immediacy, because the component that detects its own problem will immediately notify the other component without waiting for itself to be polled.

The concept of polling versus event-driven architectures has already arisen in many of the example architectures. The SNMP protocol has a polling architecture where the console must use the SNMP “get” request to poll for values, but SNMP also has the SNMP “trap,” which is an event-driven architectural feature. A device can send out a trap as an event-driven action, rather than wait to be polled by an SNMP get request from an SNMP console.

An event-driven architecture can occur in many variants. One common idiom is a publish-subscribe architecture. In these architectures, the requestor “subscribes” to information in order for the sender to “publish” the information in an event-driven manner as it becomes available. This is a form of registration where one component tells it where to send events as they arise. An example of this is the nascent “push” technology for the Internet. A browser registers for information, and the server pushes that information to the browser when the information becomes available.

In theory, all application management architectures should be based upon an event-driven architecture because of the performance and immediacy advantages. In practice, the lack of management standards at the various layers of software make this theory difficult. The network level has only really the SNMP trap as an event-driven alternative. The operating system layer is not much better. Consider the problem of registering for event-driven notification when a file-system goes down. Which operating system provides such kernel-level hooks to register for events such as this? The typical solution is simply to poll, even when using kernel-level access methods.

Another common example is the need to watch an error log file and process any new messages that are added to the end of the log file. Which operating systems provide a kernel-level API to inform you when a file has been modified? Some do, but it is not a standardized feature. In practice, in some cases, polling the size of the file regularly to determine whether it has changed becomes necessary.

These difficulties show us that a non-polling architecture is not so easy to achieve. In practice, a polling architecture is used for many areas of application management because no standard way exists to register for events at the various layers.

2.11 Open and Extensible Architectures

The level of open access to an architecture is another key concept of application management. Openness refers to the ease with which external access is possible, that is, access by some software entity that is outside the other software entity. This issue of openness arises in two areas: the level of openness of the application, and in the application management tool.

The openness of an application is directly related to the effectiveness of its management. If an application is closed, no hooks exist by which to gather performance data, nor to send administrative commands to correct problems. By contrast, an application that is open is easy to

gather data from, and, provided a secure way, is also capable of being configured, tuned, or having other administrative commands to improve it. Without this feature of openness, the only application management that would be possible would be within the application itself.

Openness of an application typically raises impressions of a fully instrumented standardized method to application management. While this is the theoretical pinnacle of manageability for an application, a variety of less drastic measures can make significant improvements in the openness of an application to an application management tool. For example, the idea of openness can involve features as simple as having a readable error log file, or as complex as having a sophisticated application CLI or API. The success of the application management tools rides on this level of open access. This issue is visited in more detail in the discussion of intrusive versus non-intrusive application management methods in later chapters.

The other side to openness is that of the application management tool. This tool is also a software component that can be designed in an open or a closed way. A closed application management tool will simply choose what metrics to monitor and what administrative commands are available. These will not be changeable in a closed architecture.

An open application management tool offers various ways to extend its own functionality. The acid test of extensibility and openness is whether in-house applications can be supported. A closed architecture will have a difficult time adjusting to different variations in application deployment and usage, let alone dealing with an internally developed application.

Another area of openness for an application management tool is in its integration points for other third-party systems and network management software. A lack of open integration limits interoperability and diminishes the leveraging factor. Working with an existing network or systems management infrastructure makes application management easier to introduce.

An interesting feature of modern software is that GUIs are typically closed architectures. The designer of a GUI will typically attempt to take into account every need and provide a user-friendly way to meet that need. This creates a very powerful and detailed GUI. And it typically creates a closed GUI. If you have ever been blocked by some sequence in a GUI, you know what this sense of “closed” means. If the designer did not anticipate an eventuality, often you have no way around. For this reason and a few other reasons, the majority of system administrators dislike GUI-based tools for some of the more difficult tasks such as widespread deployment of software. Very rarely is a GUI open enough to handle some of the less predictable occurrences that the system administrator is used to handling. An open architecture with a CLI or an API is often far more powerful and less limiting than a GUI with preset sequences.

Availability Management

If one requirement applies to the application, it is that it must be available. Gone are the days when an alternative manual process was acceptable while the system was having problems. Modern business depends on the technology and the higher productivity and automation that is achieved. Paper pushing is not an adequate substitute. Dependence on the technology to handle the large volumes of information is unavoidable.

Nothing is more final than the system being down. Whether it is directly losing revenue or “merely” losing credibility and customer satisfaction, this is never a good thing. Availability management is the discipline of ensuring that this never happens. This chapter explores the place of application management as part of an overall availability management program.

Achieving high levels of availability is one of the most difficult tasks for an IT department. It is difficult enough to do in a centralized technology environment, and the situation is merely exacerbated by the modern distributed nature of many application architectures. Although this distribution is unavoidable to achieve the benefits of end-user data access and flexibility, it imposes its own new set of challenges to availability planning. One of the driving forces behind the rise of the application management discipline has been exactly this problem of distributed availability management.

Availability management is a far more general discipline than application management. True availability management encompasses a variety of technical and organizational issues within an IT department and its technology infrastructure. Because so many factors can cause an availability problem, availability management touches on the disciplines that control all these factors. Hence, availability management is a kind of encircling shroud around various technical disciplines such as:

- Network management
- Systems management
- Application management
- Problem management
- Security management
- Capacity management
- Change management
- Backup and recovery

Ensuring high availability of applications is not only a technical issue. Availability management also covers procedural issues including:

- Disaster contingency planning
- Application development and design practices
- Security practices and procedures
- Staff management issues

Although all of these areas appear to cover unrelated topics, they have a lot of common goals. Throughout all of these disciplines are two basic methods of improving availability:

1. Preventing outages
2. Recovering quickly from outages

Both of these major areas contribute to overall availability. Although prevention is clearly better than cure, both concerns need to be addressed.

3.1 Downtime Prevention

Prevention of unplanned outages is the best method of achieving high availability. This is where the main value of application management appears through proactive problem detection and correction. However, downtime prevention is far more general than just the application management concerns. You only have to look at the causes of unplanned outages to see the different issues that can help improve the overall availability of the entire system. Some possible causes of unplanned outages include:

- Hardware failure—CPU, disk, network devices, etc.
- Software failure—application, database, operating system, etc.
- Malicious acts—theft, vandalism, security intrusions, viruses, etc.
- Operator or administrator error
- Accidents and natural disasters—fire, flood, earthquake, etc.
- Staffing issues—strikes, key staff departures, etc.

Although many practices and procedures must be part of downtime prevention, many tools are available to aid the process. Application management is clearly one of the key parts of this issue so as to prevent downtime of the application by monitoring it through all the technology layers. It touches a surprisingly large number of the above topics.

Hardware failure is not as large an issue as it has been in the past. Huge advances in the technology of hardware components such as CPUs or disks have made failures in these components much less likely. Hardware is typically more reliable than software in modern times. One of the reasons for this reliability is the design of hardware for higher availability through advanced architectures. Clustering technologies and high availability mirrored systems are examples of architectures that make higher availability possible. Reducing costs of hardware means that these architectures are achievable as a reasonably accepted part of IT cost planning. In addition to these advanced hardware architectures, you should not forget the basics of hardware failure prevention, such as power supply control, air conditioning, and general environment concerns.

Software failure is clearly the main area addressed by application management. The number of failures from software are reduced through proactive monitoring and correction. This occurs at all the software layers, including the operating system, database, middleware, and applications. The preventive aspects of application management include a few key points:

- Continuous monitoring—the application management “robot” relentlessly watches the situation and does not miss problems as a human operator might.
- Deep monitoring—problems that are mostly hidden to the user, such as hidden error log messages, are brought to the surface because the application management tool watches even these deep levels of data.
- Automatic and early correction—where permitted to do so, the application management tool can try one of its pre-packaged methods to resolve detected problems, often early enough to prevent a more serious problem. These improve both prevention of downtime and recovery time.
- Early notification—informing the administrators about the possibility of a problem, particularly when considering predictive and early alarms, gives the administrators a chance to resolve the problem before it gets serious.

Malicious acts cover both organizational and technical issues. Security is one of the basic procedural issues that affects the risk of unplanned downtime. The issue of security is a huge topic that covers many areas, ranging from physical security of systems to the technological concerns such as encryption and single sign-on. Physical security issues include isolation of critical machines, locks, security badges, and many other practices. Technology security issues include encryption, single sign-on, firewalls, password policies, and password change enforcement. Also, a number of different tools can automatically police some of the technology aspects of security, including:

- virus detection tools
- security monitoring tools—intrusion detection, user policing, etc.
- security vulnerability analysis tools

Operator error would appear to be something about which little can be done. Issues such as training and internal procedures bubble to the top as solutions. However, application management does arise as an alternative technology that reduces operator error. It does so by providing packaged expertise that raises the capability level of the operators or administrators. This includes expert advice to help operators figure out what to do, and packaged administrative commands and tools that make easy doing the required tasks in an automated way that is not prone to typos or other human errors.

Natural disasters and accident prevention are clearly more an organizational issue than a technical one. Although a hardware failure might seem to be a “disaster,” the real meaning of this term here is more major problems such as acts of nature or acts of man. Some of the more extreme possible occurrences include fire, earthquake, and flood.

Preparedness is the best prevention here. Fire safety is clearly a huge corporate issue well beyond the realms of the IT department, but it applies here as well. What fire extinguishers are available should a small fire occur in one of the servers? How much loose paper is lying around and where is it placed? Are the machine air vents free from obstructions? Even if it does not cause a fire, the frying of a machine can clearly cause unplanned downtime.

Staffing issues are clearly organizational issues. The issue here is the management effectiveness and the hiring of adept staff. Any technology cannot do much to resolve issues such as strikes or labor market shortcomings. However, application management is about making the best use of existing people. By adding the packaged expertise to their existing skill sets, the employees can make more productive use of their abilities.

The above discussion has covered the issue of prevention of downtime. Unfortunately, not all unplanned downtime is avoidable, hence the need for this discussion. The other key part to improved availability is the ability to quickly recover from an occurrence of unplanned downtime. This minimizes the impact of a problem and keeps overall availability levels high.

3.2 Downtime Recovery

Once an unplanned outage occurs, you must recover the system to its working state as quickly as possible. Efficient recovery minimizes downtime and can sometimes even avoid the end-user or customers from perceiving that downtime has occurred. The recovery process involves a number of logical steps:

1. Diagnosing the cause of the problem
2. Determining the required corrective action
3. Performing the corrective action
4. Checking that the problem is resolved

These steps are clearly sequential. Delays in any of these areas will ultimately reduce the recovery effectiveness. Automated tools are the best way to speed up these processes, and application management has much to offer in all of these areas.

The ultimate level of automated recovery is the proactive corrective action. By automatically launching a corrective action upon the detection of a problem, an autonomous agent can immediately use its pre-programmed corrective actions. It goes through the four steps quickly by detecting the problem, determining its originating object either directly or via correlation analysis, finding the required corrective action from its library of actions, and launching the action. A check of its success is also trivially occurring, since the agent continues to monitor whether the problem condition exists.

The next level of rapid recovery is not fully automated, but involves cooperation between the user and the application management tool or other corrective tools. Automation can apply at various levels:

- Problem diagnostic tools
- Expert advice tools
- Problem corrective tools—administrative tools
- File backup and recovery
- Database backup and recovery

These tools can basically be separated into two major groups—database/application recovery tools and data recovery tools. The tools for diagnosis and correction aim to fix the problem with the database or application to get it running again. Backup and recovery tools aim to manage the data involved in that process to ensure that nothing goes amiss with the integrity of the critical business data.

Clearly, many recovery strategies require some forward planning. Leaving the “backup” out of “backup and recovery” is not going to make recovery that useful. Planning for rapid recovery is clearly a part of the procedures that needs to be examined. This ranges from the simpler aspects of local backups to the full issues of disaster contingency planning such as geographical remote backups and escrow storage.

An important part of recovery is the use of failover techniques to offer availability even when the full recovery sequence is somewhat involved. This is where redundant architectures come into play such that if one component goes out, the others can continue processing data. If one application server goes down, another should be available, and the end-user notices nothing amiss except perhaps some slowdown from the reduced capacity. Redundancy of data storage is another issue to ensure that the loss of one data store is not the end game. Some of the technologies that come into play for failover strategies are summarized as:

- High availability storage technology
- Clustering technology and redundant processing
- Mirroring

The other side to downtime recovery is the more relaxing issue of fast “recovery” during planned downtime. Strategies for minimizing the downtime at scheduled periods are what I will now discuss.

3.3 Planned Downtime

Change in the environment is unavoidable as the need to offer better service increases. Sometimes this means that the system must be brought down for planned upgrades or other activities. Minimizing these scheduled downtimes is another aspect of improving availability levels. For some systems, any downtime, planned or unplanned, is a revenue impact. Once again, you have two basic avenues to reducing planned downtime:

1. Preventing planned downtime by avoiding the need for it.
2. Rapidly recovering the system when it must go down.

These same basic ideas apply for planned downtime as well as for unplanned. The difference is that the situation is not an emergency, and careful planning can be brought to bear on avoiding and minimizing downtime. Some of the main reasons for scheduled downtime include:

- Backups
- Configuration changes
- Upgrades to hardware or software

The strategies for these different requirements have aspects of both avoidance and reducing time spent on the tasks. Backup of files or databases has both considerations. Avoiding outage due to backups involves changing to online backups rather than cold backups. Reducing the outage due to cold backups, if they are unavoidable, means using fast backup utilities to get the task done as quickly as possible. This is an area where the native database utilities are not often the best, and this issue has spawned many good third-party fast backup tools. These tools are also about fast recovery, but they apply to recovering quickly from unplanned outages, whereas the speed of the actual backup sequence is the main factor for reducing planned outages.

Various maintenance tasks are the other main cause of planned downtime. The above list shows the main issues such as re-configuration, hardware maintenance, and software maintenance. Sometimes this use of planned outage for maintenance is part of a strategy of avoiding unplanned outage. To have a little planned downtime is far better than to suffer a catastrophic and unpredictable outage. Various maintenance tasks can have the effect of helping with unplanned outages. One example is replacing a component that is having intermittent trouble or degraded performance. Another is the addition of extra software that aids in the detection and resolution of problems, and application management software is clearly an example of this.

Minimizing the outage due to these scheduled maintenance or upgrade issues is an important consideration. The issues are clearly very different for hardware and software. Replacing

hardware components can be moved to an on-line activity through various redundancy technologies. Software maintenance or re-configuration is often easier to achieve in an online environment and avoid any outage. For example, although operating system kernel changes for tuning are often an offline affair, newer tools are available for dynamic online kernel configuration for the operating system. Database and application re-configuration or tuning is often an online practice. These strategies are all for converting a planned downtime period to an online period, to offer continuous availability of the systems and applications.

3.4 Application Design for Availability

No riskier time for availability occurs than the upgrade to a new release of an application. Although this statement is true for both commercial and in-house applications, more concerns exist for in-house development. The basic issue is quality. Of course, this is an issue about which tomes have been written, but in brief, here are some of the key issues:

- Testing, testing, testing!
- Testing tools—memory checkers, regression test harnesses, etc.
- Source code version control and tracking
- Coding issues—unit tests, error/exception handling, defensive coding, etc.

None of the above is new, but it does not hurt to say it again. Mission-critical applications should be treated this way from their design through to their delivery.

Beyond basic quality, you have other issues to consider for improved availability. These are based on the presumption that software can predict and resolve internal problems only to a certain extent. The program must be robust and reliable in its operation environment despite the surprises that might be thrown at it. The aim here is to keep the application running at all costs, and also to provide early warning about problems. This discussion is bridging the gap between basic code reliability and providing manageability via instrumentation. As examples of the former, some of the basic code reliability considerations include:

- Exception handling—file access errors, network connection losses, etc.
- Resource failures—disk full conditions, out of memory, etc.
- Plain old bugs—these never happen, of course.
- Internal consistency failures—assertion failures, invariants varying, etc.

How does the application handle all these conditions? Its fragility depends on whether it is brought down by the slightest breeze or whether it survives through outstanding exception-handling code and reliability layers. Even plain old bugs can be minimized through defensive coding, argument checking, and mutual suspicion. Modular design can keep a bug in one area from adversely affecting another user, transaction, or functional component. This is all about the level to which the software makes assumptions about results from other modules.

The next level beyond this reliability is design for overall availability. One aspect is designing for redundancy so that the failure of one component can be transparently replaced by another. This is no small feat to achieve.

Another aspect is the instrumentation of the application at key points that indicate availability risks to permit early warning. The entire issue of instrumentation is covered in more detail in various other chapters such as Chapter 7, but the general idea is to provide early warning. For this reason, even if the application can continue under some of the above danger conditions, it should provide information to the application management tool to permit notification and proactive resolution of the problem before it gets to the point where the application simply cannot function any further.

The final point is that sometimes the application just has to come down. Maybe a coding bug is causing a SIGSEGV on UNIX or some internal problems allow nowhere to store the data. In either case, to offer a graceful failure is important. Even SIGSEGV can be intercepted to run a limited shutdown routine. This abnormal shutdown sequence should attempt to assist in both the diagnosis and the recovery of the system by providing diagnostic data and protecting data integrity. Good application design on this last point will facilitate rapid recovery of the application once it has been downed.

3.5 Investing in Availability

Availability is clearly the number one concern of the IT department. This makes important ensuring that adequate resources are assigned to this task, rather than to more futuristic plans. This resource assignment involves both staff and technology assignments. Technology without adequate staff is not going to solve an availability problem. Nor can good staff be fully effective without some of the more advanced technologies and tools. The IT manager must balance both of these major factors.

Staffing issues for availability are not just about assigning enough people, although that is clearly the most basic requirement. They are also about ensuring that they are applying their efforts most productively and to the greatest gain. To know where the greatest gain is to be found, you need to know the greatest pain. Each environment has particular problems that arise repetitively, and others that are surprises. Be sure to measure these problems to determine what can be done about them and which to attack first. Incident postmortem meetings are a good step to diagnosing what went wrong and for brainstorming solutions to any issues. Usually, both strategic and tactical considerations are necessary to the plans for improving the situation. Solutions should examine both sides of the coin: prevention and minimization of impact. The issue of impact has considerations not just in the duration of the outage, but also in whether critical data is lost, in whether transactions must be reentered, or in other impacts on the end user.

Solutions to outage concerns often arise through better technology. A great many technologies have been examined throughout this chapter. Hardware technology issues are a key concern. Software technologies are equally important, and clearly, application management is one of the major issues here. Data management via backup and recovery and rapid restart tools is

another key technology concern for availability. However, not all the tools in disciplines such as application and data management are aimed towards availability. Other issues such as performance management and data integrity are important, but not directly related to availability. Be sure to examine the components of the various tools to determine what improves availability, what applies to other concerns, and how to balance all of these issues.

Problem and Event Management

One of the major concerns for application management is the detection and correction of problems. This issue is typically interrelated with the event concept in modern application management solutions. Problem management is a more general discipline than event management. Problem management deals with the entire management issue of tracking problems through their life cycles. Event management represents a subset of this cycle that applies to application management. Full problem management involves both event management and help desk problem tracking disciplines.

This event management side of application management is often termed “management by exception” because it focuses on the exceptions that caused the alerts. The assumption is that no news is good news. An application is working perfectly until monitoring detects otherwise. As exceptions are discovered, they are managed through an event life cycle from their creation to resolution.

The basic aim of problem management in application management is quite simple: detect problems and correct them. Detecting problems involves the two opposing sides of detecting as many problems as possible, while not reporting false alarms about non-problems. Attempting correction is the next step following detection. Methods of correcting problems range from automatic corrective actions through various manual aids, including interactive diagnostic tools, post-mortem information storage, or static expert advice textual information.

Hence, application management plays a different role in problem management to a help desk or consolidated service desk. The two are complementary and usually both are present in a large organizational structure. Application management tools serve to create events based on monitoring. The help desk serves to collect, assign, and track each event. However, the help desk software does not attempt to create events or to actively play a role in correcting them.

4.1 Event Generation and Problem Detection

The primary goal of an application management tool is to detect problems. Unless a tool can detect problems with reasonable accuracy, it has little value. Typically, problem detection occurs through the generation of an event. Several distinct categories of problems can be detected, including availability, performance, and load. These are generally detected in two basic ways:

- Boolean status via visible defects.
- Measured numeric values.

The methods of detecting visible defects are varied but basically depend on analysis of application components. Examples include absent application processes or files, zombied processes, or some other clearly problematic situation. The status is Boolean: either a problem exists or it does not. No numeric scale of values exists. Availability is a critical issue that is typically measured this way. Error messages detected in application log files are another common example. The presence or absence of these messages is usually the issue, rather than the number of such messages. In these cases of absent daemon processes or error message text, you have no level to configure, only a Boolean setting of whether an alarm should be issued for the situation.

Measured metrics are numerical values to which numerical algorithms are applied, to determine whether to issue an alert event. Numerous measurement examples include response time, cache hit ratios, error log file size, CPU idle percentage, network throughput, and so on. These are all numeric values with a range of values. The number might be an absolute measure, a percentage, a count, or a ratio. In contrast with Boolean alerts, whether any given number should cause an alert is not immediately clear. At what numeric level of CPU utilization should an alert be raised? Various solutions to this question are available, including:

- Simple numeric ranges
- Rate of change
- Absolute versus relative values
- Statistical historical analysis

Simple range thresholds are common. For example, a threshold on CPU utilization might be set to alert when CPU is greater than 90%. Clearly, these simple thresholds should support a variety of basic operations. Some values should alert for high values (e.g., CPU utilization), whereas others should alert for low values (e.g., cache hit rate). All of the common relational operators are desirable, so as to express the following different criteria:

- Greater than a value (value > threshold)
- Less than a value (value < threshold)
- Ranges: greater than a value and less than another value (threshold > value > threshold)

- Equal to a value (value = threshold)
- Not equal to a value (value != threshold)

Clearly, to support more than one such range is desirable. One reason is to offer different severity levels on different ranges (e.g., warning for 80-90%, and alarm for greater than 90%). Another reason is to offer different actions that occur at different values. All of the ranges must be configurable to support different threshold settings under different circumstances.

Rate of change is another important alert metric. Consider the following scenario: one filesystem has been at 98% for a month, and another jumped from 40% to 80% in a single day. With a simple threshold at 90%, you would get an alert for the stable filesystem, but no alert for the growing one. Clearly, some measurements exist for which growth is more important than absolute value. To key alerts off the daily growth rate of a filesystem may be more useful than its absolute value.

Related to rate of change alerts are crossing alerts. In some cases, you may want to alert on a value if it occurs “on the way up” rather than going down. This fairly simple test is based on the previous value of the metric.

Another numeric issue is to determine what numeric form to use: relative or absolute. Consider the filesystem example again. Should you alert on the basis of bytes free or percentage free? Usually the percentage is more valuable, but if it is an immense filesystem, then perhaps 1% of disk space is still plenty left over. Here, rate of change is probably better than either measure. But it does show that you should consider the meaning of a number when deciding what to alert on. Absolute counts are often not that meaningful unless you have something to compare them against, often to create a ratio or percentage measurement. However, absolute counts can also be useful for some values.

Statistical analysis is an advanced method of determining alerts. This is even more general than rate of change, which examines only the most recent previous value. Statistical analysis of historical data attempts to automatically detect when the situation is abnormal and deserves an alert. The type of analysis can be quite simple, just as when the value is three standard deviations away from the historical mean. It can also be quite complex, involving trending or line interpolation. The curve might also have a time dependency due to reduced loads on the week-end. The aim is to compare a single point against a group of past values to determine whether it fits the pattern or is abnormal enough to report. Sample size for the points is important in order to establish a significant pattern. You cannot get a reasonable view based on only a few points. The size of the sample might be a reasonable number of points, or perhaps it is the requirement to have some time period covered, such as the entire past week. So in an initial period, the event manager is collecting data and not reporting alerts. This is dangerous in that an abnormal situation could occur during the collection period and then be assumed normal. Therefore, any historical analysis method needs to have a manual method of inspecting and overriding the results, or perhaps some backup method where an alert is guaranteed under some simpler circumstances. For example, a filesystem capacity measure could have a statistical analysis, but also a fallback

to alert under simple circumstances, such as when it grows more than 10% into the 90% range in a short time period.

Predictive alerting is the next level of historical analysis. This involves guaranteeing an alert at some period prior to its occurrence. For example, you want to know at least 24 hours in advance that a disk is going to fill up tomorrow. This is a generalized form of growth rate and statistical analysis. This applies to values that are not too volatile and somewhat predictable, such as disk storage. Sometimes unpredictable values can be averaged to get a reasonable trend. For example, you would not try to make predictions based on raw CPU utilization, which goes up and down, but you might want a prediction based on the trends in the hourly average CPU utilization. This might also require time-based analysis to make different assessments of business and after-hours CPU utilization trends.

Smoothing peaks is an important consideration for alerting. You would not want an alert on every peak of CPU utilization. Peaks can be smoothed using simple methods such as not alerting for n consecutive points above the threshold, or if the CPU stays above the threshold for a certain period of time, say 30 minutes. Averaging the values over a period is another form of smoothing.

Time dependency is an issue for all the different types of alerting algorithms. The simple thresholds could be 90% CPU utilization during evenings and weekends, but 50% CPU during business hours. Usually the fact that the machine is busy after hours is not important, because either it is just doing backups, or the users are not real customers. Statistical analysis and predictive algorithms are also enhanced by time-based considerations.

Sometimes you want to have time as a measured type itself. It could record the time at which the last backup occurred, or the duration of a backup. You could alert if the last backup was more than 24 hours ago, or if the backup duration is more than a few hours. Both problems could indicate failure of the backup process.

An interesting wrinkle in the event generation issue is that inverse situations can occur. For example, sometimes you want to be alerted on high CPU utilization, and sometimes on low CPU utilization. For example, high CPU might be abnormal during work hours but expected during off hours when the backups are running. Therefore, you want to be alerted for low CPU utilization during the off hours, because it indicates a problem with the backups. Situations such as this make the alerts appropriate for high CPU utilization in one time period and low CPU utilization for another. The situation need not be time-based but could depend on some other environmental issue such as the presence or absence of a particular job, user, or other condition.

The practical use of an event generation system can be simple or quite complex. Not all of these various alerting algorithms are necessary. One thing that is desirable is the capability to choose multiple different algorithms, such that you can have a simple 95% limit on a filesystem but also a growth rate alert. Multiple conditions help catch all the potential cases. You could use “and” or “or” conditionals on multiple conditions. Each different condition needs manually configurable parameters such as the threshold values, the time delays, sample sizes, and so on for all the different decision algorithms.

4.2 False Alarms

Not all alarms are desirable. An event management system aims to provoke alerts only on real problems. Unfortunately, some common types of false alerts need to be considered, and sometimes configured to remove. Each user has a different environment, and what is an alert in one environment may not be in another. Issues differ even within the environment.

The simplest form of false alert is related to environmental differences. Some machines should get an alert at over 50% CPU utilization, whereas others run above 90% constantly. This type of false alert can be resolved through proper configuration, or via a rate of change alert or the statistical models that attempt to learn what is acceptable via a learning period.

Scheduled downtime is another common cause of false alerts. The event manager needs to learn or be told that events should not be generated when the database is brought down for its scheduled maintenance period. Similarly, scheduled backups should cause events related to disk I/O to be suppressed for that regular time period.

Intermittent alarms are another cause. CPU utilization has peaks and troughs and does not necessarily deserve an alarm for each peak. A flashing red icon or repeating pager message is not the desired situation because it will soon be ignored. Instead, the packaged event management for systems typically has some form of peak smoothing involving a count or time-based suppression of individual peaks.

The most complex form of false alert is the alert cascade, where one real problem causes an alert but also causes a cascade of other events that hide the real problem. This is not actually a false alert, so much as too many real events. The attempt to solve this problem moves us into the realm of event correlation.

4.3 Event Correlation

Event correlation means different things to almost everyone. The problem is that several distinct issues all have the same aim of the reduction of redundant events, and all are collected under the umbrella term of “correlation.” Areas that appear to get lumped under this general term include:

- Event filtering
- Event aggregation
- Root cause analysis
- Rule-based correlation

Some of these areas are similar, but others are completely distinct. For example, event filtering and event aggregation are probably subsets of rule-based correlation, but root cause analysis is an entirely different kettle of fish. Each of these areas deserves analysis in turn.

4.3.1 Event Filtering

Event filtering is the simplest form of correlation. Event filtering involves removing unwanted events. Some of the possible aspects of event filtering include:

- event de-duplication
- scheduled blackouts
- event views

Event de-duplication involves removing repeated events from the same object. For example, do not send two events that a filesystem has filled up. You may want to know how many times this event has occurred and how long it has been open, but you want to see only one instance of the event. Perhaps after a certain period of time, the event should be escalated. Another issue is that if the initial value that caused the alarm changes marginally, to get a new event may not be interesting; but if the value changes significantly, it may indicate a whole new severity event. A particular issue is that if the value returns to normal levels, the event should be canceled. However, if the value is volatile, it will cause a stream of events and cancellations, which an event de-duplication algorithm should take into account. An example would be an intermittently flashing alarm on CPU utilization if the peaks are not smoothed.

Scheduled blackouts are another form of event filtering. You do not want alerts about a development system on the weekends. Alerts should not be generated for the cache hit ratio during a database backup. These situations are generally time-based according to established weekly or monthly routines. Event filtering with these features can remove these unwanted alerts.

Event views are related to event filtering. Events are not really filtered from the system, but rather are simply not shown to the user. Per-user event views enable DBAs to see only database alerts, whereas SAs see only system alerts. The views need to be based on job roles. The simplest types of views are for different host origins or different applications. DBAs want to see Oracle events, whereas SAs want to see CPU and disk alerts. The granularity of the viewing also needs to be low enough to get down to the individual event or metric. For example, the help desk might need to see Oracle or filesystem availability alerts, but only the capacity planning group needs to see long-term predictive alerts related to Oracle loads or filesystem capacities.

4.3.2 Event Aggregation and Rule-Based Correlation

Event aggregation is the process of merging multiple events into a single compound event. The aim is to avoid floods of events in favor of a sane number of events. This involves combining related events. For example, you would combine a high filesystem I/O processing alert with a low database cache hit ratio alert, or you would combine a low CPU idle alert with a high run-queue length alert. Both these conditions would become a single composite event in each case.

Rule-based correlation aims to take the numerous raw events and apply rules that reduce them to a manageable number. Event filtering is the simplest class of rules, with features such as de-duplication, time-based blackout, and unimportant event filtering. Rules are generally Boolean or arithmetic conditions. More advanced rules are desirable. For example, time-based event aggregation involves conditions such as two events occurring within a particular duration.

The scope of event aggregation is also important. Some of the simpler cases are node-level correlation. For example, tying together a filesystem alarm with an Oracle cache hit ratio alarm is reasonably simple to specify. More difficult is inter-node or enterprise-level event correlation. This involves making sense out of alarms from throughout the entire enterprise. Some events are related and some are disjoint.

Prioritizing events is another aspect that rules can handle. Event prioritization is specialized in that it characterizes the views according to the severity of the events. Some event types are more severe than others. Some event origins are more important than others, such as the differing importances of the same event on a production server versus a test server. You do not want a page on Sunday morning at 3 a.m. just because your desktop's screen saver is taking too much CPU on your laptop. Event prioritization aims to filter out the unimportant events. However, here we are getting beyond the scope of real event correlation into the issues of help desks and how they should handle events.

4.3.3 Root Cause Analysis

The main distinction between root cause analysis and rule-based correlation is that root cause analysis introduces a whole new class of events. The original source events are not the events that are shown to the end-user. These new events are the underlying causes, whereas the source events are the symptoms. The primary aim of root cause analysis is to map the cause events to the symptoms.

This is more like expert advice and expert problem diagnosis than it is event correlation. Root cause analysis aims to perform a sophisticated analysis of the raw events and produce the resulting diagnosis. Some limited forms of root cause analysis are possible via rule-based systems. However, rules are generally not powerful enough to solve problems involving dependencies between components. Generalized root cause analysis involves both built-in information and user-specific inter-dependency information about each component. This information and root cause analysis algorithms are used to produce a diagnosis event.

Topological information is a special type of component dependency. The diagnosis of network-related problems is particularly strongly related to the need to specify topology relationships. These complex graphical structures can clearly not be easily specified via simple rules. A sophisticated root cause analysis engine can determine from a flood of alarms which ones are "cause" events and which are "effect" events. The failure of a router component can be diagnosed as the cause of alarms related to all the applications depending on that network path.

4.4 Problem Diagnosis and Correction

Correction of a detected problem is the next level beyond problem detection. An application management tool should provide pre-packaged tools to automatically correct problems or to help the user manually diagnose and correct problems.

Architecture is a major aspect of problem correction. An autonomous agent is necessary to achieve fully automatic corrective actions without a console dependency. This permits a lights-off monitoring mode that also corrects problems.

The next level of problem correction, assuming that automated correction is not possible or has failed, is to help the user diagnose and correct the problem. Problem diagnosis involves two main issues:

- Having enough data to make a diagnosis
- Having the tools to analyze the data

Diagnosis requires information about the problem symptoms. This involves both current data and historical data. Current data involves features such as ad hoc reports that can be run to analyze an object or set of objects. For example, reports to show which processes are CPU hogs will help diagnose a processor utilization issue. These and other tools can help determine what the current situation is and diagnose the problem. Chances are that these tools can also correct the problem, or at least suggest a remedial action to take.

Historical data is also critical for many problems to provide postmortem data. The historical storage of aspects such as SQL scripts and the current list of users provides information to diagnose problems such as high CPU usage. Even if the CPU problem has disappeared, the data remains in historical storage, and the user application with the poorly tuned SQL script can be examined. Using this data, postmortem analysis and diagnosis is possible for transient or intermittent problems.

Extensibility is important to problem correction because, usually, existing work practices help resolve problems. Procedures, scripts, and tools will naturally arise to solve common environmental problems even without a formal application management project. You want to be able to add these to both the expert advice text (for known procedures) and to the set of diagnostic or corrective tools (for existing tools).

4.5 Problem Management

The issue of problem management is an IT discipline that is important to application management. Detection and correction of problems is only half the picture. You need to have some “management” of this process to achieve measurement and problem tracking.

This issue is moving into the realm of the help desk or consolidated service desk. These disciplines are complementary to application management tools. Help desks centralize the handling of problems but may hand off to the application management tool for diagnosis and correction. Help desks are strong in issues such as ensuring that all data about a problem is recorded (e.g., time, location, etc.) and that the problem is assigned to someone for resolution.

Problem tracking ensures a consistent approach to all events and problems. This involves tracking a problem to ensure that it is resolved. Reporting of the problem counts, statistics, and

resolution rates is another important aspect of problem management. Metrics such as MTTR and MTBF are important to track.

An advanced consolidated service desk might also offer features such as asset management and change coordination. These allow the issue of problem management to be treated as a holistic process. Change management is important to problem management because of the simple fact that change causes new problems. If some new problem arises, the first question to ask is what has changed. This issue accounts for the value of these change management features in a service desk. However, this topic is taking us well beyond the realms of application management into a more general discipline about which many volumes have already been written.

C H A P T E R 5

Performance Management

Ensuring adequate performance of applications is a major part of application management. Performance issues are just as serious as availability issues. In fact, a severe performance problem effectively becomes an availability problem as end users refuse to use the application. Poor performance also leads to reduced productivity of end users and lack of flexibility to address business concerns. This makes the performance of applications critical to the health of the business.

Managing performance is an important discipline for advanced IT personnel. This involves skills that are both business- and technology-oriented. The human side of performance involves issues such as technology architecture and user interface ergonomics. The technology side involves the use of techniques such as network management, systems management, and application management. Although this book focuses on technology issues, I briefly look at some of the broader issues.

5.1 The Human Side of Performance

Performance management is the full discipline of ensuring the performance of the overall business process. This extends beyond application management into the human side of business processes. An efficient application combined with an inefficient bureaucracy is still just an inefficient bureaucracy. This explains the interest in Business Process Re-engineering (BPR) throughout modern corporations. This is a corporate management initiative aimed at replacing inefficient human procedures with more streamlined procedures.

Surprisingly, this sometimes involves replacing fast computers with slower ones. That the overall business process is more important than the CPU capacity is one of the driving forces behind client/server implementations. The human factors such as data accessibility, flexibility,

and local control far outweigh the advantages of a more powerful central CPU. Placing the processor and the data closer to the people helps the people become more effective. This model is made possible technically by the advent of better processing power. You can place significant computing capability on the desktop or in a departmental server, instead of having to centralize all the power in the glass house. The use of a client/server architecture can represent the first step toward improvement of the overall performance of the human organization.

The executive management of a corporation must address these concerns. They are not really the domain of the CIO or IT architect. However, in order to fully exploit technology as a business-level competitive advantage, do consider these overall human factors. Further discussion takes me too far from my central theme, and I now turn to technology issues.

5.2 The Technology Side of Performance

One word runs as a common thread through all performance issues: queues. Everything in computer performance is about queues. You have I/O queues, CPU job queues, transaction queues, message queues, and so on. Growth of any of these queues from the resource contention resulting from multiple requests is the underlying cause of any performance problem. Really, you have only one way to improve performance—reduce those queues. This is the basic end goal of all the following basic performance improvement techniques:

- Capacity upgrades for components
- Parallel architectures
- Tuning and configuration

So what do all these have to do with queues? Upgrades of hardware or software components reduce queues by increasing the throughput of each queue (e.g., a faster CPU empties a queue more quickly). Parallel architectures basically aim to provide more queues so that each one is shorter. And the various tuning techniques attempt to either load-balance across multiple queues or improve each individual queue.

To some extent, this discussion explains why capacity planning and predictive modeling center around network queueing theory. This type of modeling examines the performance improvements to expect from either upgrading individual components (e.g., the CPU) or moving to more parallel architectures (e.g., parallel I/O via differing RAID levels). The only downside to these techniques is the cost of upgrading the hardware. The alternative is to use application management techniques to attempt to tune the existing system hardware and software.

5.3 Tuning Techniques

Unfortunately, the cost of revising the entire application architecture is immense. The undertaking is also risky, as with any technology change. The alternative to a major change is to make better use of what is available through tuning parts of the system. This is cost-effective and can often save on the need for more hardware capacity or new components.

Tuning the overall system is an interesting and rewarding task. Performance tuning of the existing system involves two basic disciplines:

- Network tuning
- Application tuning

Tuning the network is a well-known task that is mostly beyond the scope here. Techniques include separating networks into separate segments or subnets, increasing network component capacity, or moving to better switching components. However, monitoring the performance of the network layer is still possible via application management technology.

Tuning the various layers of the application is the next step. This takes tuning from the network hardware through the operating system and database to the overall application. The various application layers are categorized as:

- Network layer
- Operating system layer
- Database layer
- Middleware layer
- Application layer

The performance of these layers can be monitored via software tools. These are where application management comes into the picture, with help in tuning databases and applications that are already in place. Application management involves not only tools to improve performance, but also the very important area of measurement.

5.4 Performance Measurement and Monitoring

Software developers are poor at guessing where their application is spending its computation time, perhaps because the most efficiently coded module can be the largest performance hog simply by virtue of being executed too frequently. Network diagnostic engineers are probably much better at performance diagnosis than developers, but the same overall comment holds true. To measure the situation is better than to attempt to draw a conclusion without enough information. This rule explains the high value of network diagnostic tools and the people who know how to use them.

Similar comments regarding measurement hold true throughout all layers of the application. Measurement is the only true way to determine where the cause of a performance bottleneck can be found. Is it the network, filesystem, database, or the application itself? Application management tools provide performance measurement at a detailed level to determine the real cause.

Monitoring is a further step up from measurement. Monitoring is automated continuous measurement combined with logic to raise alerts at the first sign of a potential performance

problem. Performance monitoring is the best way to achieve proactive control of performance instead of waiting for the problem to strike before measuring.

5.4.1 Network Performance Monitoring and Tuning

Performance management of the network layer involves several combined activities. The first level is the overall availability management via the existing network management tools. These tools serve to detect and diagnose downtimes due to failures of network components. A variety of enterprise network management consoles are based on SNMP and other technologies. This area is not really the domain of application management tools.

The next level is measuring overall performance metrics of the network traffic. Overall statistics are useful for each machine, such as its data transfer rates, packets sent in and out, and so on. These statistics can be gathered either via a network sniffer technology or via some of the operating system API calls and commands that report on the machine's kernel statistics related to network operations.

The next level of network performance monitoring is the granular analysis of network data packets. You need to determine how much network traffic is due to Oracle versus Lotus Notes. The individual packets can be mapped via their socket port number and other details to the particular application that causes them. This approach brings the network performance attributes back into the realm of the application management tools.

5.4.2 Operating System Performance Monitoring and Tuning

You have two main performance issues at the operating system level. The first is the monitoring and tuning of the performance of the operating system. The second is the use of operating system low-level information to gather performance statistics for higher-level software components, but this is better seen in the later discussions of database and application performance monitoring.

Monitoring the performance of the operating system involves either kernel-level access to internal statistics or the use of commands written using these kernel access points. UNIX operating systems have many command-line tools such as `sar`, `df`, and `ps`. Windows NT has a lot of API calls, on top of which are graphical system monitoring tools. Many performance metrics should be monitored, including CPU utilization, memory, disks, and printers.

Tuning the operating system typically involves tweaking kernel settings. Often, you need to recycle the operating system for these settings to take effect. However, some level of tuning is often possible without recycling the machine.

The brute-force approach to speeding up a server is to upgrade its hardware capacity. This might mean a larger processor or more memory space. This certainly does offer the best solution to improved performance but naturally comes at a cost. This cost can often be avoided or delayed by proper tuning of the various software layers of the applications that run on the server.

5.4.3 Disk I/O Performance Monitoring and Tuning

A common low-level bottleneck is file I/O for storing data to a disk. Identifying the hot disks can help tune the overall application. The resolution to the performance problem might be a high-level change, such as moving a tablespace to a different disk, but it can be measured at a low level. Monitoring is achieved by detecting the low-level request counts and the length of the associated queues.

I/O request measurements are not the only relevant performance metric. Disk capacity may seem more relevant to availability and downtime than performance, but monitoring of disk space availability is often relevant to database or application performance. Low swap space levels can have adverse performance impact on the operating system, database, or application.

Advanced technology will greatly improve the situation regarding I/O monitoring into the future. For example, the i960 chip from Intel aims to offer more intelligent I/O processing. The resulting I₂O initiative is an interesting example of a hardware vendor taking steps to improve manageability of the overall system. Other initiatives in the areas of intelligent storage and storage area networks will lead to even more manageable systems for the storage of data.

5.4.4 Database Performance Monitoring and Tuning

Database tuning is one of the main areas of expertise for the DBA. No coherent methodology transcends the various database vendor products, but instead a collected wisdom exists about what works and what does not for the particular database. For example, many books on Oracle discuss the various parameters in the “init.ora” startup configuration file. Unfortunately, these parameters can be effective only within limits because of the continuously changing loads on databases. For example, the growth of ad hoc database analysis tools such as reporting tools or PowerBuilder applications means that end users can easily overload a database. This dynamic environment highlights the need for continuous performance monitoring rather than a “set and forget” style of database tuning.

Monitoring requires the ability to gather performance data about databases. Modern commercial databases are instrumented with their own internal performance metrics that are externalized through APIs or client executables. However, few or none of them have their own built-in continuous monitoring or alerting on these performance metrics. This 24x7 monitoring is one of the main advantages of application management tools.

Monitoring of database performance can occur at multiple levels. Simple metrics are available, such as cache or page hit ratios, stored procedure hits, and other general measures of performance. Also, deeper analysis is required, such as the extraction of the currently running SQL script when performance degrades. For example, this helps the DBA know what SQL script a poorly written PowerBuilder application is running.

The above comments regarding monitoring are more useful to the DBA than to the application developer. The next step beyond monitoring is tuning the application by implementing changes to its configuration or design. Common general design comments always apply, such as de-normalization to improve performance through increased availability of redundant derivative

data. In addition, some tools help analyze in more detail the performance of the SQL script with respect to its indexes, access path, and general performance pitfalls.

5.4.5 Application Performance Monitoring and Tuning

The main promise of application management is application-level monitoring, and this includes monitoring performance characteristics. This is achieved in two basic ways: the monitoring of the underlying components, and the direct monitoring of the application through its own higher-level access points. Both of these methods provide the capability for performance monitoring of an application.

Component-level monitoring of lower layers of the application stack can detect particular performance problems. Components such as CPUs, networks, disks, and databases all have their own performance indicators. If one of these components is performing poorly, the application will almost certainly perform poorly. For an application management tool to bring all this measured data up to a combined application-level view is a simple matter.

The other method is the direct application-level analysis. This occurs via whatever interface points the application has provided. Commercial applications such as SAP R/3, BAAN, and Oracle Applications provide sophisticated access points through which a lot of performance data can be gathered. The monitoring solutions for these applications are responsible for taking this data and automatically and continuously checking it for problematic symptoms.

5.5 Performance Problem Diagnosis and Correction

Monitoring is the method by which performance problems are detected early. Diagnosis is the method through which the cause is identified and the corrective action is selected. The detection of the performance problem early is valuable in itself, but a full application management solution needs to offer assistance in the diagnosis and correction phases. Problem diagnosis and correction are where performance monitoring meets performance tuning. Automatic, deep diagnosis and localized correction of performance problems is the best type of performance tuning available.

Sometimes the very act of monitoring itself brings about an obvious diagnosis. Through application management's monitoring of every application layer, a performance problem at one of these low levels is easily identified as the cause. Application management sees through the top levels of the application stack down to a hot disk, a low database cache hit rate, or a slow network link. The alert event associated with these low-level objects is enough to diagnose the problem.

In other cases, more detailed diagnostic analysis is required. Commands to find the cause of a problem can be executed if the problem still occurs. For example, high CPU utilization can be diagnosed through a command that reports which processes and users are using the CPU at the current time. Of course, this is also something that could be automatically executed by the application management tool as a diagnostic or corrective action.

In cases where the performance problem is transient or intermittent, you need to record historical data. How can you diagnose the cause of a short CPU spike if it disappears quickly? The answer is that the application management tool must automatically detect the problem and run diagnostic commands to gather more data. PATROL does so via recording the historical values and then annotating them with extra data when they go out of range. For example, the PATROL Knowledge Module for Unix detects peaks of CPU usage and annotates the points with extra data about the processes and users that were running at that time. This historical data and annotations are persistently stored for later post-mortem analysis of a performance problem.

5.6 Interactive Response Time

The dawn of client/server has brought a new performance requirement: real-time response time. Users expect to use software in an interactive manner and receive the results of their actions in a timely fashion. The concept of response time from the user perspective is a key issue in systems and application management.

This is entirely different from the mainframe picture of some years ago. Batch processing was the order of the day. Jobs were prepared and batched through the number cruncher. No such thing as time slicing or response time existed. The job took as long as it needed and was measured according to that form of efficiency. The measurement was throughput of jobs rather than the individual response time for each job.

Time slicing, real-time operating systems, and client/server have all contributed to the rise of interactive computing. End users now expect to perform the operation immediately. They might be on the phone talking to the customer about their transaction. The customer will not tolerate any unnecessary delays. So the ultimate measure of performance is the response time, the time taken from the user entering data or a command and the computer returning with the answer. Our impatient society now expects this to occur in the blink of an eye.

The focus on user response time requires attention in an application management plan. The dream of true end-to-end measurement, diagnosis, and analysis of user response time is not yet delivered completely by any vendor tools. The reason is clearly the lack of standardized exposure of this type of response time information by the various network, system, and application components. Nevertheless, a lot can be done in application management to measure various aspects of response time. We now examine various methods and technologies that can be used.

5.6.1 Simulated Transactions

A simulated transaction is one of the simplest methods of analyzing overall response time. It is like a transaction-level ping. The idea is to simulate true response time by issuing a dummy transaction that is harmless to the system. For example, for an email system, the simulation might be to send a test message to the email system. A database or ERP application might use a harmless read transaction. More adventurous would be to add and delete a record. This imposes a recovery data integrity risk if an outage occurs between the two, although this should not be a problem if adequate recovery tools are used.

Whatever the chosen transaction, the goal is to send a transaction and measure how long it takes. This measurement needs to be close to the users, so the architecture might involve “sacrificing” a PC to act as a monitoring station for each major group of users, according to geographic, physical, or logical user divisions.

The simulated transaction clearly has some disadvantages. It cannot usually diagnose the cause of a detected problem, because it does not get data that is granular enough to break response time into its components. Simulation also adds some performance impact to the network and application server from its additional transactions.

The main advantage of the simulated transactions approach is its simplicity. Provided an application has a client executable, a command-line interface or an API, usually you can quite easily set up a script that performs such a fake transaction. This script is then integrated into an application management tool that measures the time taken and alerts when it degrades. This method can offer both availability and performance alerts and can isolate the particular affected user population.

5.6.2 Network-Level Pinging

This method is an even simpler version of transaction simulation. It limits the data to the network level by analyzing response time for well-known network tests. The goal is to perform network commands such as “ping” or “tracert” for the network segments and paths used by user transactions.

This method has similar advantages and disadvantages to application transaction simulation. It provides a simple overall measure that is easy to add, but cannot offer detailed diagnosis. However, in some cases, it can be better at detecting where along the network path is a delay or obstruction. This method also clearly lacks any measurement of the response time for the application or database server.

5.6.3 User Activity Measurement

Another complex technology involves intercepting all client-side events and user actions. At the operating system level, it can intercept many low-level events. For example, you can detect user-level mouse and keyboard actions, as well as outgoing and incoming network events.

This method is clearly a high-technology approach. The low-level operating system interfaces are not always clean and certainly differ across platforms. The information available is not always complete. Even if all the information is available, this method still suffers from the same diagnosis problems as simulated application transactions. Usually, little granular data exists about the behavior of the transaction between the time it leaves and the time the results return. This means that you can measure and alert on problems, but cannot provide a detailed diagnosis identifying the offending component along the path. This fact is particularly true for three-tier or multi-tier application architectures.

The key advantages of this approach are that it is non-intrusive, does not increase network or application load, and does not need the cooperation of the application. This method is called

non-intrusive because it does not require any application or client modifications. Also, no additional transactions occur across the wire because this method only watches transactions made by the user. This method can also more accurately determine which user has the problem, assuming that the software is installed on each user client. You also have no sacrificed PCs in this architecture. Another key advantage of this method over simulated transactions is that the application does not need to be open to permit dummy transactions. Not all applications actually have an open interface whereby scripted transaction can be entered. For example, in-house or legacy applications might be very much tied to the client's GUI code internally, rather than cleanly separated at an API level.

5.6.4 Network Packet Sniffing

Another method of intercepting existing transactions is to watch the wire rather than the client. Various packet sniffing technologies can intercept network packets. Although originally used for network diagnosis, they have moved into the realm of application management. The PATROL Knowledge Module for Networked Applications is an example.

Sniffer tools make use of low-level network technologies to intercept packets. Generally, they need not be hardware components, but are software technologies that configure and interrogate the Network Interface Card (NIC) of the machine. Another key point about these technologies arises out of the network architecture of Ethernet and other protocols: the sniffer need not sit on either the client or server. Packets are sent around the network and effectively pass through every node on the same segment. So a NIC in promiscuous mode can intercept all packets to and from nodes within its segment. This fact is important because it means that a sniffer technology can sit on another box rather than directly affect the server or client.

Application management basically attempts to take this packet data and convert it into higher metrics such as application response time. This involves decoding the packets and is not a trivial technology. To decode the packets into their network-level data is comparatively easy. For example, you can determine the socket port number, packet size, originating host, destination host, and other low-level network data. Attributes such as these are standardized and can be extracted. This possibility allows a sniffer technology to easily build up some low-level application-level metrics by mapping port numbers and other attributes to the applications they serve. For example, all traffic to an application server on a particular port can be measured.

Another result of the low-level decoding of packets gives some network-level response time metrics. The sniffer can detect packet acknowledgments and retries, giving some measurement of latencies involved in transmitting a single packet. However, this does not accurately map to application-level transactions, since each such transaction may comprise multiple packets, and the packets containing the returned results are not mapped to the originating packets at these low levels. Hence, packet acknowledgment latencies give only a partial measure of application transaction response time.

Decoding the application-specific portion of the packets is necessary for full application-level response time measurement. This is a difficult task because this portion of the network

packets is not standardized and may differ not only across different applications (e.g., SAP R/3 versus Lotus Notes), but also across different versions of the same application. Hence, application decodes are difficult to do, are highly specific, and require either application vendor cooperation or effectively an exercise in reverse engineering of the protocol.

Once achieved, application-level decodes provide good application-level response time measurement. They provide overall transaction response times, and can also break them down somewhat into the various network segments and application server activity. They do not fully reach into the application to see what happens between the application server's receipt and return for the transaction.

Packet sniffing is also non-intrusive on the application and client sides. It does not affect the network traffic or processing at either the client or the server level. The only impact is the processing code for scanning and decoding each network packet. However, this task can be performed on a separate management node rather than consuming CPU time on the client or server.

5.6.5 Intrusive Instrumentation

The basic problem with all of the above methods is that they try to capture data from the outside. The promise of application instrumentation standardization, such as Application Response Measurement (ARM) and WBEM, is the exposure of detailed internal data. The ideal situation is that every hardware or software component along the transaction path emit data about the transaction. For example, the application client could expose when it started the transaction, each network node could report when the transaction arrived and departed, the application server could report when it received and forwarded the transaction, and the database could report details on its changes to the data records. Application and database servers should even report more granular performance data about where within the application or database the time is spent. Assuming the centralization of all this data, an application management tool could then not only measure the response time for a transaction, but also break out its time costs across all the software components and network paths. This capability allows problem detection and very detailed problem diagnosis.

The only real disadvantage of this method is that it hasn't happened yet. As we discuss later, many competing instrumentation standards are available, and not enough adopters of any of them exist. Even once all new hardware and software is better instrumented, non-instrumented older components will exist for as long as they remain. Until this situation clarifies, and certainly will eventually, the other external methods will be needed. The best solution in this situation is some combination of data from the various methods. For example, any of the techniques from simulated transactions, network ping, or packet sniffing can be combined with some form of non-intrusive application management at the server side. This type of combined architecture could monitor the overall response time and also break it down to some extent into the components of client activity, network latencies, and application processing time.

5.7 In-House Application Performance Instrumentation

Although few commercial applications are instrumented in a standard way, in one area, this value can be sought on home soil. In-house applications are an area within your control and also an area with a very high payoff from instrumentation. The use of instrumentation is not just for response time measurement, but also for generic exposure of performance data to facilitate better performance monitoring and automated tuning.

In-house applications of the past were not typically designed with monitoring in mind. Fortunately, a lot of performance monitoring can be performed externally to an application without its explicit cooperation. The operating system statistics for processes and files provide useful information on measures such as CPU usage, I/O activity, and file size growth.

Nevertheless, certainly a lot more can be measured if the application is designed to cooperate to externalize its measures. This fact must clearly be factored into a revision of the application, which involves development effort and the costs of a new application revision rollout. The benefits are there in the long term with an up-front cost from development.

Several levels of instrumentation exist. The simplest is to enhance the data that is sent to the application log file such that enough information is available to perform useful performance monitoring. For example, the application can log when it receives and when it completes a transaction. This information can be extracted by the application management tool to compute response times. This method shows the extra advanced performance monitoring possible, provided the application can be extended in this way.

A higher level of instrumentation is to provide a client executable that is specifically for monitoring performance and perhaps also for making administrative changes. This client executable can be automatically launched, either periodically or continually, to examine the measures it outputs. This client might well push through a dummy harmless transaction in order to measure its response time.

The highest level of instrumentation is to build specific application management components directly into the application. Typically, this involves using the API for the application management tool. This method of deep instrumentation is capable of full control. However, when considering the issue of performance monitoring, to add this API into the application is often unnecessary work, since the log file or client executable methods can be just as effective in computing the same performance measures. A full discussion of instrumentation via application revision appears in Chapter 7.

5.8 Application Development Architecture

Application instrumentation helps make an application more manageable and therefore indirectly improves performance. However, nothing substitutes for a bigger engine. The architecture of an application has far more impact on its throughput and performance characteristics than its openness to management tools.

The top-level view of application performance is most concerned with the architecture. Sometimes a different architecture is advantageous for human reasons, as is often the case with

new client/server projects. Other times, you have technical reasons to move to a different application architecture. Perhaps the current architecture causes too much network load as it centralizes the processing of all data.

The design of the architecture has to have the overall performance goal in mind, which depends on the type of application. The main criterion for a multi-user transaction processing system is the overall end-to-end response time. The aim is to minimize the time that the user spends doing nothing, in order to make better use of his/her productivity. With CPU costs relatively low compared to human employees, we are now in the situation where wasting CPU time can sometimes be desirable just to get better response to the users. User feedback dialogs to indicate the percentage completion progress of a transaction are such an example of “wasted” CPU time just to keep a user happy. Another example is sending a brief summary of data before the full data text, so that the user can cut short the screen once the desired item is found. For example, for an email client, should the server send it just the message subjects and author details, or the full text of the message? The email client interface should also allow the cutting short of a listing of all email messages as soon as the user double-clicks on one that appears. If instead the user has to wait for the whole list to update, such as if all messages are downloaded, this waiting is wasting time and considered poor performance. On the other hand, if the email server does send only the subject headings, then double-clicking on an item will have slower performance, because the client will have to request the message text from the server. This scenario shows the difficult balance involving user interfaces in end-user client/server applications.

In many cases, the design of the end-user interface is a key productivity issue. If the user has to go through a sequence of screens, each time waiting for the computer, this time is wasted. Perhaps not all of the screens depend on the sequence. Instead, the operator could move onto the second screen immediately after the first and pipeline the results to the application. Revising the application to handle just such productivity enhancements to the interface can give a large overall improvement in end user satisfaction. The underlying idea is the pipeline to improve the throughput of user transactions rather than a sequential ordered process.

Not all applications are interactive, although they seem to be the trend. For applications where the user does not require immediate feedback, many techniques are available to improve the overall system performance. The idea of batching from mainframe days is very effective. The aim is to pump all the transactions into the system without waiting for the results of others. The results come out the other end as soon as they are available, giving high throughput at the cost of losing a clear relationship between the input and results of an individual transaction.

Another aspect of application architecture is its handling of network and database resources. The aim is to strike a balance with regard to the use of network resources for data transfer, and processor and storage resources for data analysis and archival. Modern commercial off-the-shelf applications and databases are becoming more flexible in their possible architectures in order to offer better performance options. Choices between architectures such as two-tier or three-tier are now available. These give more choice in how to distribute the processing, control the network traffic, and decide which processors perform what work.

Service Levels and Capacity Planning

From the initial chaotic beginnings decades ago, IT management has moved into a more professional era where measurement, management, and forecasting are starting to play their part. Perhaps this is made possible by improving technology that frees administrators from much of the day-to-day chaos, or perhaps it is due to years of wisdom about the value of measurement in the process. Whatever the reason, the concepts of Service Level Management (SLM) and Service Level Agreement (SLA) documents are a rising modern trend. Service level management is combined with formalized contract agreements that are applied to both internal and outsourced IT operations. This sets the standard for the desirable level of service, from a user or customer perspective, and shows in black and white whether that goal was achieved.

6.1 Measurement Levels

You can rate the measurement and planning level of an IT department according to the different disciplines that are adopted. With the rise of expertise about information systems management has come a range of activities that an IT department should undertake. These levels of automation can be categorized as follows:

1. Fire-fighting
2. Automation
3. Measurement
4. Prediction

The first level is a chaotic level where each day brings new surprises. This is the introductory phase of an IT department. It involves the absence of any tools or directed automation. Problems are solved in an ad hoc manner.

The second level involves the introduction of tools for automation via application management tools. This provides immediate relief by proactive detection of problems combined with either automated correction or diagnostic aids. This cuts down on the burden that availability and performance demands provide. Reduced burdens permit activity toward raising the standard to the next level.

The third level is measurement of the overall process. This involves taking a step back from the battle-front and examining the overall situation. Measurement is the only way to know the true picture and also to know whether progress is being made towards improvement. The various aspects of measurement include the tracking of problems in a centralized service desk and the adoption of measured Service Levels.

The fourth level is the highest layer of sophistication through long-term predictive planning. This is the discipline of capacity planning that measures predicted future demands against the technical capacities that must support the business load. Prediction is the last step through the chain from initial reactive fire-fighting to localized proactive automation to overall measurement to future planning.

6.2 Management Trending Reports

The initial level of measurement of the organization is the creation of basic trending reports. Management reports typically require weekly or monthly summaries of the overall status of a situation. The data must be rolled up into daily, weekly, monthly, and yearly summaries. These allow trends to be discovered, service levels to be analyzed, and planning for future capacity needs. Some of the report types that are useful include:

- Availability reports—for what percentage of time was the application available?
- Performance reports—what is the average CPU usage or application response time?
- Incident reports—how many alarms were generated?
- Responsiveness reports—how quickly were alarms acknowledged, and how quickly were they resolved?

The basic goal of these reports is to provide overall information on the general situation. The reports should not focus on the exceptions, but instead should show the general trends. These allow IT managers to assess the overall effectiveness of the technology infrastructure. These reports need not necessarily be externally communicated, but rather used for internal planning.

The basic data for the generation of these types of reports is gathered by an application management tool. For example, incident reports are a simple measure based on the events generated. The output format of management reports need not be very complex, since these are

mainly internal management tools rather than official documents. Simple textual reports or HTML reports are easy to generate without requiring complex reporting tools.

6.3 Service Level Management

Service Level Management (SLM) is a more formalized generalization of management trending reports. The issue of effectiveness measurement is a well-known requirement for running a business. The same applies to running an IT department. To measure effectiveness in a fire-fighting mode is hard. But once a sane, automated process is in place, measurement is the means to the next level of effectiveness. The aim is to clarify the overall situation from the many data points.

Measurement is the key issue with service level management. This measurement brings the benefit of clarity and objectivity to a complex high-level management dilemma. Without any quantitative measurements, how does the Chief Information Officer (CIO) measure the departmental progress, and how does the Chief Executive Officer (CEO) measure the CIO? Must the CEO form an opinion on the basis of corporate users, who are never satisfied with application availability levels or help desk call-back time? The truth is that the CEO will never hear about when the IT department does a good job and really only the volume level of the complaining will decrease. The absence of clearly defined metrics leaves the IT department at the mercy of perception.

The promise of SLM is to measure the overall effectiveness of the IT department in serving its internal and external customers. These real numbers quantify the performance of the department. They are tangible measurements of the reliability of the system for the end user. Instead of being held hostage for an outage on one day, the overall trends across weeks and months become clear. Trends such as seasonal variations and end-of-quarter demands become clearly understood. The benefits of an SLM process include:

- CEO can measure the CIO's performance in the position
- CIO can measure his/her performance in the position and learn to improve
- Clear goals and objectives for IT personnel
- Feedback as to the effectiveness of certain practices or changes
- Outsourced IT departments can be more closely monitored
- CIO can show their departmental value for comparison with potential outsourcer promises
- User complaints can be met with tangible reports instead of perception control
- Benefits and returns from past IT investments can be demonstrated
- Justifications for new IT investments can include quantifiable benefit predictions
- Chargeback for IT services becomes possible as measurement becomes more granular

These types of advantages are available once a service level management process is put in place—not to say that this is easy to introduce, nor that the benefits are immediate. SLM is a long-term strategy to increase the effectiveness of the IT infrastructure and raise the visibility and perceived value of the IT department by end users and senior executives.

6.4 Service Level Agreements

The basis of management of service levels is official contract documents. The typical name for these is the Service Level Agreement (SLA). This is a formal agreement between end users of technology and the suppliers of the technology services. This model applies regardless of whether it is an internal IT department with internal customers from other business departments or external outsourced IT providers providing the same services.

A formal model is clearly required for external outsourcer arrangements, because of the involvement of different corporate parties. However, the same benefits apply to the internal model, since the only real difference is organizational—i.e., of which corporation IT personnel are a part.

Defining a realistic service goal is the most difficult part of the process. This is basically a negotiation process, either internally or externally. End users and customers, including internal customers, all want the best service that no money can buy. And the IT departments want to ensure that the requirements are achievable without working 80-hour weeks. Somewhere in the middle lies a realistic and satisfactory service level agreement.

The first issue to resolve for an SLA is what technology areas it will cover. The clearest candidates are the most mission-critical applications such as online applications for 24x7 user transactions. The web site is clearly a mission-critical application for many organizations, either from the sales it generates, or purely as a marketing tool. Another major candidate is the email system, since many businesses have begun to depend on email either explicitly or implicitly as it becomes the basis for keeping people informed about business operations.

The second issue for the SLA is to determine in detail what to measure. A great many metrics could be presented. Some of these are from the user perspective, whereas others are from the technology point of view. Measuring the user perspective is preferable, because this is a more accurate measure of the value of the technology. From the user point of view, some of the metrics to collect include:

- Availability
- Response time
- Throughput of transactions
- Resolution times for help desk calls
- Change or upgrade request service times

These are some of the more quantifiable metrics. Also, other intangible measurements can be gathered, via surveys or other methods, including user satisfaction and perception of IT value.

The other side of service levels is the technology side from the supplier point of view. Some overlap occurs here, such as availability measurements, but some measurements are not even seen by the end user. Not all problem alerts are indicative of problems seen by the user,

either because they were not serious, or because they were detected early enough so that they were resolved before affecting the end users. Some such measurement metrics include:

- Availability
- Events and alerts generated
- Response to events
- Resolution times for events
- Performance metrics

Measurement of service levels is an area where conflicting goals may interfere with the correct operation of the SLA. The measurement must be accurate and objective. Be sure to define how measurement of service is performed, and by whom, when developing an SLA document. The best scenario is to have the users measure the service levels as the services affect them. However, the technology providers are in a far better position technically to perform the measurement. Having the suppliers performing the users' measurement is clearly a recipe for subjectivity, either intentionally or unintentionally via natural human tendencies. Various solutions include checks and balances on the measurement process, such as joint involvement, duplicate measurements for comparison, or the use of an independent third party.

Chargeback is another consideration when building an SLA. Chargeback involves measuring resource usage on a per-user or per-department basis and charging the users for the amount of resources that they use. Sometimes this can be real billing or merely chargeback incentives. The aim of incentives is to force users to consider what their requirements are costing, and then factor that into their usage profiles. Chargeback can take many forms, from internal reports to formal cost invoicing. For internal IT organizations, providing the customer departments with reports on their usage profiles can still be valuable. If nothing else, this shows them what value they are getting from IT technology. Whereas this is purely informative, the next level is to base some of the internal budget decisions for IT requirements on the usage by different departments. This could be purely administrative or could involve cross-charging of budgets or cost centers. More formal arrangements are required for the use of chargeback as an external billing mechanism.

6.5 Capacity Planning

Capacity planning and service level management are not often discussed together. The people in an organization responsible for these disciplines are usually separated. However, capacity planning represents the highest level of attention to service levels—prediction of future service levels. This prediction aims to ensure that service levels are adequate far into the future.

Capacity planning is the discipline of preparing for future usage and load requirements. This is the pinnacle of performance and availability management. At some point, the existing systems cannot be optimized further, and new investments in hardware or software are required. This requirement can be deferred by activities such as application management and performance

tuning, but not avoided in the long run. The aim of capacity planning is therefore to know when is the right moment to introduce new infrastructure capacity into the overall system.

Justification is perhaps the hardest part of capacity planning. We could define capacity planning as “the gentle art of convincing the CEO that you need more stuff.” Indeed, the convincing part is not always easy when the benefits are not so obvious. In a typical CIO-CEO discussion about your department, does the CIO ask for a strong sport utility vehicle, the CEO ask why a lawnmower can’t do the trick, and they settle on a small sedan? This kind of discussion is hard to avoid when the current state of affairs and future benefits are hard to quantify. Capacity planning involves the provision of this type of quantifiable analysis of the costs and benefits of new equipment, and the overall costs if this upgrade is not performed.

A clear need exists to measure the current state of affairs to gain an objective measurement of how things stand today. Similarly, you need to know the likely future impact of the addition of new load requirements or new applications into the environment. Thus, capacity planning breaks into two key areas:

1. Measurement of the present
2. Forecasting for the future

The “planning” part of capacity planning is not much use without measuring the in-place systems. Forecasting is not going to be effective without live data about loads and performance. However, an important part of forecasting is the decision on when to introduce new capacity. This is the balancing act between optimizing current capacity and upgrading it.

6.5.1 Capacity Exploitation

Any discussion of capacity planning tends to focus on the assumption that upgrading capacity is the panacea for the applications. Of course, this is often true in terms of the performance of the application, but it belies the cost of the upgrade. This upgrade cost is not only the price on the new hardware’s or software’s invoice, but also the cost of changing out the existing infrastructure. This change management cost includes testing the new equipment, planning for the change, and the downtime that occurs during the change. Also, you have the cost of preparing for a roll-back to the old situation in case of some unforeseen major problem.

These combined costs of capacity upgrades show that capacity planning is a choice not only about what to upgrade, but also about when to do it. The possibility of delaying an upgrade for a few months or perhaps years through better exploitation of existing capacity is a favorable financial position. Of course, this must not be done to the detriment of user performance, but often room for improvement is available in the existing capacity.

This optimization of existing capacity is where application management meets capacity planning. The performance measurement and tuning aspects of application management offer this functionality. The cause of performance problems is often not due to capacity, but rather due to excessive load caused by a small percentage of users. Monitoring the problem users and

tuning according to their resource usage are often effective in improving overall performance for all users, particularly for a system that has had a sudden and marked degradation in performance. This is almost certainly a case where application management can help diagnose the performance problem. Other key areas that application monitoring will detect include contention for resources and queue lengths. A major cause of response time problems is the contention for shared resources by different applications or transactions. This leads to queues for the resources, and the wait time in this queue is a major contributor to response time degradation. For this reason, you need to gather both contention and queueing metrics in application management. This includes aspects such as database lock measurements, CPU queues, I/O usage analysis, and other key resource areas. Analysis of this data can often yield clues as to the best optimization of the system with existing resources.

On the other hand, for a system that has had slow and gentle degradation according to gradual rise in its load, a capacity upgrade may be the only solution. But even in this situation, you may be able, via better balancing of the load, to exploit existing hardware and software capacity. This is often the question that measurement will answer, by showing resource usage profiles across machines and over time.

6.5.2 Capacity Measurement

Measuring the current state of affairs regarding capacity is a little more detailed than service level measurement. Service levels are more centrally focused around the external behavior of the system with respect to its users, response time, and overall availability. Capacity measurement requires more detailed internal technical information regarding the load that is applied to the system. Certainly the “effects” as measured in a service level metric are crucial, but so too are the “causes” of these effects, because these are the signals of what might need to be upgraded.

The groups responsible for capacity planning are commonly different from those whose responsibility is the daily management of downtime and performance problems. This decentralized responsibility can sometimes have the downside that the two groups choose independent tools and architectures for problem management and capacity planning. This is a highly inefficient architecture because monitoring and capacity measurement require exactly the same data.

The first step in capacity planning is the gathering of data about the current state of affairs. This data has to be gathered using a probe on each of the nodes. Here some overlap occurs between capacity planning and application management. The data required for monitoring performance is the same as that needed to record the performance characteristics of the application.

Once the data has been collected, it must be analyzed. This same data can be used for management reports, service level measurement, and capacity planning. These three areas are effectively levels of complexity applied to the data in terms of analysis. The highest level is the attempt to perform future capacity forecasting using this collected data.

6.5.3 Capacity Forecasting

Forecasting involves both business and technical aspects. In fact, the business aspects are the key inputs in a predictive model. What is happening to the business that will affect loads on the IT infrastructure? Is the company hiring more people, opening new offices, or expanding overseas? Is the corporate direction toward offering more external web access to customers, suppliers, or distributors? Does a cost-cutting trend of network computers replacing desktop PCs place more demands on servers and the network? These types of business questions give insight into the future load on the system.

Business predictions need to be merged with technology issues. The basic question is whether the current infrastructure will handle the predicted future load. In the long-term, the answer is usually “no.” The issue is more how to make best use of the current infrastructure and what transition plan to put in place to move toward the new capacity requirements. Monitoring and performance management help to delay the need for capacity upgrades by making the best use of the hardware and software. However, a five-year plan is certainly going to require capacity upgrades. If these can be delayed through careful measurement, the cost is deferred and the price of the new technology is likely to be reduced when purchased a little later. Capacity planning needs to ensure that this sort of cost delay tactic does not occur to the detriment of end user performance by careful timing of planned upgrades with regard to increasing business loads.

Capacity forecasting can be performed in various ways. These range from informed guessing to formal analytical modeling using gathered measurement data. Let us examine some of these different levels in more detail.

The basic level is to use measurement data and an informal process of analysis. This has an undeserved poor reputation because it sounds like gut feel or guessing in a void. In fact, the intelligent use of real measurement data in combination with a good understanding of the overall IT architecture is a very valuable practice. Getting the right people in a room with real measured data, covering both loads and performance characteristics, can give good results. Of course, putting those same people in the room without the data is a weaker exercise, best to be avoided. The presence of real measured data is critical to proper understanding of the status quo and making informed predictions upon which plans can be made.

The major alternative to the experienced use of rules of thumb is to perform some level of analytical modeling. This can range from simple statistical extrapolation of measured data through to full theoretical models of the entire environment. The trade-off here is clear. Basic statistical analysis is simpler to achieve but weaker in predictive accuracy. Modeling is more difficult to achieve but offers the power for any level of accuracy.

Modeling is the most advanced form, because it expresses all the interdependencies of the different components in the environment. This completeness also makes it the most difficult to implement because of the cost of creation of the model. The basic requirements for a realistic model of the overall system include:

- Component inventory—network devices, servers, and applications
- Component relationships and dependencies
- Current measurements of performance characteristics
- Current measurements of load characteristics
- Predictions of future loads—e.g., due to business plans

The simplest form of model is that representing one component. For example, a model of a database server can be useful even without other components if the incoming transaction rates and the effect they create on the database are adequately described. Single node models are therefore a useful first step in performing capacity modeling.

Although localized models are interesting, the full power of modeling techniques occurs in expressing the effects of one component on another. Statistical models are poor at representing this relationship because it may not follow a well-defined mathematical relationship. This limitation leads to the need for some other way to represent the component dependencies in a way that permits detailed analysis. Some of the major categories of analytical models include:

- Discrete event simulation
- Queueing models

The simplest form of analytical model is to break application events into discrete events within the system. The impact of these events can then be propagated throughout the model. Simpler statistical models have difficulty in expressing the exponential problems that occur as multiple applications or transactions attempt to access a shared resource. This dependency of performance on the overall load is typically not a simple mathematical function, but a characteristic of the particular entity under different types of transaction arrival models. Discrete event models overcome this problem by treating each component and event as a separate item. You can then run sequential models to determine the actions of the system over time with different event profiles.

The generalization of discrete events is to express them in general mathematical terms. Rather than listing them as a sequence of points, they become a mathematical formula for their arrival distribution. This type of model is a key capability of analytical modeling systems. The representation of queueing processes, such as contention for a resource, is important in that it affects many distinct elements of the model. For this reason, queueing models play an important part in analytical modeling for capacity planning. These queueing models require the inputs such as transaction arrival rates and distributions and then apply the queueing analysis to show some of the expected end behavior.

A clear advantage of modeling is the ability to view the system according to different configurations or load changes. The other advantage is the capability for addressing multiple scenarios. Making statistical projections from gathered data does not permit the analysis of entirely different architectures. Using theoretical models allows the capacity planner to experiment with

different architectures and their characteristics. This creation of multiple models is no small feat, but it allows early analysis and detection of major pitfalls in architectures under consideration.

Since this modeling is all a form of simulation, the accuracy of the model is of paramount importance. Many factors affect the overall accuracy, including the validity of assumptions, measured data, component models, and the overall modeling algorithms. Various ways are available to test the accuracy of the model. The simplest is to attempt to input the current scenario of the IT architecture that is in operation, and compare the predictions made by the model with what is actually seen in the environment. Another way is to vary the basic parameters to see whether this leads to believable variations in predictions. However, it might also indicate a chaotic system rather than problems with the model. The accuracy of the model can also be tracked over time, particularly with regard to whether its predictions about a planned configuration change are close to the observed behavior after implementing the change. This tracking means that analytical modeling is an ongoing process and involves not only early planning, but also postmortem analysis of the validity of the models that were used.

6.6 Application Design Planning

In-house applications are an important part of future capacity concerns. A strong impetus exists to make better use of IT technology because it can bring major business gains. However, this must be done in a way that does not over-strain existing infrastructure resources. The temptation during application development is to do what needs to be done without considering the impact. Fortunately, application developers tend to enjoy performance tuning and will respond to interest in the performance of their application. Unfortunately, developers are also often poor at guessing where the performance bottlenecks of an application will be, particularly with respect to the differences between test and production demands on the application.

Measurement of the performance characteristics under realistic test situations is the best solution. This method has a few major limitations. Firstly, it assumes that a test situation will be realistic compared to production usage, which will require a lot of effort to get right. Secondly, this measurement can occur only rather late in the game, too late to make a major change in the application architecture if the performance characteristics are too weak. Leaving capacity analysis of a new application too near the end of the project risks wasting resources from late re-writes or even abandonment of the project. Cheaper in the long run is to ensure that capacity constraints are considered early enough to avoid this risk.

Early analysis of capacity requirements of an application is difficult. It involves the development of a realistic understanding of the performance characteristics of the proposed design. This might be via a detailed model of the architecture, or just an intuitive feel for how the application will handle a load. In either case, these performance issues need to be paid attention early on in the development cycle. The initial specifications should have performance as a key constraint, and ongoing milestones such as prototypes should be reviewed to see whether performance constraints are met. A little planning in the short-term can save a lot of embarrassment in the long-term.

Application Instrumentation Standards

Standardization is an ongoing trend in computer technology. These days we all take for granted the standards that allow us to network computers easily and print documents on a huge range of printers. Network management gradually evolved into the modern state where SNMP is the predominant management technology. So too, it is expected, will application management technology evolve to take advantage of some standard methods or technology. The basis of this standardization will ultimately be instrumentation of all applications.

7.1 Application Instrumentation Defined

The term “instrumentation” has become somewhat confused. Strictly defined, application instrumentation refers to the process of revising an application to make it “instrumented” with management information. However, some discussions use instrumentation to define any method of annotating the application, regardless of whether it actually intrusively changes the application. I use the strict definition here.

Instrumentation represents the deepest level of planning for application management. This means changing and even redesigning parts of the application to allow it to be managed more easily. Instrumentation is an option for in-house developed applications and not for commercial off-the-shelf applications (except for the vendors themselves).

By instrumenting the application, you have full control over what is measured, monitored, and managed. This attention to manageability certainly offers improvements to the application management process. For example, care can be taken to ensure that some metrics such as transaction throughput and end-to-end performance can be measured, because these are usually difficult (but not impossible) to measure without instrumentation.

7.2 In-House Application Instrumentation

The use of instrumentation for in-house applications is clearly a major issue. The basic choice is whether to use an intrusive instrumentation approach or a non-intrusive method. Intrusive instrumentation requires changes to the application, whereas the non-intrusive approach is to make the best of the already existing interfaces. The basic trade-off here is more development effort for more management features. Intrusive instrumentation offers the potential for better application management at the cost of additional development effort.

Unfortunately, development effort is not the main cost factor since both intrusive and non-intrusive instrumentation require development effort. The larger concern for instrumentation is the cycle time required to address intrusive application management for in-house applications. An intrusive solution requires a revision of the application itself, which includes the full development cycle, from requirements gathering, through development and testing, to redeployment of the in-house application in a production environment. The current version of an application cannot benefit from application management through instrumentation, and this fact delays the return on investment.

For this reason, consider a non-intrusive solution as an interim application management plan. A non-intrusive solution can manage the production application without revising it and usually without even bringing it down. This can also often be built by a member of the administration staff rather than deferring to the application development group.

Non-intrusive management and intrusive instrumentation can work together. Non-intrusive management can add value to the existing deployed application, and instrumentation can be planned for in the next release of the application. In fact, the plans to revise the application may well already be under way, allowing the possibility of adding instrumentation to make administration easier.

7.3 Year 2000 and EMU As Instrumentation Opportunities

Two items loom on the horizon for IT managers in the next few years. The first is the Year 2000 issue, affecting applications world-wide where two-digit date representations will potentially cause software failures. The second is the European Monetary Union (EMU) changes, preparing for a common European currency (the “Euro”) and affecting European IT departments. These changes are portrayed as huge problems by the press at large, to whom the glass is apparently always half empty. Indeed, the average pundit might almost expect that all computers will spontaneously explode as the clock strikes midnight in 2000, after listening to some of the talk in the community. But these changes can also be viewed with a glass-half-full attitude and seen for the opportunities they offer to IT managers.

The main opportunity is to change your corporate applications for the better. Since EMU and Y2000 are well-publicized changes for which the need for development effort and funding is widely supported, gaining the IT budget to revise your application is not a hard sell anymore. It might be the chance you need to move the corporate application architecture into the client/server world. And it can easily be the opportunity to solve one of your biggest IT management

issues, that of the management of your in-house developed applications. The best way to take advantage of this opportunity is to instrument your application to allow the creation of an improved application management solution.

7.4 Instrumentation from an Application Vendor Perspective

Instrumentation is an issue not only for end-users and in-house developers, but also for two types of software vendors:

- Application vendors
- Application development tool vendors

Both of these vendor classes would find manageability a useful feature to add to their products, for the competitive advantage. This statement is true for both packaged applications and the tools with which to build them.

Application vendors can benefit from application management through the lower cost of ownership for their application. However, the adoption rate for intrusive instrumentation of any kind, including the various standards, has apparently been low. Few vendors offer more than an SNMP MIB to allow SNMP-based access. Nevertheless, certainly in the future, the application vendors will move to a more intrusive form of instrumentation, although what form it will take is as yet unclear.

Development tool vendors are in a similar position. Offering application management as a standard feature of their development tools can only add value to the applications thereby produced. Gartner Group reports that some vendors such as Forte and Unify already have end-to-end performance instrumentation information added to their feature sets. In the future, all of the major tools likely will offer some form of manageability support.

7.5 Current Application Management Standardization Levels

An interesting issue is the current level of standardization of management issues across the various business application layers. These layers include the network, hardware, operating system, database, middleware, and the application.

The network management area is largely standardized upon SNMP as the protocol and data model for network management. The purpose of SNMP was to be “simple” enough that the managed devices could contain SNMP server-side support. The success of this approach shows in the fact that even hardware components have been instrumented with SNMP support to allow low-level remote network management.

Standardization of the monitoring and management issues for operating systems has only partially succeeded. The POSIX standard has somewhat achieved a common level of functionality across the UNIX variants. However, even the area of GUI views differs across the major operating system platform groups. A hardly surprising occurrence is that the management issues across areas such as Windows NT, UNIX, OS/2, VMS, and MVS could not be more different.

Each vendor provides its own tools for monitoring and administrative commands. Even the relatively similar UNIX variants offer monitoring tools and administrative commands with different syntax, layout, and functionality.

Database standardization has occurred to some extent in the prevalence of the relational model and the SQL query language. However, the issues of management of databases have only recently received standardization interest. Currently, no standardization exists for the management requirements for the databases from the major vendors.

Middleware and application levels have basically no standardization. Some have SNMP instrumentation with their own MIB in a vendor-specific manner. Clearly, the issues of the management of middleware or application components are new, and no standardization has occurred within any of the vendors in these areas.

Interestingly, the main area of standardization for all layers of application management is SNMP. Many operating systems, databases, and applications have SNMP support via their own instrumentations and MIBs. Unfortunately, the SNMP model is too limited to encompass all application management requirements, and is mainly used for monitoring purposes.

Various reasons explain the slow addition of instrumentation for application management to off-the-shelf applications and development environments. Application management is a relatively new discipline, and instrumentation of the applications to support it takes time. Another issue may well be the sheer number of standards, as you shall now see.

7.6 Application Management Standards Efforts

A number of efforts are ongoing in the area of bringing some standardization to application management. The various attempts at application management standardization using more capable management instrumentation functionality are yet to be widely adopted by application vendors or application development tool vendors. Some of the various efforts to define application management standards include:

- DMI/DMTF—Desktop Management Initiative/Desktop Management Task Force
- AMS—Application Management Standard
- ARM—Application Response Measurement
- MOSES—Massive Open Systems Environment Standard
- WBEM—Web-Based Enterprise Management
- SNMP application MIB (appl MIB)
- JMAPI—Java Management API
- WMI—Windows Management Instrumentation
- UMA—Universal Measurement Architecture

If nothing else, this list lends some credence to the cynical joke that “the best thing about standards is there are so many to choose from!” But it is only a matter of time before this list gets shorter.

The long-term dominance as a management standard is the reason for the battle. All of these efforts have their sponsors in the corporate world. Many of them have groups of corporations involved or endorsing the effort. Despite their corporate backing, most of these standards are offered up as open efforts for the public good.

7.7 Desktop Management Initiative (DMI)

The Desktop Management Initiative (DMI) is an initiative of the Desktop Management Task Force (DMTF). The DMTF can be found on the Internet at <http://www.dmtf.org>. This standard requires a Management Information File (MIF) format that describes the desktop assets, both hardware and software. This has had some success in terms of standardizing the software assets on low-end desktop machines. Vendors such as Hewlett-Packard have shipped personal computers with DMI enablement.

The first phase of the MIF format in the DMI was a mainly static standard useful for inventory and asset management functionality. To realistically achieve the more dynamic goals of application management on its own is unlikely. Hence, the MIF has been extended by standards such as the Common Information Model (CIM) and AMS incorporation. In fact, the WBEM initiative recently transferred to the DMTF organization. Hence, the current status of the DMI cannot be examined without considering the WBEM and AMS initiatives as a whole.

7.8 Application Management Standard (AMS)

The AMS is a standard produced and backed by Tivoli Systems. However, it is an open standard and has a large number of endorsements from other organizations. It is largely based upon the Management Information File (MIF) format that is part of the DMI contribution.

At its simplest level, AMS involves providing a description file, called an ADF (Application Description File), that describes the application. This ADF is the conceptual model that represents the information about the application and how to manage it. The attributes of an application that need to be described are aimed at covering the application through its life cycle. They include file components lists and other information useful for distribution and deployment, and interface descriptions for monitoring and administrative actions.

AMS is not necessarily an instrumentation standard, since it involves merely these external interface definitions. Therefore, the AMS enablement of an application could conceivably be done by a third party rather than the original application developer. However, instrumentation of the application is the best way to take advantage of all the AMS capabilities. Overall, AMS is a well thought-out standard that has the potential to be a long-term player.

7.9 Application Response Measurement (ARM)

The ARM standard is currently supported by a large number of major corporations, including Hewlett-Packard, Tivoli Systems, and IBM. The ARM standard is aimed mainly at the performance management area, and in particular, at issues such as end-to-end response time.

ARM is an intrusive standard that defines the instrumentation required to achieve performance analysis, and particularly end-to-end granular performance management. A variety of ARM API calls are defined that an application should use to instrument itself with this information. A management tool can then use the information to give advanced performance bottleneck diagnosis and response time measurement.

The ARM standard is one whose presence in the industry seems to be increasing. Even though this standard is not as all-encompassing as others such as WBEM or AMS, it has the potential to become a common method of performance instrumentation. In fact, one can envision that ARM could become a data feed of instrumented performance data to the higher-level standards.

7.10 Massive Open Systems Environment Standard (MOSES)

MOSES is an effort sponsored by Uniforum that produced a lot of useful work in various areas of systems and applications management. The effort seems to be predominantly UNIX-based for this reason. Unfortunately, the MOSES effort does not currently seem to be attracting much attention.

7.11 Universal Measurement Architecture (UMA)

The X/Open group has a standardization effort called the Universal Measurement Architecture (UMA). This is another standards attempt at defining the measurement of computer software for issues such as performance and capacity. This standard does not appear to get much attention in industry or in the trade press. Further information can be found at www.opengroup.org, which replaces the older web site, www.xopen.org.

7.12 IETF SNMP Application MIB

The Internet Engineering Task Force (IETF) has many initiatives and working groups. A number of these are related to SNMP and its role in systems and application management. SNMP is already the most adopted standard for network management. In addition, many software applications are instrumented to offer their own MIB. Notably, all the major RDBMS vendors and many major application vendors offer SNMP MIBs for their software. These MIBs all tend to be strikingly different, and the standards bodies are attempting to come to a more standard form for these MIBs. Hence, IETF working groups are defining a standard RDBMS MIB and a standard application MIB. These are attempts to standardize on SNMP representations of databases and applications to offer more standardized application management.

Certainly, a MIB and SNMP can expose any useful instrumented metrics. The main concern with this effort is whether SNMP is powerful enough to manage applications in an event-driven asynchronous manner, and not impose a polling non-scalable architecture on the management console. This seems an unresolvable issue unless SNMP is improved in the asynchronous area, such as the more generalized use of SNMP traps for a publish-subscribe methodology.

Monitoring is not the only area of apparent limitation. The capabilities for administrative actions to be issued via SNMP seem too limited. For SNMP to be a major option for future application management solutions, it needs to address these major architectural issues.

On the other hand, SNMP has the strongest history and largest head start in terms of existing widespread industry support. An SNMP MIB is the only significant management component added to many commercial databases and applications by corporate vendors.

7.13 Java Management API (JMAPI)

JMAPI is an effort sponsored largely by Sun, although it does have other support. It aims to leverage the power of Java into the systems and application management area. However, the key power of Java is its portable graphical capabilities, and this area is not that relevant to the server side of management. The choice of programming language to achieve instrumentation or to read values from an instrumented application does not appear to be important. This area does not use graphical features, making Java just one of many languages or access methods to choose.

On the other hand, Java's graphical capabilities may shine in the management console side of instrumentation. A standard method of accessing instrumented applications combined with a standard set of graphical management tools could be a powerful combination. Hence, although JMAPI does not appear to be garnering as much interest as AMS or WBEM, it cannot be ruled out, because it is based upon a fully functional programming language with potentially unlimited capabilities.

7.14 Windows Management Instrumentation (WMI)

Microsoft has a number of different initiatives related to systems and application management. For example, it has the Zero Administration Windows (ZAW) initiative to reduce total cost of ownership, the Microsoft Management Console (MMC), and the Windows Management Instrumentation (WMI) interface.

WMI defines a programming interface through which developers instrument their applications with management information. Although interesting in its own right, this is somewhat tied in with the efforts Microsoft is making with regard to the WBEM standard. The information published through the WMI layer would be managed by WBEM-enabled components.

7.15 Web-Based Enterprise Management (WBEM)

WBEM is a standard for enterprise-wide systems management that uses existing web technology to deliver management functionality. However, the evolution of WBEM has actually reduced the amount of pure "web" technology. The original concept espoused a web browser console and an HTTP-based access protocol. More recent changes have moved the direction away from this idea to that of support for any generic management client, and the use of other protocols and access methods such as COM. However, the Extensible Markup Language (XML) protocol may still be involved.

The WBEM initiative was formally launched in 1996 by five large companies with interests in the systems management arena: BMC Software, Cisco Systems, Compaq Computer Corporation, Intel, and Microsoft. WBEM is a publicly available standard backed by an alliance of vendors who will deliver on the technology. The public endorsement of WBEM by Computer Associates and the marketing power of Microsoft give this standard a great deal of potential. The original WBEM standard can be found at <http://wbem.freerange.com>, but it has now formally been transferred to the DMTF at <http://www.dmtf.org>. The most powerful technical components are currently those in Microsoft's WBEM Software Development Kit (SDK), which is a public SDK enabling anyone to interface with WBEM. This SDK is the basis of the PATROL Adapter for WBEM.

WBEM has a wider scope than most of the other application management standards. Whereas many of these define only application management rules, WBEM defines several major areas:

1. Data model (schema)
2. Standard schemas for common management objects (e.g., databases)
3. Protocol and access methods

The data model is called Common Information Model (CIM). It defines how management information is to be represented. This schema is then used to describe various managed objects, both hardware and software, so that the WBEM standard includes various standard schemas.

Perhaps the only standard that could claim to have been as far-reaching at its inception is the SNMP standard. SNMP does contain both a protocol and a modeling component for its managed objects. This power made SNMP well-established for network management. Unfortunately, although it has been used by some, SNMP does not appear to be powerful or extensible enough to accommodate the full requirements of applications and systems management. Hence, WBEM is analogous to SNMP in terms of its vision of both a protocol and a specification, but WBEM will not be limited to network management.

The movement of WBEM away from pure web technology has made it a more generic management technology. The concepts are quite general, with WBEM defining the data model (CIM) and also methods to both publish and subscribe to management data. WBEM defines "providers" that publish management data, and "consumers" that are clients that subscribe to this data. The concept of providers allows WBEM to control data from many other sources. Some of the already supported areas include SNMP, DMI, WMI, Win32, registry, and performance database.

Several factors make WBEM look like a candidate for the crown of application management standard. The backing of the Microsoft juggernaut seems sure to empower WBEM on Windows NT and other Windows operating systems. The extensible architecture of WBEM also means that it is powerful enough to bring together the disparate technologies such as SNMP, ARM, and even the various proprietary vendor-specific methods that we have not discussed here.

Future Trends in Application Management

Application management is a relatively new issue for corporate information technology departments. The issue arose from a number of trends in management. The application itself seems to be on the crest of a wave in terms of its overall importance to the enterprise. The trend of increasing complexity of applications leads to difficult problems in maintaining application availability. These lead to the trend of increasing importance for application management.

These are the trends that have brought us here. Some of these trends will persist, but others will come to the forefront. The internet brings with it a whole paradigm shift, and application management is already changing here. Applications themselves are changing to address manageability issues, bringing unheralded power to application management. Standards are growing to address the complexity issue of management, aiming to combat the trend of increasing application complexity.

Now take the time to examine a number of past, current, and future trends. One thing that is certain is that the application will remain the important component. As the hardware and network infrastructure becomes a commodity, the application still remains a complex creature. For this reason, if no other, application management is here to stay.

8.1 Client/Server versus Mainframe

The current direction of the trends regarding use of client/server versus mainframe depends on the predictions to which you listen. However, the trend in the past has clearly been away from mainframe and toward client/server systems. Not so long ago, industry pundits were predicting the imminent demise of the mainframe. In answer to these critics, IBM has posted excellent growth in MIPS shipments. The mainframe and the dinosaur don't appear to have as much in

common as some had believed. The current situation appears to be that although client/server or “open systems” has achieved significant usage, the mainframe still has a number of advantages under some conditions.

The client/server trend arose in response to a number of clear limitations of the mainframe combined with advances in other parts of the hardware and software industry. These advances made client/server feasible in technical terms. This allowed for an attempt to remove some of the limitations of the mainframe paradigm. The technical advances surrounding the growth of client/server are many and cover all layers of technology:

- Hardware and CPU power
- Networking hardware and algorithms
- Distributed databases
- Middleware
- Distributed transaction processing

The immense growth in CPU power made feasible a desktop machine with power many times the equivalent of the older machines. While also pushing along the power of mainframe machines, this technology made possible performing some very useful non-trivial business processing in far less expensive machines. Clearly, the cost of the hardware is another aspect that made possible using hardware in places other than behind the glass wall.

Networks are another major area of advancement. The speed and reliability of networks grew immensely, making them reliable for data and transaction sending. A distributed architecture is only as powerful as its weakest link. Without the advances in networking technology, client/server architectures would be much less desirable.

Countless advances in software technology have also smoothed the way to client/server. The software side of networking technology, with more reliable data protocols and compression algorithms, is a clear example. The distributed aspects of database technology are a discipline that has grown and now offers the reliability needed for distributing the processing of data. Middleware and transaction processing layers are another area affecting client/server architectures.

All of these software advances have helped pave the way for client/server. Of course, a snowball effect has also occurred, in that, once started, client/server architectures served to further push the need for advancements in these various areas. The industry viability of client/server made it attractive to make forays into these areas of technology. What was once at the forefront of theoretical algorithmic research is now everyday technology used in real client/server systems.

Not all the advances were technical in nature. The maturation of the management of distributed networks has played a major part in making them more reliable. This network management discipline involved advances in both technical software tools and also organizational techniques such as the centralized service desk. Network management was at first a black art but has become more of a science as it has grown to maturity.

8.1.1 Goals of Client/Server

All of these technical and organizational improvements at the various layers made possible early experiments with client/server architectures to achieve various desirable characteristics from the software systems. Some of these characteristics include:

- Reduced cost
- Better performance
- Improved accessibility
- Greater responsiveness

Cost is the primary driver for the client/server architecture. High mainframe costs resulted in the early attempts to use a client/server paradigm, aided by newer, more powerful machines. However, as I shall discuss in a moment, the cost benefits of client/server may not be so clear-cut, due to management costs.

Performance improvements are another clear goal of client/server. By using smarter clients, you have the capability to have better responsiveness to simpler processing tasks. For example, banking software can perform loan estimates and projections locally instead of going down the wire from a dumb client to the mainframe. Those tasks that cannot be performed fully in the client, such as performing an actual banking transaction, can be performed closer to the user via a distributed database. These options give an overall improvement in performance for the user.

Performance might just be another way to describe accessibility and responsiveness. The client/server model gives the opportunity for better responsiveness to the huge explosion in end users. This has further increased the load on business applications, causing the need for increased power. The performance advantages of client/server are a direct way to address these needs for greater access and faster response times.

8.1.2 Experience with Client/Server

Although the use of client/server is in its infancy, compared to mainframe, a number of clear trends in client/server have arisen. The distributed nature of client/server architectures brings with it a number of unique challenges that are still being met in ever-improving ways. The basic needs of availability, reliability, performance, security, and integrity are just as important, but require that greater efforts be met.

The good news about client/server is that it has delivered on many of its original goals. The advantages to end users are real, and user satisfaction is a reality for client/server projects. This satisfaction is driven by the real payoff in terms of performance, accessibility, and responsiveness. Client/server brings response time down and brings data more quickly to the user's fingertips. The increased power to the end user, and sometimes to the customer directly, brings with it tremendous productivity and overall business advantages.

What of the cost advantages of client/server? These are where the news is not so good, since client/server implementations have taken more time and money than expected. Gartner Group estimates the cost of client/server as 70 percent more than traditional mainframe, and cites that the majority of companies thought that client/server took longer to implement, cost more, and was more complex than expected. However, from the same survey, Gartner Group reports that only a tiny 4 percent considered the project a failure. This would seem to indicate that although costs are higher than desired, the positive end results in terms of user satisfaction and business value remain.

With the business benefits of client/server clear, the need to reduce client/server costs is paramount. These high costs arise mainly through the need to support a high level of complexity from multiple-vendor hardware and software, and the distributed nature of the components. Complexity can be handled over time, but client/server is a relatively new discipline that has not yet had time to evolve to offer more reliable solutions. For example, although distributed network management practices are widespread, the adoption of application management has only recently begun. The up-front management of complexity through application management is too often set aside, leading to a reactive support nightmare once the application is in place. Although the addition of application management to an existing deployed client/server project is a way to get this benefit, this is a request that senior executives are not expecting. Greater benefits can be had by early planning for application management, by including it as part of the project, just as network management is now an identified part of a network rollout project.

Application management is a concept that transcends the underlying technology and applies equally well to client/server or mainframe applications. The aim of application management is to bring management to the application level to deal with the application rather than with its underlying hardware or software technology layers. All of these considerations apply to both mainframe applications and client/server applications. Clearly, centralization of application management of mainframes is less of an issue, but the superior returns from an application-level focus still apply. Application management is a discipline that solves the basic needs of both client/server and mainframe applications by keeping them online, reducing failures through proactive prevention, recovering from failures, and identifying and removing performance bottlenecks. However, despite the basic benefits for both mainframe and client/server, client/server is likely the area with the greater costs. Client/server offers the potential for the greater advantages in implementing application management as a way to combat the costs of supporting the complexity of the client/server environment.

8.2 Application Availability Trend

The clear trend in corporate software is the central nature of the application. This may appear to be nothing new, since applications have always been at the heart of business data processing. However, what is new is the extent to which application availability has become business-critical. The traditional suite of business applications involves functionality components such as:

- Payroll
- Order entry
- Stock inventory management

How important is the goal for these applications to remain live? Some of these are clearly more important than others. For example, payroll is not really that critical, although your employees might not agree. But from a management perspective, a glitch in payroll processing is not so critical because it is not needed 24 hours a day, and a problem can be resolved without a fire drill. Payroll is far enough away from the customers to make the issues less dramatic.

Order entry and order processing are closer to the customers, making them more important. The level of importance here depends on the type of business. For example, orders for stock in a retail chain of stores are not exactly real-time because some delay will occur before an order entry or inventory problem affects what merchandise is on the shop floor. Certainly, the smooth handling of stock inventory is critical for business profitability, but a one-time outage of the inventory software is not going to cause customers to go elsewhere. Other businesses have more critical issues in order handling. For example, airline reservations failures will cause immediate loss of business revenue as customers turn elsewhere. These applications need to be online during business hours, and often 24 hours a day, particularly when the international factor arises. Some of the particular applications where downtime is most critical include:

- Airline reservations
- Banking tellers and ATMs
- Point-of-sale—retail shop floor purchasing software, pricing, credits, etc.
- Insurance agencies

Again, this idea is not anything new. Business application availability importance has always differed across the industries. What is new is the extent to which some of these applications are being opened up to customers, and also to business “customers” such as suppliers and dealers. Applications that are more open must be available when the customers want them, and must have adequate performance whenever they are used. This openness makes the availability of the application more critical to the core business functioning of the corporation.

The internet is one of the key driving forces in the increased need for online applications. The availability of the internet itself is becoming more critical. The availability of intranet servers is also increasing in importance as the intranet becomes more widely used for internal and sometimes inter-business information communication. Applications that are going online include examples such as:

- Internet servers themselves
- Package tracking applications for shipping companies
- Online banking transactions

All these trends add up to an increased value in application availability. The downtime of an application is becoming more significant, and maintenance downtime windows are reduced or disappearing.

8.3 Application Diversity Trend

An effect of the increased importance of the application appears in the mushrooming number of different types of applications. Commercial viability of application software has brought a huge range of off-the-shelf applications. This news is good in that the power of the computing infrastructure is enhanced by having these applications to perform the requirements of the business. However, this diversity brings with it increasing challenges for management, as much of this book testifies.

Commercial applications are not the only contributor. The incredible advances in software development technology have made applications no longer the domain of commercial entities. Departmental personnel are empowered with a variety of software development kits using 4GLs and rapid prototyping environments. Users are empowered with applications built quickly using Powerbuilder, and these start to generate unplanned SQL commands on databases. This is good news for users, but again potentially bad news for administrators, who have the difficult task of determining what poorly-written Powerbuilder application's SQL script is causing the new performance problems.

Empowered users are happy users, and client/server computing offers this situation. Software development tools and ad hoc reporting tools are examples of the new power in the hands of end users. The local processing power of the client side of client/server makes this possible. The advances in user interfaces, with the ever-present GUI, make these applications easy to use, and users begin to explore their capabilities, pushing the envelope of the overall system.

All of these different trends bring about the current incredible diversity of applications. The behavior of applications differs among different commercial vendors and even in-house developers. This is the challenge for the administrators attempting to manage the system. Clearly, the need exists for coherent application management to reduce the cost of supporting these applications.

8.4 Commodity Infrastructure Trend

Throughout the industry, software has become more important than the hardware on which it runs. The value of a computer is not inherent to the hardware, but is driven from what software can make the computer do for the user. This idea leads to a trend whereby all forms of application infrastructure are heading toward a commodity pricing model. The hierarchy of business application software infrastructure could be categorized as the following layers:

- Hardware—CPU, memory, disk, etc.
- Network—wires, routers, bridges, etc.
- Operating system—controlling execution, files, printers, etc.

- Database—data storage, querying, archival
- Application—business logic rules

Hardware is clearly a commodity market. CPU speed, memory, and disk storage are cheap, and getting cheaper. Hardware manufacturers are combatting this effect by marketing campaigns aimed at brand differentiation, but the hardware commodity trend is clear. Little inherent value is in a chip, some memory, and some disk, unless software makes the CPU do some useful computations, and stores some useful data on the disk.

Network infrastructure is also in the same category. As network capacity costs go down, a commodity market is created. Network devices are an infrastructure needed to support an application, but they have no inherent value without software on top of them. The standardization of network-related technology has left little room for distinction. Certainly, network vendors aim to differentiate based on performance and manageability issues, but the trend is here.

Operating system software is also subject to the commodity trend. Although vendors will try to differentiate their operating systems on fundamental issues like performance and reliability, the real value of an operating system is what software will run on it. No one will pay a lot for an operating system nowadays, but they all expect it to be there.

Will applications ever become a commodity? This question is logical and is the extension of the trend. How far will the commodity trend affect software? In a sense, applications should be immune from commodity pricing because the application's functionality is where the real value of a computer system is found. However, the underlying cause of a commodity trend is issues such as standardization, lower manufacturing costs, and resolution of customer expectations. These factors are all present in the application market, and some effects are already clear. The development costs of good software are way down now, partially because we have learned better software-building techniques, but more so because a lot of software has already been written and entire blocks can be reused. Not only do good programming libraries exist, but also application vendors already have working applications pre-built. The cost of providing end users with basic office desktop applications such as word processing and spreadsheets is now largely driven by the cost of software maintenance rather than new development, plus the low costs of duplication, packaging, and manufacturing.

Business applications have also started to be affected by this trend, although to a lesser extent. For example, databases are standardized around issues such as SQL, and offer much common functionality. Since the value of a database is not inherent but dependent on what the applications do to the data, these databases are clearly affected by the commodity trend. However, the applications that manipulate this data, which are at the end of the chain of infrastructure, are still the least affected by commodity pricing trends.

8.5 Application Vendor Management Features Trend

Vendors of databases and applications have started to recognize that manageability affects their bottom line. The cost of managing the application is an add-on cost that detracts from the value

of their application and takes money out of their pocket into the hands of Value Added Resellers (VARs) and implementers. Not surprisingly, the vendors are adding features to their products aimed at this very area.

Some vendors are adding their own specific management products. The problem with this approach from the IT point of view is the absence of coherent standards. This means that each vendor comes up with its own methods for management, and still no consistent method of handling management of multi-vendor enterprises exists. The vendors are the experts on their own application and you would think they should offer better management capabilities. However, manageability is more than internal knowledge about the application. The management capabilities are a sideline for these vendors, leading often to weaker architectures, and in some cases, leading to the abandonment of manageability projects.

Other database and application vendors are working with application management vendors to ensure that their databases or applications are manageable. Sometimes a combined approach is used by vendors by offering their own management solutions and also helping independent vendors manage their products.

Vendors of development tools for building applications are also seeing value from adding management features. Gartner Group reports that Forte and Unify now have instrumentation features such as end-to-end response time recording for the applications generated using these tool suites.

In the future, this trend will certainly increase and will move toward some level of standardization. This standard could arise from one of the various standards efforts, or some de facto commercial variant. Analysts such as Gartner Group are speculating that this standard might be based on WBEM, simply because of the backing from Microsoft, and the likelihood that Microsoft will work hard to make the standard a reality.

8.6 Web Management Trend

The phenomenal rise of the internet has begun to affect the markets for administration tools. This trend includes both the need to manage the internet itself, and the use of the internet technology to manage other environments. The management of internet components involves both server availability management and content management. The use of internet technology for management includes the use of browser interfaces and reliance on internet protocols.

Browsers are becoming the most requested user interface. An increasing number of end users are demanding the capabilities to view system status through a web browser. Managers are beginning to want summary reports available in formats viewable by a browser. The trend toward the web browser as the next-generation user interface seems clear throughout the computer industry as a whole, and also in the area of enterprise management.

Several of the current crop of application management standards efforts are based around the internet. The Web-Based Enterprise Management (WBEM) initiative is sponsored by a group of major vendors including BMC Software, Cisco Systems, Compaq, Intel, CA, and Microsoft. The Java Management API (JMAPI) is backed by Sun and the web power of the Java

language. These standards are indicators that the future of systems, network, and application management may well be in the internet.

A key issue is the use of web browsers as the user interface for management tools. The PATROL suite includes the web browser interface called PATROLWATCH for Web Browsers, which allows both Netscape Navigator and Microsoft Internet Explorer to be used as user interfaces. This product provides browser-based views of enterprise statuses, events, and alarms. The trend toward full browser enablement of the user interface for management tools seems clear.

8.7 Network Computers versus PC TCO Initiatives

The Total Cost of Ownership (TCO) of desktop PCs has become a major topic of concern to IT departments. Experience has shown that the cost of PCs is not just their initial hardware and software outlay, but also the intangible costs of supporting, upgrading, and tracking corporate-wide PC users. Two basic outcomes result from this raised profile of costs:

- TCO reduction initiatives
- Network Computers

Whereas TCO initiatives aim to reduce PC costs, the Network Computer trend aims to replace the PCs. Network computers are the thin-client web-enabled desktop machines. This initiative is the brute-force approach to PC TCO reduction, by elimination of the PC through replacement with a simpler machine. The advantages of Network Computers include:

- Initial hardware cost is lower.
- Centralized application servers reduce deployment and administration costs.
- Internet interfaces become the standard.

On the other hand, some of the potential disadvantages include:

- Upgrades are required to servers to support additional load.
- Network load increases from additional user traffic.
- Application performance depends on the network response.
- Acceptance may not come from users accustomed to full local control.

The main requirement for the success of Network Computers is that the critical applications can run on them. This basically requires web-based interfaces to front all the applications that were previously running on top of PC machines. This includes not only business-critical application clients but also personal software such as word processors, spreadsheets, and email. Fortunately for the Network Computer vendors, this trend toward web interfaces is already well established, with many applications already having interfaces for current web browsers.

In the face of this potential threat, PC vendors have adopted some aggressive TCO reduction strategies, for example, Microsoft's Zero Administration Windows (ZAW) initiative and

Intel's Wired for Management initiative. These aim to make the best use of PC technology and software by making managing large PC deployments easier. The overall TCO is reduced through eased management and control. This is directly aimed at addressing the areas that caused the TCO issue to arise in the first place.

Network computers and PC TCO initiatives are in direct competition for the same IT dollars. Interestingly, the widespread use of PCs arose as the client/server trend replaced mainframes with more distributed architectures, and now Network Computers move back toward centralization. PCs have a huge existing user install base, but some corporations have already announced large projects involving Network Computers. In the end, the market will decide.

8.8 Framework Trend

The adoption of systems management frameworks is a clear trend. Frameworks are an underlying technology that aim to provide the basis of all systems management tasks. Systems management products such as application management, scheduling, distribution, and security management can then be delivered on top of this framework level of abstraction. This common set of services for all Distributed Systems Management (DSM) tools aims to reduce the development effort to allow rapid delivery of advanced tools. Some of the commercial products that are often referred to as frameworks include:

- Tivoli Management Environment from Tivoli Systems
- CA-Unicenter from Computer Associates
- IT/Operations from Hewlett-Packard

The adoption of frameworks has been reported by various analysts. Despite their competitive positioning, Gartner Group even predicts that a large percentage of corporations will have more than one framework in use.

Application management is not a framework, but rather a service that can be provided on top of a framework. This explains why PATROL is not positioned as a framework and does not provide third-party software distribution, scheduling, or security administration services. In fact, rather than compete with or displace frameworks, PATROL integrates into all the above products via its PATROLVIEW integration components. This capability allows PATROL to fit into any environment. At user sites, PATROL is commonly used for application management at all its layers (operating system, database, middleware, and application), and a framework is commonly used for services such as centralized console, distribution, scheduling, and so on. In an environment with multiple frameworks, application management becomes a service that can be provided in a common way by all of the frameworks at a site.

8.9 Business Process Management

What is the next thing after application management? The issue of service level management is already well-known and becoming part of application management. The ultimate evolution

seems to be toward business process management. Logically, business processes are the next layer closer to the financial objectives of a corporation. The evolution from technology management to financial objectives management looks like:

- Network management
- Systems management
- Application management
- Service level management
- Business process management

The full development of features such as real-time business process monitoring is still relatively in its infancy. Business-level views are common via mapping existing measurements into a logical view. The next level is monitoring and administrative functionality based on more complex models of business processes. Capabilities such as problem diagnosis and business risk analysis require specification of the layout of a business process, both in physical and logical terms. This involves plugging existing infrastructure metrics into a site-specific model of how the business operates and what its overall financial goals are. This situation is even more general than reasonable high-level metrics such as service levels. However, even service levels still have a technology focus, whereas business metrics monitor and report on the real financial objectives of a corporation.

Application Management Practice

Introducing PATROL

PATROL is a suite of application management tools. This is where the theory ends and the rubber hits the road. PATROL provides application management throughout the entire application dependency stack, from the network and operating system all the way up into the high-level ERP application views. PATROL delivers the benefits promised by the theory of application management.

PATROL is based on one simple idea: software that automatically detects and corrects application problems. This simple idea gave rise to the PATROL agent architecture, leading to a sophisticated solution that was awarded a U.S. patent in August 1997. Conceptually, PATROL is software that sits on an application's machine and watches continually for application problems. This is automation of an otherwise human task. Traditional application management approaches have involved human administrators "watching" for problems through regular checking. More often, this approach leads to solving issues as end users start crying for help when the application goes down or slows to a crawl. PATROL automates this task by knowing what to look for and what to do about it.

PATROL knows what problems to look for because of the expertise about the particular application. PATROL's concept of Knowledge Modules (KMs) encapsulates this expertise. These are prepackaged information modules that tell the agent how to handle particular environments, databases, or applications. For example, the PATROL Knowledge Module for Oracle tells PATROL what problems to find in an Oracle database server and how to handle them. Some of the problems it knows to look for include availability and performance monitoring. Availability monitoring involves checking for a downed server, watching operating system processes, checking connectivity status, and scanning the error log for fatal messages. Performance monitoring involves measuring the database low-level metrics such as cache hit rates and also higher-level concepts such as throughput and response time.

Once a problem is detected, PATROL can do a number of things about it. The most powerful is to automatically run a command to correct the problem. This is the simple idea of fixing the problem once it is found. For example, PATROL can automatically relaunch a downed application server, or launch a particular tuning command once performance falls below a set threshold. This is rapid-fire automatic improvement.

Unfortunately, not every database or application problem can be automatically corrected this way. If it cannot be fixed, the follow-on simple idea is to tell someone about the problem. PATROL can also notify the administrator of the application about the problem. Notification can be as simple as a red flashing console icon, or as sophisticated as email or pager notification.

And once the administrator knows about the problem, PATROL's final simple idea is to assist the administrator in fixing the problem. For this reason, each PATROL Knowledge Module includes commands that can be run manually and expert advice to guide the administrator to the solution of a problem.

From these functions, we see that PATROL's fundamental idea is to maintain application availability by managing at various levels. It detects application problems, tries to correct them, tells administrators about them, and helps administrators fix them. This simple concept is that of using PATROL for application fault management. Of course, PATROL offers advanced functionality in other areas, but this basic approach to keeping applications working well by problem solving is at the heart of PATROL's application management.

9.1 PATROL Components

The PATROL suite is comprised of numerous software components. There are a variety of alternatives in using the different components in an application management solution. The list of key PATROL components includes:

- Agents
- Consoles
- Knowledge Modules
- Integration Modules

However, the basis of PATROL is still the agent and console, as with any application management method. These two components are the absolute minimum requirements for application management to occur.

The agent is the daemon software that sits on every managed server. Although it lacks the immediate visual appeal of a GUI, the agent is one of the most advanced pieces of PATROL technology. In particular, it is a fully autonomous agent that is capable of independent execution. I have much more to say about the agent later on in "Autonomous Agents" in Chapter 10.

The console is the graphical view through which an operator or administrator views the status icons and event lists (see Figure 9-1). The console is a critical piece for application management. It brings all the monitored data to one central console and allows remote control and

administration of the software applications. The main PATROL console is supported on UNIX and Windows NT and is separated into two key console types:

- Operator Console
- Developer Console

The Operator Console is the main production console for monitoring and administration of applications. The Developer Console includes these features plus a set of development tools.

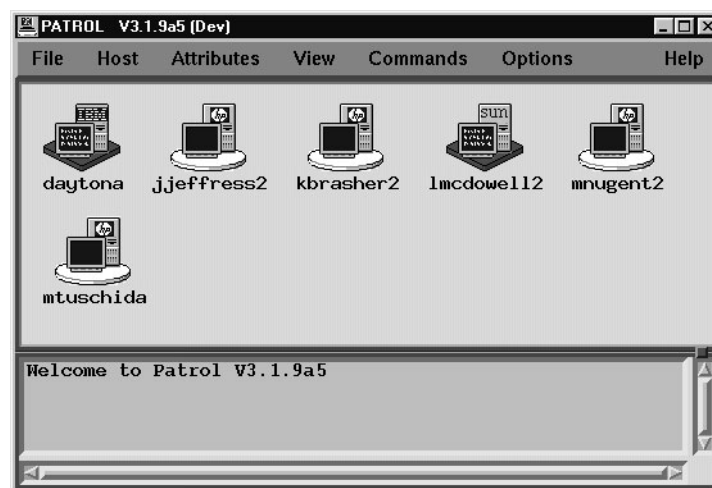


Figure 9-1 PATROL Console for Unix host-level view

Although the idea of agents and consoles is rather generic, the next key component is a more distinguishing one that is unique to PATROL. This is the idea of Knowledge Modules. These are effectively programs that contain the instructions that tell the agent how to monitor the particular software entity to which this Knowledge Module pertains. Some of the features that Knowledge Modules encapsulate include measuring values, alerting on problems, and auto-correction of these problems where possible. Thus, Knowledge Modules are prepackaged expertise about how to manage a software application. Each agent monitors one or more software components through execution of different Knowledge Modules. The agent without a Knowledge Module is just an execution engine without any work to do. The separation of the agent technology from the application-specific knowledge offers many advantages in terms of flexibility and extensibility.

The other unique feature of PATROL is the multitude of console options. PATROL offers a number of console alternatives to the UNIX or Windows NT options. Netscape Navigator and Microsoft Internet Explorer can serve as a console through the web browser console integration components called PATROLWATCH for Web Browsers (see Figure 9-2).

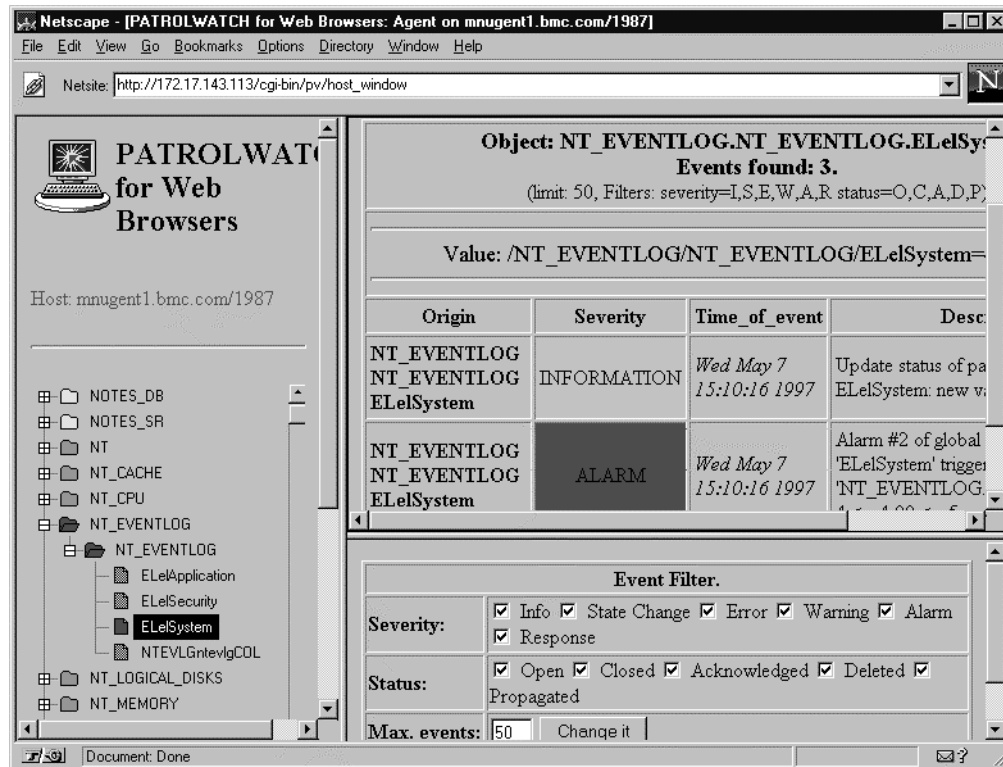


Figure 9-2 Netscape Navigator views via PATROLWATCH

Another unique feature is available whereby third-party consoles can be used with PATROL via PATROLVIEW integration components. This allows support of frameworks and network management consoles such as HP OpenView IT/Operations, Solstice Domain Manager, Tivoli TEC, CA-Unicenter TNG, and also help desks such as Remedy ARS. All of these various console solutions are satellites around the agent.

9.2 Autonomous Agents

The basic mechanism of PATROL is an agent on each monitored machine that sends information back to the console or to other integrated viewing components. This agent is autonomous, which means it can operate alone without any connected console or external interface. It will happily work away at the monitoring, alerting, and correcting activities based on the Knowledge Modules that it has loaded.

PATROL aims to provide agents to cover the management of any application on any type of system. The PATROL version 3.x suite includes coverage of over 20 hardware platforms,

numerous databases, various applications, and much more. The first coverage requirement is the hardware or operating system platforms. PATROL has autonomous agents for at least the following major platforms groups:

- UNIX Variants
- Windows NT
- OS/2
- OpenVMS
- OS/390 (MVS)

The list of UNIX variants that support a native PATROL autonomous agent include HP-UX, Solaris, SunOS, AIX on RS/6000, SGI Irix, Digital UNIX, and numerous others. Also, support for NetWare and AS/400 is available via specialized Knowledge Module architectures, even though no native agent currently exists.

9.3 Knowledge Module Categories

Knowledge Modules are another area where PATROL covers quite a range. Although the majority are delivered with the PATROL suite, some of these Knowledge Modules are also developed and/or delivered by independent partners. The Knowledge Modules available for PATROL cut across all the different layers of an application. Various Knowledge Modules exist at each of the levels, including:

- Operating systems
- Middleware applications
- Groupware applications
- Internet applications
- RDBMS databases
- ERP applications

The functionality provided by the Knowledge Modules is conceptually similar across all the levels. The basic functionality is monitoring, alerting, and corrective actions, i.e., detecting and correcting problems. The other half is administrative commands that package expertise in performing actions on the application or monitored entity.

9.4 Operating System Knowledge Modules

The PATROL Agent provides a basic operating system insulation layer over which the Operating System Knowledge Modules add monitoring and administrative functionality. This shows that the separation of the agent technology and the Knowledge Modules occurs even at the lowest layers. The agent is not hard-coded with any monitoring commands even at the operating system layer. Instead, the agent merely provides access to data collection methods that must be

controlled via a Knowledge Module. Knowledge Modules exist for operating system level monitoring such as CPU, memory, disk usage, printers, and so on. The following operating system KMs exist to add monitoring to PATROL:

- UNIX KM
- Windows NT KM
- OS/2 KM
- OpenVMS KM
- OS/390 KM (MVS)
- NetWare KM
- AS/400 KM

The monitoring functionality is both similar and different across the various platforms. The basic components that are monitored include the CPU, memory, filesystems, disks, tapes, processes, printers, and so on. These are largely where the similarity ends. The particular monitored parameters and administrative commands distinguish the different operating system features and idiosyncrasies.

9.5 Database Knowledge Modules

The list of databases for which Knowledge Modules exist include those below, all typically supported across multiple platforms.

- Oracle
- Oracle Symmetric Replication
- Oracle Parallel Server
- Informix
- Sybase Replication Server
- Microsoft SQL Server
- Rdb
- DB2 Universal Database
- DB2 for OS/390
- CA-OpenIngres and CA-Ingres
- PROGRESS
- ADABAS

The functionality provided in the various RDBMS Knowledge Modules includes standard functionality and much database-specific functionality. Standard functionality includes availability parameters, performance parameters, capacity/workload parameters, administrative menu commands, reports, and SQL capture.

This list of database Knowledge Modules remains up-to-date with the new releases of each commercial database and continues to support older versions. For example, the PATROL Knowledge Module for Oracle supports Oracle8, in addition to version 6 and 7 releases. Any databases not on this list may have Knowledge Modules available via third parties in the PATROL Developer Network.

9.6 Groupware, Middleware, and Internet Knowledge Modules

PATROL also has support for various groupware applications and middleware components via Knowledge Modules specifically for these components or as part of other Knowledge Modules. Some of the major ones include:

- Lotus Domino/Notes
- Internet Servers—Netscape, Microsoft IIS, Cern, Apache, etc. (see Figure 9-3)
- Microsoft Exchange Server
- MQ Series
- IMS

The monitoring of these various technologies is important to application management. Messaging and transaction middleware functionality must be monitored in a distributed application. Groupware is often mission-critical in that it becomes an expected part of internal corporate culture.

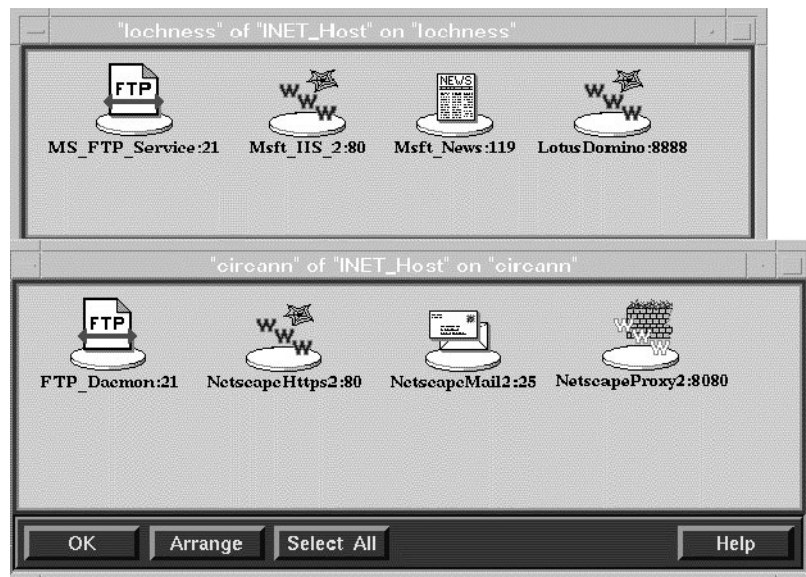


Figure 9-3 Monitoring internet servers including Lotus Domino

9.7 ERP Application Knowledge Modules

The enterprise resource planning (ERP) applications are among the most vital business applications. PATROL provides off-the-shelf monitoring and management components for many major ERP applications. A large number are supported already via PATROL, and more are available through the PATROL Developer Network (PDN). The ERP applications that are some of those supported via Knowledge Modules include:

- SAP R/3 (see Figure 9-4)
- PeopleSoft Applications and PeopleSoft HRMS
- Baan Applications
- Oracle Applications

The ERP applications are the top level of PATROL's application management. PATROL has an range of different ERP products that take data from the full application stack.

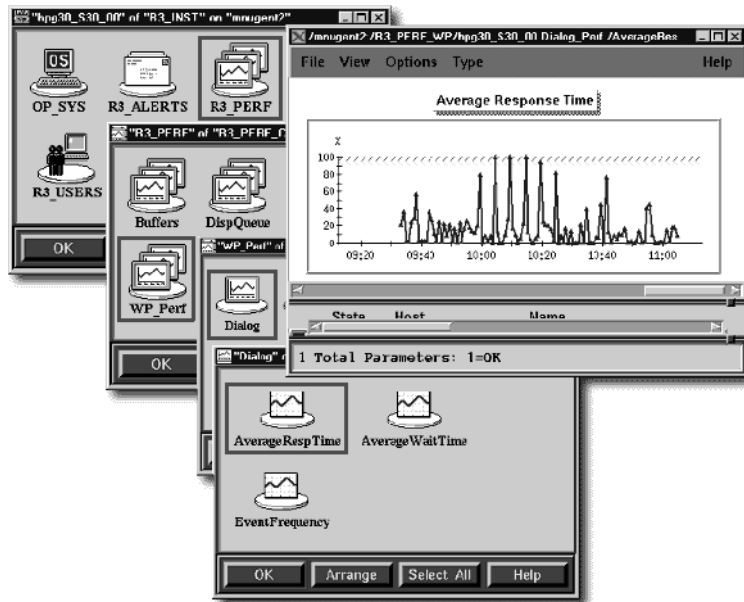


Figure 9-4 PATROL Knowledge Module for R/3

9.8 In-House Applications

When no off-the-shelf solution exists, such as in-house applications, PATROL can be extended via its SDK to add this feature. PATROL has commercial applications covered directly or via the PATROL Developer Network, but it cannot automatically do so for in-house applications. The solution is typically to build your own Knowledge Module for the in-house application.

The details of in-house application monitoring and Knowledge Module development are discussed fully in Chapter 15. The basic solution is to build a new Knowledge Module. In some cases, it is a simple matter of a log file monitoring solution for the application's textual log file. In other cases, using the existing RDBMS Knowledge Modules makes sense in monitoring an application using such a database.

9.9 Consoles

The data and events generated by Knowledge Modules require consoles to display the information. PATROL offers its own variety of consoles, and also a number of integration components for the consoles or help desks of other vendors. Some of the console or viewer options include:

- PATROL Console for Windows NT and Unix: Operator
- PATROL Console for Windows NT and Unix: Developer
- PATROL Event Manager Console for Windows NT and Unix
- PATROLWATCH for Web Browsers (Netscape Navigator or Microsoft IE)
- PATROVIEW—Integrated third-party consoles

This list shows that the PATROL user can choose from a variety of methods to view the status or perform administrative actions. UNIX platforms support the PATROL console in an X/Motif implementation. The PATROL Console for Windows NT (see Figure 9-5) is a native rewrite of the console to low-level Windows API calls.

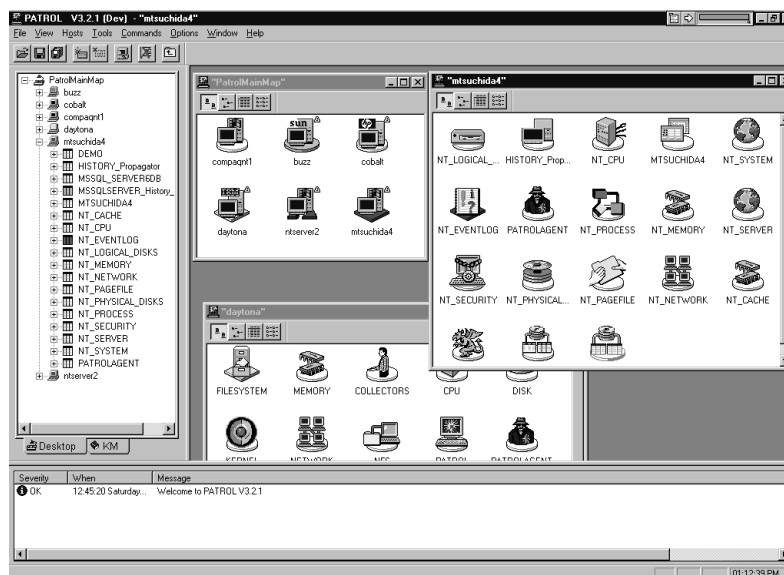


Figure 9-5 PATROL Console for Windows NT—operator view

These console choices offer full flexibility in configuration of different consoles to different users to view the same agents. The multi-console support of the agent permits multiple independent connections from all consoles and integration pieces. In fact, any number or combination of these viewer components can be used without restriction.

9.10 Web Browser Console

The familiarity of a web browser interface is available to view PATROL data and events. The official name is PATROLWATCH for Web Browsers (see Figure 9-6). This console component is an integration into either Netscape Navigator or Microsoft Internet Explorer.

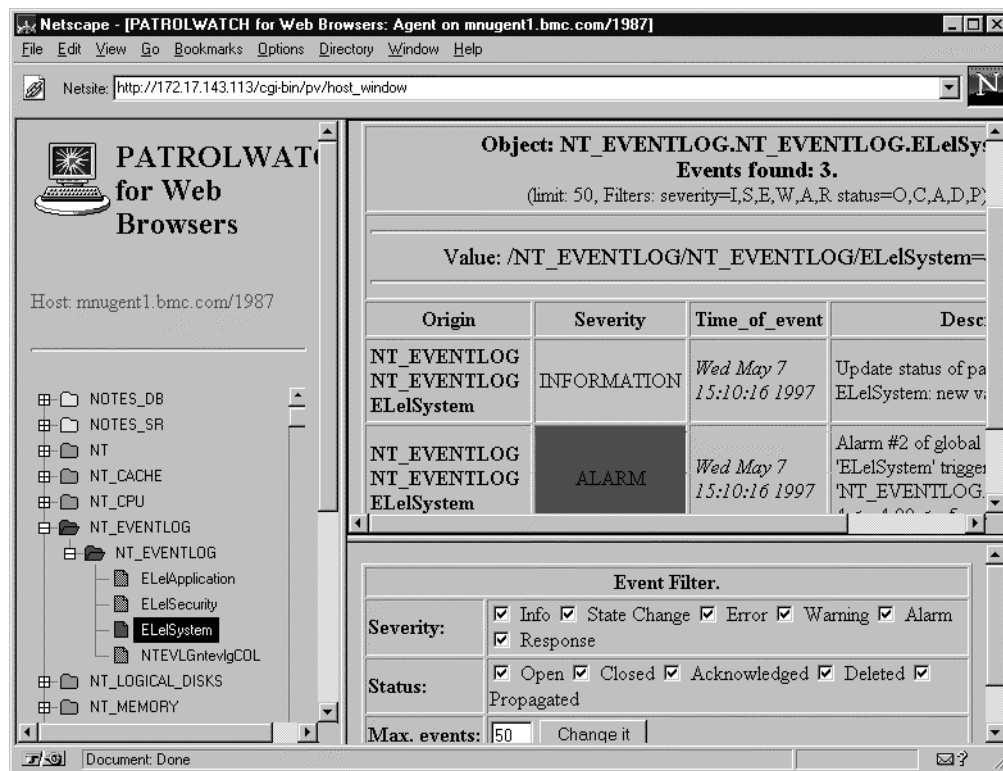


Figure 9-6 PATROLWATCH for Web Browsers

PATROLWATCH for Web Browsers offers several forms of monitoring functionality. It shows multiple agent hosts with indications of their status, in both an iconic drill-down form and a colored outline view. You also have color-coded views of the events coming from the agents. One of the key advantages of the Web Browser console is the familiarity of the interface which makes it popular with users.

9.11 PATROVIEW Integrated Consoles

Another major component of PATROL is its integrated components in the PATROVIEW suite. PATROVIEW integrates PATROL data from the agent's Knowledge Modules directly into other enterprise consoles and help desks. So PATROL can be used to complement existing network management consoles or help desks rather than attempting to replace them. PATROVIEW adds application management into monitoring of other areas such as the network or system layer while still retaining the existing investment. Some of the supported enterprise consoles or help desks include:

- CA-Unicenter TNG (see Figure 9-7)
- Tivoli TME 10 NetView
- Cabletron SPECTRUM
- HP OpenView IT/Operations
- Solstice Domain Manager
- Remedy AR System

The different capabilities of these consoles and help desks mean different levels of integration with all these products. PATROVIEW always offers event forwarding to the third-party console as a base function. In many cases, it also offers native iconic views within the network management console. This use of third-party API calls to populate the native views and data repositories with PATROL objects makes PATROVIEW the deepest type of integration available today into these consoles.

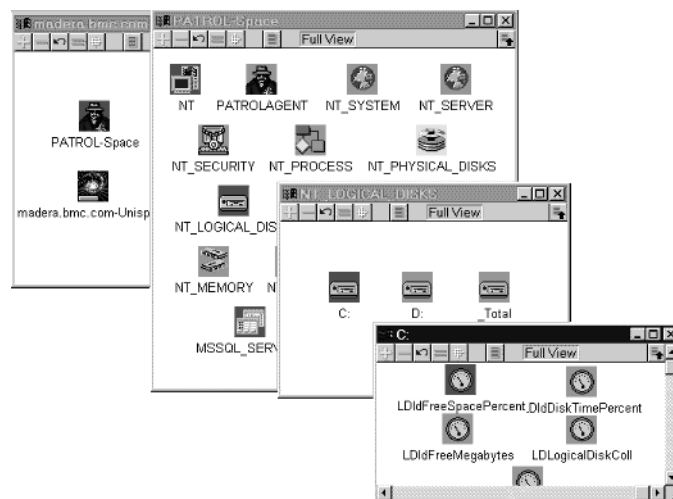


Figure 9-7 PATROVIEW for Unicenter TNG

This broad range of viewing options from PATROL consoles and PATROLVIEW integrations allows PATROL to be used standalone or as part of a broader systems management discipline using an existing investment. Both of these possibilities can be used together without interfering with one another, because the agent's multiconsole architecture applies equally well to PATROLVIEW consoles.

9.12 PATROL Flavors

I need to distinguish the various parts of the entire PATROL suite that extend its scope beyond pure application management into other related disciplines. Tools and products in a variety of IT management disciplines fall under the PATROL umbrella. The various categories of functionality include:

- Application management
- Data management
- Performance optimization
- Backup and recovery

Data management refers to a variety of tools, many starting with the name of "PATROL-DB," that are tools for users to perform various administrative tasks. These are tools that are manually operated as a usual tool and offer functionality such as schema browsing, database general administration, alter, change, and reorganization functionality. Performance optimization includes products that help users to tune systems, databases, or applications. Backup and recovery products for open systems include PATROL Recovery Manager which supports synchronous recovery of multiple heterogeneous databases through reliance on the PATROL Agent and Knowledge Modules. All these satellite products have their place as complementary tools to an application management strategy.

A common unofficial terminology is to refer to PATROL's application management products as "PATROL classic," although it is officially named PATROL Application Management. Personally, I prefer to talk about PATROL "original recipe" and the newer "hot and spicy" components. In any case, this book is about application management, and this suite of application management products is the predominant topic throughout these pages.

9.13 Internationalization

PATROL has a number of internationalized and localized products. Organizationally, both a European and an Asian porting group work with PATROL. European ports include operating systems such as Siemens SNI, BULL, Olivetti, and the various European flavors of Windows NT. The Asian porting involves the introduction of Multi-Byte Character Set (MBCS) support to deliver agents, consoles, and Knowledge Modules with Asian language support. Although it is an ongoing process, some of the current products available in Asian languages include Japanese Kanji (see Figure 9-9) and Korean versions of the agent, console, and major Knowledge

Modules such as Oracle. The extent of support can only increase for these European and Asian versions in the future.



Figure 9-8 Japanese internationalized PATROL Console

PATROL is moving toward a more complete localized architecture. For example, PATROL version 3.2 rolled out some of the major support for it in the form of message catalogs. These message catalogs are used by the agent and have PATROL Script Language APIs that allow Knowledge Modules to use catalogs. Future versions of PATROL are expected to further merge internationalization features into the base product architecture.

9.14 User Groups, Organizations, Events, and Publications

PATROL has grown into a large suite of products and a number of resources are now available to assist users. Taking advantage of these information sources can help you optimize your PATROL experience.

9.14.1 User groups

A number of user groups have formed for PATROL around the globe. Within the U.S., regional PATROL users groups are in New York City, Virginia, and Minnesota. Also, user groups exist in many other countries including Canada (currently both Toronto and Ottawa), Germany, France, England, and Australia. This list is constantly being updated, and you should check the BMC web site for whether a user group is in your area. Details of PATROL user group contacts in the U.S. and Canada can be found on the BMC Software web site, www.bmc.com/events/usergroups. User group information for other countries should be available from the local BMC office.

These PATROL user groups are coordinated by PATROL users and partners. Within the U.S., these user groups are required to be independent not-for-profit organizations in order to meet BMC requirements. Information on forming your own user group for the U.S. and Canada is also available at www.bmc.com/events/usergroups.

9.14.2 Organizations

In addition to user groups, several major organizations are available for individuals or partners interested in development, integration, or implementation related to PATROL. These organizations are:

- PATROL Developer Network (PDN)—development and integration
- Authorized Service Provider (ASP)—implementation
- Various BMC Software partner programs

The PATROL Developer Network is both for individuals and for partner organizations. For individuals, it provides tools and training resources related to Knowledge Module and PATROLVIEW-like console integration development. For partners, it provides the same assistance plus the possibility of involvement in the PATROLMARK program to get certification and possibly resell partner-developed PATROL integrated products. The PDN can be found at www.bmc.com/developers.

The ASP program is also both for individuals and for partner organizations. The ASP permits individuals and partners to become PATROL-certified as experts in the implementation and deployment of PATROL solutions. This provides training and certification opportunities and a toolkit of software tools and white papers related to implementation issues. The ASP is administered by a PATROL partner and can be found at www.certisource.com.

A more general BMC partner program also exists: information on the various partner programs available can be found on the BMC web site at www.bmc.com/partners.

9.14.3 Events

Many events are related to PATROL, from both user groups and BMC Software. Each user group typically has regular meetings according to its own decided schedule. BMC Software offers a variety of events as part of the PDN, the ASP, or the training from education services group, including various teleseminars.

BMC Software offers the “PATROL Technical Symposium” every year in Houston. This symposium is near the head office and attracts users, partners, and PATROL personnel as attendees, exhibitors, and presenters. This is a popular event that has been growing each year. Information and registration forms for the symposium are on the BMC web site at www.bmc.com/events/symposium.html.

Outside of the U.S., events are also hosted by local BMC offices to cover various PATROL topics. Information on these events in Europe, Asia/Pacific, or Latin America can be

found on the BMC web site in its list of events (www.bmc.com/events), or directly from the local BMC office.

9.14.4 Publications

A number of user groups put out a regular newsletter as part of their user group activities. The Canadian PATROL user group even publishes its newsletter on the web, and others are sure to follow.

BMC Software produces both the “Tech News” and “WorldView” magazines. Tech News can be found on the BMC web site via www.bmc.com/technews. Both of these magazines offer PATROL articles of interest to users and partners alike. A PATROL Developer Network technical newsletter is also available.

9.15 Recent PATROL Developments

The PATROL industry is continually producing new products. Unfortunately, it’s now too late to incorporate detailed discussions about the latest developments into this book. Some of the most interesting areas that will have to await the second edition include:

- **BEST/1 and PATROL**—The combined use of PATROL with BEST/1 is a promising area and integration products are starting to appear. BEST/1 offers advanced performance data collection, predictive models, and capacity planning features.
- **COMMAND/POST and PATROL**—The recent special relationship between PATROL and COMMAND/POST is also another interesting area.
- **PATROLMARK Solutions Catalog**—There are literally dozens of new Knowledge Modules and Integration Modules available from PDN members. This list includes products to monitor firewalls, security monitoring, log file monitoring, and much more.
- **PATROL KM for Networked Applications**—This product uses network packet analysis to monitor network traffic and packet latency response time on a per-application basis. For example, it can determine how much of the network pipe is used by FTP versus Lotus Notes. This product was originally code-named PATROLNET.
- **PATROL 3.3**—The upcoming core 3.3 release offers interesting features in installation, configuration, and deployment. There is also planned support for Kerberos, COM, Visual Basic, and various other enhancements.
- **PATROL Adapter for WBEM**—PATROL 3.2 now supports the Web-Based Enterprise Management (WBEM) standard. This integration is a two-way mapping between the CIMOM and PATROL, with support for CIM to KM mapping.
- **E-commerce solutions**—These solutions have been recently announced and involve the extension of the existing PATROL Internet products to e-commerce applications. This will be an interesting area to watch.

9.16 About BMC Software

BMC Software was, until a few years ago, predominantly a mainframe software company. Founded in 1981, BMC Software formed its business upon compression technology, and quickly grew by providing mainframe database administration tools for DB2, IMS, and CICS. Currently, more than 100 mainframe database tools are provided for areas such as backup and recovery, integrity, reorganization, and much more. BMC Software continues to expand its large suite of mainframe database tools.

However, in 1994, with Max Watson at the helm as CEO, BMC Software embarked upon a concerted effort to extend its reach into the open systems arena. It achieved this growth by the acquisition of new technologies and the extension of its existing mainframe technologies into the open systems spectrum. PATROL was a key acquisition for BMC Software and launched BMC Software into the applications management market. PATROL is now the flagship product of BMC Software in its open systems market.

BMC Software is a Forbes 500 company with revenues in excess of \$750 million U.S. per annum. Broadening the company beyond its mainframe heritage, CEO Max Watson has grown BMC Software into one of the largest independent software companies in the world.

9.17 History of PATROL

PATROL is at one stroke both an astounding entrepreneurial success story and an example of innovative corporate strategy through acquisition. Within five years, PATROL has moved from being a newly founded startup to being the flagship product of a major corporation.

PATROL was founded by Martin Picard as an independent Australian startup in the early 1990s with a focus on application management. One story tells of how its conception was based upon a fishing rod with an attached bell on Martin's yacht as he sailed from the U.S. to Sydney. The bell served to sound the alert when a fish was on the line. PATROL was certainly attractive bait for BMC Software, which purchased PATROL for more than \$33 million U.S. dollars in early 1994. BMC Software was then a large, although largely unheralded, mainframe database tools company, based in Houston, Texas, and steered by CEO Max Watson. It was seeking a path into the open systems market, and PATROL was an attractive target.

The first version of PATROL produced by the Australian startup showed the foundation of what were to become the key points of its architecture: agent, console, and Knowledge Module. PATROL version 1.0 offered a graphical console and an agent both fully based on UNIX. The Knowledge Module existed to encapsulate the individual properties of the various databases that were monitored. However, the representation was limited to declarative specifications, and to dynamic operations from shell scripts and SQL commands.

PATROL version 2.0 added numerous features to the underlying architecture. Notable achievements included the VMS agent and the powerful PSL script language for fully dynamic Knowledge Modules. The breadth of coverage of Knowledge Modules was expanded on the basis of the new powerful script language. However, version 2.0 was limited in terms of its scalability because of its console-centric architecture. The agent was not autonomous, but was

dependent on the presence of a console to control it. The fact that each agent pumped every single data value for a collected metric back to the console could create a network and CPU bottleneck at the console. This effectively limited the scalability of PATROL to fewer than a hundred agents.

Version 3.0 punctuated the premier release of PATROL with a much more scalable architecture. The deep pockets of BMC Software allowed a much larger effort in development. This release saw the introduction of the autonomous agent architecture, which is the foundation for the PATROL architecture to this day. This architecture was later rewarded for its originality with a U.S. Patent. With the removal of the console-centric dependency, the scalability of PATROL was unbounded in terms of the number of agents per console. Other major advances included the addition of various PATROLVIEW integration modules, true SNMP support by the PATROL agent, major PSL enhancements, major event management enhancements, a PATROL agent open API, and true multiconsole support. The platform coverage of PATROL was expanded to Windows NT and OS/2, along with numerous Knowledge Modules for various databases and applications.

The PATROL architecture remained relatively similar to the original 3.0 architecture in the 3.1 and 3.2 releases of PATROL. Version 3.1 added a number of significant presentation enhancements such as multi-line graphs, annotated data points, and a command-line interface. Version 3.2 offered an improved Windows NT console with a native look and feel, and a suite of MVS Knowledge Modules founded on an MVS agent. BMC Software had succeeded in merging its open systems flagship product with its history of mainframe competency.

PATROL Architecture Overview

One of the strongest features of PATROL is its architecture. The autonomous agent is a key innovative component of this architecture. Other features such as PATROL Script Language (PSL) and the concept of PATROL Knowledge Modules (KMs) serve to make PATROL open and extensible.

The autonomous agent is the center of the PATROL architecture. The various console viewing components and integration components are like satellites around each agent. The key integration technologies are those that interface with the agent to gather its data or provide it with more data or events.

However, many other architectural points need to be examined. A variety of console solutions exist, including UNIX, Windows NT, web browser, and integrated third-party consoles. The PATROL KM is also a key architectural issue in its own right. Its encapsulation of both static and dynamic knowledge about an application is the key to the extensibility of PATROL to all layers of the application technology stack.

10.1 Autonomous Agents

The key concept of the PATROL architecture is the autonomous agent, which is the center of the architecture. A console connection is not a requirement and the agent can continue monitoring, notification, and correction actions in a lights-off environment. In fact, many consoles or integrated components can connect to a single agent without interfering with each other. This capability allows multiple views of the same objects and permits backup consoles as well as consoles that come and go at the whims of the end users. None of this affects the autonomous agent's continuous processing to perform monitoring of the application.

This autonomous agent architecture was introduced in the version 3.0 release of PATROL. The 3.0 release brought with it many scalability-related enhancements. The introduction of autonomy is mainly one of configuration and internal design. Specifically, this autonomy was achieved through the offering of the three key features:

- Local control
- Local execution
- Local storage

Local control was achieved by making the agent perform its own decisions. The agent does not rely on the console to tell it what to do. Certainly, the console can affect the behavior of the agent and change its configuration, but the agent does not require this direction. Instead, the agent will begin its own execution sequence according to its loaded Knowledge Modules, right from the start of the agent's execution. It does not wait for the console to connect and tell it what it needs to do.

Local execution is achieved via the internal execution and scheduling engines within the agent. The PSL interpreter runs fully within the agent's internal architecture. It is self-contained. The scheduler is also running according to the agent's own configuration and does not require any external interaction. Right from the start of the agent's execution, the sequence of interpretation and scheduling of commands begins.

Local storage is critical to autonomy for two reasons: knowledge and data. The local storage of the Knowledge Modules, and particularly the PSL code, permits the local execution to occur. Without it, the agent would need to wait for a console to connect and give it the knowledge. The agent automatically loads the required Knowledge Modules from its local knowledge repository and begins execution of the knowledge. It continues to run discovery scripts, parameters, and can run corrective recovery actions, all without needing a central monitoring console.

The second point is that the local storage of parameter data values and events is required for autonomy. Although the agent could still run autonomously, without a local storage area, it would have to throw away all data and events. With the local storage database for parameter history and events, the autonomous agent can store its data and events without worrying about whether a console is currently connected. Whenever a console connection does occur, the agent can serve up the current and historical data as requested by the console. The console can get the information, but the console is not required there to help store it.

A particular example of local intelligence is the PATROL Agent's handling of alarms. The agent itself performs the decisions about whether to raise an alarm by comparing a parameter value to its threshold. This avoids the need to send values to the console, which would cause a scalability problem.

This local decision making is not limited to the raising of the alarm. The agent can respond to the alarm via a locally launched recovery action. This avoids the inefficient round-trip decision-making message that would occur in a console-centric architecture. The recovery action can execute a "write" action on the application, if configured to permit this. Or it can perform

other operations not directly changing the application, such as storing information for postmortem reporting; or generating a notification command via email, pager, or SNMP trap; or integration with another framework.

10.2 Knowledge Module Architectures

The Knowledge Module is the key to allowing PATROL to monitor multiple databases and applications. It is the basic architectural idea that leads to the fundamental extensibility of PATROL. Some of the key architectural aspects of a Knowledge Module include:

- The abstraction layer exists between the application and the agent's data model.
- Declarative and procedural components together comprise a Knowledge Module.
- The agent's resources are shared by each Knowledge Module for efficiency.

The Knowledge Modules represent an abstraction layer that translates the details of the database or application into the simple data model used by the PATROL Agent. By encapsulating any application-specific logic into rules within the Knowledge Module, the next level of services can be independent of the underlying database or application.

This abstraction layer is partially achieved via the portability of the PSL language. This reduces the amount of platform-dependent logic that is required for each application. However, this is portability for the same application across multiple platforms, but not abstraction over different applications or databases from different vendors. This cross-vendor abstraction layer is achieved via the back-end implementation of the Knowledge Module. This is the lower-level layer whereby the PSL or C code that comprises a Knowledge Module processes data and objects that are specific to the monitored application. This is the lowest level of the abstraction, where all the application-specific logic is embodied. This layer gathers the data that is plugged into the agent data model that represents all the objects discovered by the Knowledge Module.

This data model is a relatively simple model that logically represents all the database or application components. The basic components of the model are the "instances" that represent the actual software entities (e.g., a server component of a database), and "parameters" that represent the measured data for useful metrics about the component (e.g., response time or throughput of a component). This sort of information is plugged into the data model.

A clear separation exists in the underlying data model for Knowledge Modules into declarative or static knowledge and procedural or dynamic knowledge. The surrounding structure of a Knowledge Module represents the declarative logic, such as what type each data value is (e.g., percent, ratio, absolute value), or alarm threshold settings and scheduling components. All of these are declarative or static values, although some of these apparently static values can be overridden for more dynamic behavior using the procedural features.

The procedural component is achieved via PATROL Script Language (PSL)—an internal agent script language that is executed internally by the agent. As a fully functional programming language, PSL offers the full suite of procedural logic capabilities. Issues that are typically

solved in procedural ways are the dynamic aspects of parameters, discovery, and menu commands. These involve how to gather the data from an application (i.e., what commands to run and how to process the returned data), how to discover the application components (e.g., what processes to look for), and what to do for administrative menu commands. Recovery actions are another clear dynamic requirement because they perform actions against the application itself.

Both declarative and procedural services are provided by the agent. Each Knowledge Module's requirements are shared by the agent such that multiple Knowledge Modules do not require multiple agents. This approach permits efficiency and also simplifies the deployment of additional Knowledge Modules into an existing monitoring scheme. Procedural aspects of the Knowledge Modules are shared within the agent via scheduling algorithms that timeslice between each PSL script. The declarative aspects are shared by the agent through processing of declarative constraints whenever new data is generated from the procedural components or from external accesses to the agent.

10.2.1 Packaged Event Generation

The purpose of an application management tool is to automatically detect problems. Therefore, to have prepackaged event generation expertise is desirable. In PATROL, this is what Knowledge Modules represent. They contain information that tells the agent what is considered a problem and requires an alert event.

The first role of the Knowledge Modules and their packaged event detection is to know what collected metrics represent. The meaning of a value affects its event characteristics. For example, an operating system Knowledge Module, such as the PATROL KM for Unix, distinguishes different types of metrics. The fact that CPU utilization measurements are more volatile than filesystem capacity measurements is encoded as part of the knowledge. So the Knowledge Module smooths the alerts on CPU utilization unless they occur for too long a period of time, but can immediately issue an alert if filesystem utilization jumps to a percentage that is too high. This type of simple encapsulated event behavior is an issue for all Knowledge Modules. The PATROL KM for Oracle handles cache hit ratio alerts differently from tablespace extent alerts.

The underlying Knowledge Module architecture is powerful enough to express most alert types. Some of the simplest forms are specifiable as declarative types on parameter metrics, such as simple thresholds and peak smoothing. More advanced analysis such as rate of change, statistical analysis, and historical predictions can be coded in PSL through the use of the PSL history, mathematical, and arithmetic functions.

10.2.2 Correlation

Correlation is another aspect that can be encoded into a packaged form. For example, the PATROL KM for Oracle can interrelate events from the database and the filesystem objects. This capability is localized node-level correlation. Another example is the PATROL KM for Sybase that takes its own metrics and outputs a higher level event according to the problem it detects. This is prepackaged problem diagnosis via node-level event correlation.

Application-level correlation is another aspect that can be prepackaged. This can be single or multi-node correlation, depending on the application architecture. This means that SAP R/3 problems are sorted out into their components' related alerts. For example, an R/3 alert can be correlated against database or filesystem problems, giving an overall view of the application itself. This is multi-node correlation because the R/3 application is physically dispersed across multiple machines, and the Knowledge Module brings this multi-agent data together.

Not all levels of correlation can be encoded into prepackaged components. Although simple correlation can help diagnose Oracle or R/3 problems, no way exists to prepackage the site-specific interrelationships. Although everything that is under control of the R/3 architecture can potentially be correlated to some level at the application level, not all components are this way. The higher-level business logic is not something that can be prepackaged. Configuration of event generation and the use of console-type event correlation or generalized root cause analysis is therefore something that complements the expertise that can be prepackaged.

10.2.3 Recovery Actions

The proactive nature of recovery actions arises from a combination of the Knowledge Module design and the autonomous agent architecture. The recovery action is an agent-executed corrective action that is launched once a problem is detected, in order to attempt to correct it. Recovery actions are proactive in nature because they attempt to automatically resolve the problem and do not require user intervention. Indeed, the console-independent autonomous agent architecture ensures that no console is needed at all.

Recovery actions can also have escalated multiple actions. A sequence of recovery actions can be defined to attempt alternative solutions to an alert. Recovery actions are also automatically stood down if the problem is found to be resolved. Recovery actions can take a wide range of actions, such as reports or deep analysis for annotation with postmortem data. Another use is clearly to perform notification via pager, email, SNMP trap, or third-party integrated consoles. However, although this notification is common, this is only one possible alternative method of notification, whereas the proactive nature of the recovery actions for true corrective actions is more unique.

10.2.4 Automatic Discovery

The discovery script is another component of the Knowledge Module architecture. This discovery script is executed repeatedly, and the Knowledge Module uses it to examine the environment for the typical signs of an application's presence, such as files, processes, or commands with specific output. The agent then creates the objects internally for the newly discovered applications and begins monitoring their parameter metrics. The autonomous agent architecture ensures that no user or console input is required for any of this process.

Another side to auto-discovery is auto-destruction. The reverse situation occurs when a software asset is moved from one machine to another. Not only will it be automatically discovered on the new system, but also it will be automatically removed from the monitored set on the

older system. So the monitoring of Knowledge Modules adjusts automatically to changes in software configuration.

10.2.5 Non-Intrusive Application Management

The strategy taken by PATROL in the development of its database and application management is a pragmatic one. Rather than attempting to define an idealized view of what application management should be, PATROL makes the most of the current situation. The clearest example of this strategy is the non-intrusive Knowledge Module architectures.

The non-intrusive nature of the Knowledge Module is not an architectural issue. Indeed, the data model and procedural elements are capable of supporting an intrusive or non-intrusive approach just as successfully, without technical limitation. Rather, the current state of limited instrumentation in commercial and in-house applications effectively forces a non-intrusive approach. In order to offer any type of application management, the Knowledge Module must make the best of all the available sources of information about the application.

“Non-intrusive” means that PATROL sits on the outside of the database or application, rather than waiting for some internal support from within the application. PATROL does not wait for vendors to comply with a standard for application management. Doing so would only serve to lengthen the cycle by forcing administrators to wait until the next revision of the database or application is available. This strategy also clearly strands users who do not upgrade to that new application version and offers no application management relief. Instead, PATROL learns all the details of the existing products and figures out how to best monitor and administer these products using the available interfaces and methods. The more advanced databases and applications tend to already have strong, although non-standard, interfaces into their products, and PATROL leverages these as well as possible. PATROL also works with vendors to further strengthen these interfaces for future application releases. In this way, a good solution is available for the present application release, and further improvements appear in the future releases.

As more advanced forms of instrumentation start to take place within the market for commercial applications and as part of in-house development, Knowledge Modules will move to take advantage of these. No major change in architecture is needed to take advantage of these instrumentation strategies. The data model is fully powerful enough to express the monitoring requirements of an instrumented application, and the procedural components are powerful enough to gather data in the manner required by that instrumentation.

The advantages of the PATROL pragmatic approach to application management can be seen in the huge range of Knowledge Modules that are available. Application management can be made available immediately. No delay occurs while waiting for vendor compliance, which is unlikely to occur anyway, given the current variety of competing instrumentation standards. PATROL’s strategy has provided broad coverage of the heterogeneous software applications throughout an enterprise, and provided it now.

10.3 PATROL Script Language (PSL)

PATROL Script Language (PSL) is a programming language that executes on PATROL Agents. Syntactically, it is similar to procedural languages such as C, Pascal, and Perl. PSL is a procedural language with all the standard constructs such as variables, conditional statements, functions, and loops.

The agent executes PSL with its built-in interpreter. This provides an abstraction layer above the operating system that makes PSL portable to all agent platforms. PSL supports both interpreted text versions and compiled binary versions. It also provides support for user-defined libraries of functions to be compiled into a library format to support sharing of code.

The PATROL environment has various tools that work on PSL. The PATROL Developer Console is the main owner of some of these tools, such as the editor and debugger. A basic text editor for PSL is part of the console. Also, a graphical debugger exists for operations such as stepping through code. PSL also supports both compile-time and runtime warnings about common programming errors such as variables used-before-set. Also, PSL profiling and optimization capabilities are within the agent for performance measurement and tuning.

The basic features of PSL are complemented with a wide range of built-in functions. These extend the basic capabilities of the language in a number of simple ways and a number of complex agent-dependent ways. The simple enhancements include functions in areas such as mathematical and string manipulation. The complex agent-specific built-in functions include:

- SNMP—trap sending, trap receiving, SNMP get/set/getnext requests
- Command launching—synchronous and asynchronous execution
- File operations—access, creation, read, write, etc
- Locking and concurrency
- Global channels for efficient channel access to applications
- Inter-agent remote access operations
- Event generation and processing
- Console-related GUI operations

Through all of these capabilities, you can make PSL do pretty much anything. In combination with the agents, PSL becomes a platform for a higher level of application management functionality. This has delivered the huge range of existing Knowledge Modules and is open to deliver the same for in-house or other applications.

10.4 PATROL Console

PATROL provides a standard console component with all the features for application management from a user perspective. It allows users to view objects and events, and also to respond to them with a variety of tools, such as menu commands, that are built into each Knowledge Module. Also, development features in the PATROL Developer Console permit the development of a new Knowledge Module or the customization of an existing one.

The PATROL Operator Console is the usual mode for monitoring with PATROL. The console is connected to all the agents and is automatically notified by the agents about any alarms. The console has a great many different features based upon this architecture. Some of the most important features include:

- Charting—graphs and multiline charts that are highly customizable. Real-time and historical views of data are available. Annotated data points display historical extra information such as the currently executing SQL script at the time of a database alarm.
- Query tool—use built-in or custom queries of the objects from multiple agents. Allows complex analysis of the agent's created objects.
- Event management—view events and perform actions such as acknowledgment on events.
- Menu commands—issue administrative actions defined in Knowledge Modules. Full GUI popups can be defined for interactive actions.

These features of the console are useful to operators and administrators alike. Another side of the console is the development features in the PATROL Developer Console. These allow the development of a Knowledge Module for monitoring in-house applications. The PATROL Developer Console can be used in conjunction with the PATROL Knowledge Module Toolkit (or SDK) as a Knowledge Module development environment.

10.4.1 Multi-Console Architecture

The agent-centric architecture of PATROL affects how the agent connects to multiple consoles. The relationship is one-to-many, with one agent on a given host speaking to multiple consoles. “True multi-console” prevails, with all consoles monitoring a given host, seeing exactly the same events, and getting the same values for parameters.

In PATROL, only a single agent operating system process exists, which handles all connection requests and all messages from each console. Hence this single agent implements true multi-console functionality, with each console receiving the same status and/or values for a particular agent object that both consoles are monitoring.

No scalability implications result from the use of multiple consoles. The PATROL-level protocol minimizes the number of events or data points sent to each console. Although adding an extra console yields an additional set of messages for each data point, this is not a great burden for the agent. The number of messages improves only linearly with the number of consoles. Although this still might seem problematic for many console users, remember that the bottleneck for message production, were it to occur, would not occur at the agent side. After all, the agent need send the messages for only one agent: itself. The console-side bottleneck is not added to by adding a new console. Each console has no performance impact on any other.

An important aspect of multi-console is the per-console viewing of application classes. Each console can specify which application classes it wishes to monitor, and these need not be the same as those of another console talking to the same agent. The agent simply monitors all of

the application classes requested by each console. For example, one console could be monitoring the PATROL KM for Unix on an agent, whereas another console is monitoring Oracle; the agent will be running both UNIX and Oracle applications, but sending UNIX messages to one console and Oracle messages to another.

These comments apply to all consoles and console-like components. No interference takes place from one console component to another. This allows any combination of the various console-like components: PATROL Developer Console, PATROL Operator Console, PATROL-VIEW, PATROLWATCH, CLI, and any other API-based application. The PATROL Agent is the central server to each of these clients.

A historically interesting note is that PATROL version 2.0 did not have true multi-console support. Instead, it spawned a separate agent process to handle each console. The major architectural enhancements in version 3.0 removed this great inefficiency.

10.4.2 Console Scalability

The console has sufficient performance to handle a large number of agents into the hundreds and thousands. The protocol between the agents and console ensures only minimal network traffic incoming to the agent by informing the console about only objects with exceptions or those that the console requests (i.e., visible objects to display to the user). This avoids a network processing bottleneck at the console and also makes the number of objects that the console needs to manage low per agent.

No technical limitations exist to the numbers of agents that can be managed through a single console. However, some clear organizational limitations apply in the deployment of a console in such large environments. Desktop management is clearly an organizational issue for a large number of agent connections. The text desktop file format permits a number of organizational methods to automatically maintain this desktop. These organizational or procedural issues are discussed further in Chapter 14.

For now, what is interesting is to understand the technical features of the PATROL architecture that allow the console to technically handle so many agents, with apparently so many objects. These technical improvements occurred in the 3.0 release of PATROL and involved improvements to both the underlying console-agent protocol and to a lesser extent the console's memory usage. The most critical improvement was reducing the overall number of messages, events, and data objects that the console was receiving in real-time. The console-agent high-level application protocol was changed for PATROL 3.0 in many major ways to reduce the message load on the console from all its agents. The only times that the agent sends the console a status are:

- Alerts from warnings or alarms
- Visible objects on the console from drill-down windows and open graphs
- Low-level protocol session heartbeat

This model scales because the agents send information to the console only on an as-needed basis rather than pumping everything back. Only alerts are sent in real-time, so that OK statuses for any objects that are not currently visible to the console user are no longer necessary. This model hugely reduces the number of objects that the console must store internally to only those that are visible or those that are in ALARM or WARNING state.

Parameter values are not sent in real-time by default. That is not necessary, since alert threshold analysis occurs at the agent. Instead, parameter values are sent to the console only when the console user requests them by drilling down into a given parameter to view its value. This feature is probably the single most important feature in reducing the volume of incoming messages to a console when many agents exist.

This protocol leads to an important change in the discovery of information from agents. In PATROL version 2.0, all objects had their statuses and parameter values stored at the console side, and drilling down into an object only required the console to access its locally stored knowledge. In PATROL version 3.x, this setup is no longer generally true (although exceptions exist), and drilling down into an object on the console requires the console to request a status or value from the agent. Hence a change in the handling occurs when the console user drills down into a computer, application instance, or parameter.

Drilling down into the icons displays further details about the sub-objects. However, these are not always stored on the console machine, but instead are actually sending a request to the agent. This fact explains the waiting cursor that sometimes appears when drilling into an object. The second time an icon is drilled into, it may already be in the console's cache and may not require further agent information. After time, this cache is reduced, because otherwise it would build up to contain too many objects and once again have scalability problems.

Console memory consumption is low, by removing the need for the console to store information about every object on every agent. Instead, the console attempts to store only those objects that are currently visible to the user. These improvements result from the console-agent protocol.

In addition, those objects that must be stored are also manipulated through a disk-based database of objects, allowing swapping of objects from memory to disk when required. A thousand agents cannot necessarily be managed on a machine with minimal memory requirements, but the requirements are realistically achievable. The console performance bottleneck of the old version 2.0 PATROL release is gone.

An important criteria for console scalability is for the console to be able to manage sufficient console connections. Since the console must have an open connection with each agent, with the use of TCP, a kernel-imposed limit applies to the number of file descriptors. For this reason, PATROL supports UDP connections that have no such kernel limit. An additional reliability layer is imposed above the UDP layer because it is otherwise not guaranteed to be a reliable transport layer.

10.5 Agent Scalability

The agent is a scalable software component. This scalability appears according to a number of metrics that measure the load on the agent, including:

- Number of consoles connected to the agent
- Number of other clients connected (i.e., PATROVIEW, PATROLWATCH, CLI, etc.)
- Number of objects monitored by the various Knowledge Modules

The agent is scalable with respect to its console and client connections. Only minimal extra impact originates from each extra console or other client. The true multi-console nature of the agent ensures that the agent does not impose extra load on the system per console. No major difference exists between how the agent handles its variety of client connections, including consoles, PATROVIEW, PATROLWATCH, configuration tools, CLI, and the API. The underlying basis is the API itself, and this means the only difference is in the slightly differing registrations from the different client types. The number of concurrent client connections has been tested well in excess of 100 systems. The type of connection makes no difference with regard to UDP versus TCP, or with regard to the responsiveness of the network connection between agent and console. A slower network connection merely imposes slower response times on the console side to drill-down or other requests, but does not affect agent performance in the slightest.

The other side of scalability is the inward performance characteristics of the agent with respect to its monitored system. The first major point is that the agent is only ever a single agent process, no matter how many Knowledge Modules are running. The agent handles every PSL script for each Knowledge Module according to its scheduling policy, and does not ever launch a second agent.

The second major point is the self-tuning capabilities of the agent, meaning that the agent reduces its own operations if it detects a high load situation. So the agent's load on the system should always be self-moderated regardless of how many objects each Knowledge Module is monitoring.

The third major point is that the agent is efficient in terms of system load, even when monitoring a large number of application objects. No base load exists, since the agent itself takes no resources without a Knowledge Module. Not much overhead occurs in servicing console connections. Other overhead appears from the PSL language and external child processes. The load on the agent with a basic operating system Knowledge Module and database Knowledge Module is in the low single-digit CPU percentages.

The fourth point is that the load on the database or application itself is low. The Knowledge Modules use architectural features such as global channels and vendor API access methods to ensure an efficient connection to the databases or applications. This point is important since you do not want the monitor to adversely affect the monitored object. Although even the Heisenberg Uncertainty Principle from theoretical physics, which expresses the fact that measuring

something also changes that thing, proves that some load is inevitable, the Knowledge Modules serve to minimize this effect.

10.6 Event Management

The PATROL Event Manager (PEM) component is an important component of the PATROL architecture. In fact, events are central to the architecture and affect almost every architectural component. The first architectural issue is the make-up of the events themselves.

Events are the lifeblood of the monitoring performed by PATROL and its Knowledge Modules. The agent is the main creator of events. Knowledge Modules scan their monitored application for problems and raise events called alerts. These alerts are just one type of event possible in PATROL. State change events are a little more general than these alerting events (i.e., warnings or alarms) and occur whenever a monitored object changes its status. Also, informational events take place for a variety of different occurrences.

Events have a variety of attributes. They belong to a class of event that is either built-in or defined by a Knowledge Module. They contain context information such as the originating node and object name, along with any other information pertinent to the event such as the parameter value and threshold that was violated, in the case of an alarm event. Events also contain an event description as well as expert advice sections containing textual material to help users deal with events.

The events are processed throughout the architecture by the different components. Each component has a different role in the event life cycle. However, much similarity occurs in event handling throughout. Some of the components that deal with events are:

- Agents
- Consoles—UNIX, Windows NT, PEM Console
- Integrated third-party console views—PATROLVIEW
- Web browser views—PATROLWATCH
- PSL
- CLI
- API

The agent typically creates the events according to some internal agent action or through creation via a Knowledge Module alarm or a PSL operation. The events are stored locally in the agent's event repository on a local agent disk. This location is necessary to offer the autonomous agent architecture without losing events when no consoles are connected. In fact, the agent always maintains a copy of the events, even if it does forward an event to a console or another component.

The PATROL Console for Unix or Windows NT also has various event processing features. The main feature is a graphical display of event information through GUI windows. This allows the viewing of events either historically or in real-time. Events can be acted upon such as

by closing, acknowledgment, annotation of the event diary, or deletion. These features are also available in the standalone PEM Console called “emcons.”

Events are also critical to the various integration components for third-party consoles and help desks provided by PATROVIEW. Event processing is the primary functionality of these integrations. Although other features of PATROVIEW include object viewing and console launching, events are still the most critical aspect. After all, the main reason to integrate these other consoles or help desks is to ensure that PATROL alerts get sent to the enterprise console or the central help desk personnel. PATROLWATCH also provides event features in the web browser views that it provides.

The underlying event access mechanism is the PATROL API. This allows a wide range of event-related actions with a local or remote agent. Support is available for both synchronous and asynchronous queries about events. The synchronous queries are useful for user-initiated queries from consoles. The asynchronous queries offer a registration mechanism that allows the API to wake up the client when a new event has been created by the agent. This permits easy real-time event viewing in consoles and PATROVIEW without an inefficient polling architecture.

The PATROL CLI is based upon the API and offers similar event management features. This permits home users to easily analyze events from a text window in the interactive mode. The batch mode also permits scripts that perform event-related activities.

PSL has various event management built-in functions. Clearly, events can be created at the agent via PSL. However, also available are features to query the event repository, perform actions such as acknowledgment of events, and much more. PSL contains support for both local and remote inter-agent event actions.

An important feature made possible by true multi-console functionality is multi-user event management in the PATROL Event Manager. Multiple consoles share the same events, and actions applied to a given event are seen by any other consoles monitoring the same agent. For example, if two users are examining the event log for the same host and one console acknowledges the event, this fact will be shown in the second console’s event window.

The agent is the coordinator of any multi-console event operations, and the event repository is stored locally to aid in this. The local repository also allows the agent to store events when no consoles are connected in an autonomous mode.

This feature allows the use of the PATROL Console’s event management components as a form of help desk for PATROL events. Although not as fully featured as the various third-party help desk or service desk consoles, the PATROL console can be used in this way.

Also, a separate standalone event management console is available. For example, the PATROLWATCH for Windows product allows this functionality. This separate console is also available as an integrated component of the main PATROL console. In either case, this event management console can be used to view the list of open events and perform actions such as acknowledgment or escalation.

10.7 Network Transport Protocol

PATROL allows both TCP and UDP as the underlying transport protocol. Since UDP is inherently unreliable (whereas TCP/IP has reliability features), an extra layer of reliability has been implemented inside PATROL's transport level on top of the UDP layer. This involves TCP/IP-style reliability, including features such as acknowledgments, retries, heartbeats, timeouts, sliding windows, etc. This is a layer very closely resembling TCP/IP, but at a user level. Hence the UDP-based communication between the console and agent has three conceptual levels:

- UDP layer (system level)
- Reliability session layer (acknowledgments, retries, etc.)
- Console-agent high-level application protocol (parameter values, object statuses, etc.)

The implementation of the TCP communications layer is similar, but without the reliability level. In addition, UDP requires its own heartbeat, because a downed connection will not automatically be transmitted down to the console client.

The console-agent protocol is critical to the PATROL architecture. The handling of this PATROL-level protocol is crucial to the achievement of a scalable console connection.

Although the connectionless UDP protocol seems critical to achieving autonomy, the autonomous nature of the agent actually has nothing to do with the low-level network protocol. The autonomous nature of the agent arises due to enhancements at the PATROL-level protocol and is not affected by the low-level protocol. These high-level enhancements included local autonomy of execution, local repository of knowledge, and local data storage. The autonomous nature of the PATROL agent occurs for both UDP and TCP without limitation.

10.8 Agent Internal Command Scheduling

The central data structure for the PATROL Agent's scheduling of jobs is the main run-queue. This is an internal queue data structure that has all the executable commands, including discovery and pre-discovery procedures, parameters, process cache cycle, menu commands, etc. The main run-queue has jobs that are scheduled to run soon, not jobs that are currently running. The run-queue is ordered by the time that a command should run next. This time is altered only by actions such as "Update Parameter"; otherwise, the command will run at the time scheduled.

After a command has executed, it is rescheduled according to its scheduling policy. For a simple once-only command such as a menu command, this rescheduling does not occur. Periodic processes such as parameters, discovery procedures, and the process cache are rescheduled at their next allotted time. Parameter scheduling is the most complicated issue because of the various scheduling options such as immediately, day-of-week, day-of-month, etc. Other periodic commands such as discovery or process cache have a simple period (40 seconds for discovery and 300 seconds for process cache), but these are configurable for the application via the "Custom Discovery Cycle" entry in the application dialog, and for the process cache by setting the PSL variable `/processCacheRefreshInterval`.

Scheduling is also affected by the agent's internal self-tuning measures. The agent attempts to ensure that commands run at reasonable intervals, rather than all being executed at the same time, thereby causing a heavy load on the machine. Hence the scheduled time of a command, particularly a parameter, may be modified by these tuning measures, which may delay its execution until such time as the agent and the system are not so loaded. A command that needs to run immediately, such as a menu command or info box item command, is placed at the front of the run-queue with immediate execution time. These commands are user-initiated and are usually used to resolve problems, and therefore to override the user's wishes for immediate execution could be counter-productive.

10.8.1 PSL Internal Architecture

When a PSL process is scheduled from the main run-queue and executed, it is placed on the PSL run-queue, which is a queue of currently executing PSL processes. The agent runs multiple PSL processes through its interpreter, one at a time. Each executing PSL process is allotted a timeslice with a fixed set of low-level compiled instructions. The PSL interpreter runs the PSL process until its timeslice expires, or the PSL process executes one of the built-in functions that would otherwise cause it to block. The built-in variable `/pslTimeslice` can be set to any number to control the timeslice, allowing further agent tuning.

The PSL built-in functions related to the operating system that cause a PSL process to be paused are `system`, `execute` (but not `popen`), and some forms of `read` and `write`. Also, a number of built-in functions unrelated to the operating system cause a PSL process to suspend itself, including `locking`, `condition variables`, `response`, etc. When a PSL process is suspended by one of these built-in functions, it does not appear on either the main run-queue or the PSL run-queue; instead, it waits until it is woken by an external event, before the agent will place it back on the PSL run-queue to continue after completing the particular built-in function. For example, when executing a PSL `system` call, the process is suspended by the call to `system`, until the agent detects that the external process has executed and completed.

None of these PSL built-in functions can cause a blocking wait loop or a busy loop. They also do not suspend any of the other PSL processes, which continue to receive their timeslices. The agent will detect external events like file descriptor activity for a PSL process blocked on a `read` or `write`, and will then determine whether the given `read/write` operation has completed, and whether the PSL process is ready to continue. For PSL processes that are blocked on functions such as `locking` or `shared channels`, the execution of another PSL process (e.g., releasing the needed lock) can wake up another PSL process (i.e., the process waiting for the lock). Another example is the PSL `response` function, which is completed by receiving a message from the console when the user has entered all the data.

10.8.2 PSL Concurrency Issues

Since the PSL interpreter can be involved in multiple PSL processes at any given time, some concurrency issues arise. Since only one PSL process is actually having its instructions executed

at any given time (the others are on the PSL run-queue), individual PSL actions have no low-level concurrency problems. However, the level of atomicity is the built-in function, so two or more activities on shared data structures (i.e., the PSL object hierarchy) could have another PSL process execute between them. For example, if a PSL process is trying to set two variables:

```
set ("/a", 1);  
set ("/b", 2);
```

and another PSL process is using both these variables via PSL `get`:

```
if (get("/a" == 1) && get("/b")==2) { ... }
```

the second process can execute between the two PSL `set` operations, and thereby get only one of the settings.

Concurrency is certainly an important consideration, especially regarding shared channels. For this reason, PSL has various concurrency solving built-in functions such as locks and condition variables. These functions use internal agent synchronization and do not rely on operating system primitives for concurrency and synchronization. The semantics are similar to those offered by the operating system but are not based upon them.

10.9 Agent SNMP Support

PATROL has full agent-side support for SNMP, including MIB browsing and trap forwarding and receiving. These features are available via PSL and also externally via the PATROL MIB. The agent supports both SNMP V1 and V2, and will respond to manager requests in the same version of SNMP as was received.

The PSL language has a large number of SNMP-related built-in functions that are used to facilitate a variety of SNMP features. The PATROL agent is capable of SNMP trap receiving and forwarding via PSL trap sending functions. There are also SNMP `get`, `set`, and `getnext` facilities supported via PSL.

The situation regarding SNMP is significantly different in the 3.2 release of PATROL, compared to 3.0 or 3.1 releases. PATROL 3.0 and 3.1 agents had a built-in SNMP master agent. PATROL 3.2 makes use of the technology from the PATROL SNMP Toolkit by using both the master and subagent from that toolkit. This differs in that the PATROL 3.2 agent now has only a subagent contained in it, and can talk to the separate master agent process or can talk to some other SNMP master agent as required. This functionality offers advantages such as being able to use the PATROL SNMP support even if another master agent exists on the machine.

10.10 Inter-Agent Communication

Agent-to-agent communication is a powerful feature for more advanced Knowledge Modules. Examples of existing Knowledge Modules that rely on inter-agent communication include the PATROL KM for R/3 and the PATROL History Loader KM. Using inter-agent communication,

the monitored application can be displayed as a single logical entity by the various PATROL console components, and events across its components can be interrelated and correlated.

Several technologies are available for inter-agent communication. PATROL 3.0 allowed only SNMP-based access, but the 3.1 release of PATROL added features in the PSL facilities and the PATROL API.

The PATROL version 3.0 series offers limited support for agent-to-agent communication through the PSL SNMP functions. The communication occurs via the SNMP protocol, using the existing PSL functions via the PATROL Agent's SNMP subagent. Reliability can be ensured by using SNMP get/set rather than traps. All the information available via SNMP get/set from the MIB also allows powerful read access and limited write access. Although this method is still available with 3.1 and 3.2 agents, it is largely superseded by other newer methods.

PATROL version 3.1 added various new features for enhanced inter-agent access. The key technologies that are currently available in addition to SNMP are:

1. PATROL Event Manager remote agent event querying and remote agent event triggering
2. PSL remote agent execution for any PSL features (e.g., PSL `get`, `set`, etc.)
3. PATROL API capabilities

The choice can depend on the requirements. The event capabilities and remote execution methods are available via PSL and via the API. For a Knowledge Module, the PSL functions can be used. Alternatively, the API can be incorporated into a separate executable program, especially if that executable is already needed to interface with a vendor API. In any of these ways, a Knowledge Module that takes advantage of multiple agents is possible.

Also of interest is that a few other components can sometimes be used for inter-agent access. The PATROL CLI executable is really a wrapped command-line form of the PATROL API, and can clearly be used if an existing executable is required for some reason. The `pconfig` executable has also been used to perform some limited synchronization of agents via configuration settings. But this note is mainly historical, because this method worked as far back as version 2.0 of PATROL, and is now superseded by other methods.

10.11 Published PATROL Agent API

The PATROL API is a published API that permits both local and remote access to PATROL agents. It allows read operations to access PATROL data, and write operations to change the agent's configuration and behavior. This places the API as a clear integration point for aspects such as network console integration, reporting data access, and instrumentation of applications to send events into PATROL.

The PATROL API is delivered as a set of C libraries and header files that can be linked into any integration component. Although not specifically based upon C++ classes or other features, the API can also be linked into a C++ application. The minor modifications to permit C++ compilation have been performed.

This library is a wrapper around the methods needed to connect to and query a PATROL agent. Both the TCP and UDP connection methods are incorporated within the API to allow the remote access to agents. Multiple agents can also be connected to by the API so as to permit a kind of middle manager or console-style integration, such as that achieved via the various PATROLVIEW products. The API also allows local agent interfacing, which can be useful when using the API for advanced application instrumentation.

The capabilities of the API extend to a variety of areas of PATROL. The object hierarchy can be examined to determine the current state of any object, instance, or parameter. The event subsystem can also be queried to gather event information. Changes to the agent are also possible via configuration changes or even to emit new events into the agent.

An important feature of the API is that it supports both synchronous and asynchronous access. Synchronous access is the simplest form of call to the API, which leads to a blocking call, but not to a busy wait. The asynchronous methods allow the API client to register for information about an event or status, and the agent will notify the API in a non-polling mode when a matching event or status is achieved. This asynchronous mode therefore permits an advanced event-driven architecture when using the API to respond to events generated by Knowledge Module alarms.

Do not be misled by thinking of the agent API as the only API for integration with PATROL. A variety of other interfaces can also be thought of as methods of accessing the agent, including the PSL API, the SNMP API, and even the CLI that is really just an executable wrapper around the most common API requirements.

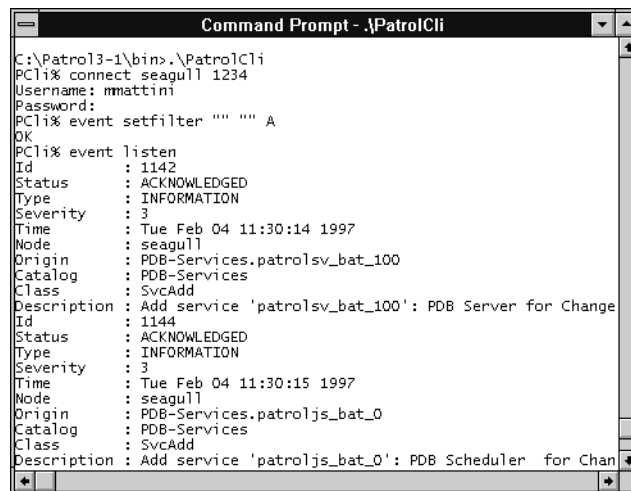
10.12 PATROL CLI

The Command Line Interface (CLI) for PATROL is a standalone executable that allows both local and remote access to agents. The CLI is available as the `PatrolCli` executable with the standard PATROL distribution. Two distinct usage methods exist for the CLI:

- interactive mode
- batch mode

The interactive interface is a human-friendly interface. The CLI starts up and presents a prompt to the user for a command. A sequence of entered commands and their execution by the CLI occurs. In this mode, the CLI can be a useful tool for interactive analysis of the state of one or more agents. For example, when dialing in from home, you can run the CLI and query the status of the agent's object tree and most recent events.

The batch interface to the CLI allows its use in scripts that examine one or many agents. The command-line arguments to the CLI permit it to perform all the commands that the interactive interface permits using the same syntax.



```
Command Prompt - .\PatrolCli
C:\Patrol3-1\bin>.\PatrolCli
PCli% connect seagull 1234
Username: mmattini
Password:
PCli% event setfilter "" "" A
OK
PCli% event listen
Id      : 1142
Status  : ACKNOWLEDGED
Type    : INFORMATION
Severity: 3
Time    : Tue Feb 04 11:30:14 1997
Node    : seagull
Origin  : PDB-Services.patrolsv_bat_100
Catalog: PDB-Services
Class   : SvcAdd
Description: Add service 'patrolsv_bat_100': PDB Server for Change
Id      : 1144
Status  : ACKNOWLEDGED
Type    : INFORMATION
Severity: 3
Time    : Tue Feb 04 11:30:15 1997
Node    : seagull
Origin  : PDB-Services.patroljs_bat_0
Catalog: PDB-Services
Class   : SvcAdd
Description: Add service 'patroljs_bat_0': PDB Scheduler for Chan
```

Figure 10-1 PatrolCli Interactive on Windows NT

Architecturally, the CLI is implemented as an ASCII front-end to a lot of the API functionality. The interactive and batch commands use the same syntax and differ only on whether the command must be manually entered for interactive mode or placed as a command-line argument for batch mode.

10.13 PATROVIEW Integration Suite

The PATROVIEW products are basically integration modules with other network management consoles or service desks. The breadth of coverage of these integrations has helped represent BMC Software as the “Switzerland of application management.” PATROL is enabled by PATROVIEW to plug right into an existing management infrastructure at the site. PATROL is expected to add complementary database and application management features to an existing network and system management scheme. PATROVIEW serves to permit PATROL to do so using off-the-shelf components. Some examples of the PATROVIEW integrations include:

- HP OpenView IT/Operations
- Tivoli TME 10 NetView
- CA-Unicenter TNG
- Remedy Action Request System
- HP OpenView Network Node Manager

PATROVIEW components provide several basic areas of functionality. Although not all integrations have the same feature levels, some of the main features include:

- Event forwarding from PATROL to the console or help desk
- Graphical representation of the PATROL objects within the console's GUI display
- Launching in context of the PATROL console

The PATROLVIEW products are based on the PATROL API (see Figure 10-2). The PATROL Event Translator (PET) is a component that is a higher level of abstraction over the PATROL API. This PET component standardized some of the PATROLVIEW-specific issues of the use of the PATROL API, such as the easy handling of multiple agent connections, and their configuration in a standard manner for all PATROLVIEW products.

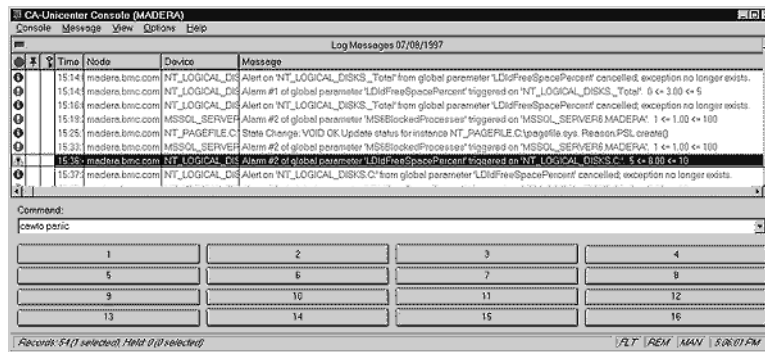


Figure 10-2 PATROLVIEW for Unicenter TNG

The architecture of PATROLVIEW integrations can be either agent-to-agent or agent-to-console. The PET component allows either local or remote access to one or multiple PATROL agents. This takes the PATROL events or objects from the agents and sends them to the integrated console or help desk. So the PATROLVIEW component is a combination of the PET and the integrated third-party product API. The architectural choice between agent-to-agent or agent-to-console depends on where the console API permits connections. For example, in the case of Tivoli, the PET can launch an event into TME via its API at the management server or endpoint (agent-to-agent), or alternatively into the component at the TEC console (agent-to-console). Another example is in CA-Unicenter TNG, where the PET takes events and pushes them into the Unicenter console's object repository, giving rise to an agent-to-console architecture.

SNMP is another method used to provide PATROLVIEW integrations. This is an effective integration method, since most network management consoles have powerful SNMP support. However, API-based integration is the trend with most PATROLVIEW components. The majority of consoles and help desks also have other non-SNMP APIs, making plugging in a component easy between PATROL and the other console that uses both the vendor API and the PATROL API (via the PET). SNMP trap sending can clearly be used to forward PATROL events to some other SNMP console if this is not the case for the vendor's console.

The screenshot shows a web browser window titled "Display — PatrolEvent-TroubleTicket (nanou)". The interface is divided into several sections:

- Actions:** A tab at the top.
- Entry-Id:** 000000000031158
- Assigned-to:** jml
- Submitter:** jml
- Last-modified-by:** jml
- Create-date:** 02/26/97 11:26:23
- Modified-date:** 02/26/97 11:26:23
- Status:** A list of status options: New, Assigned, Fixed, Rejected, Closed. "Assigned" is selected.
- Short-Description:** Patrol Event from nanou:FILESYSTEM.rootFSCapacity
- PE-Status:** Open
- PE-Type:** Alarm
- PE-Severity:** 4
- PE-Source-ID:** 0000
- PE-ID:** 988
- PE-Event-Class:** 11
- PE-Args:** Alarm #2 global FSCapacity
- PE-Diary:** <NULL>
- PE-Time:** Wed Feb 26 11:26:23 1997
- PE-Node:** nanou
- PE-Origin:** FILESYSTEM.rootFSCapacity
- PE-Owner:** Patrol
- PE-Handler:** Patrol
- PE-Event-Catalog:** 0
- PE-Life-Expectancy:** STORED
- PE-Description:** Alarm #2 of global parameter 'FSCapacity' triggered on 'FILESYSTEM.root'. 98 <= 100.00 <= 100
- PE-History:** PE_STATUS = Open
- Number 1 of 1**
- Buttons:** Previous, Next, Dismiss

Figure 10-3 PATROLVIEW for Remedy AR System

The scalability of PATROLVIEW architectures is the same as the PATROL Console. The same basic comments apply about how it scales to hundreds and thousands of agent connections. Scalability is not limited by the PATROL components, but is sometimes limited by the third-party products, although these are usually capable of handling adequate numbers of events or objects. The PET component of the PATROLVIEW architecture is reliant on the PATROL API, and hence the same comments about console scalability apply. For example, the discussions around how the console-agent protocol minimizes traffic by sending only critical events/objects or visible objects also apply to PET-agent scalability.

10.14 PATROLWATCH for Web Browsers

The PATROLWATCH for Web Browsers product is an integration between PATROL and web browsers. The supported web browsers are Netscape Navigator and Microsoft Internet Explorer. PATROLWATCH allows the web browser to display a graphical iconic view and an outline view of the state of the PATROL objects in more than one agent. Also, an event view similar to the PATROL Event Manager is available. All this displaying occurs in the native web browser interface.

The PATROLWATCH architecture is similar to that for the various PATROLVIEW components. The PATROL API is used by a component of PATROLWATCH to communicate with multiple agents. This API is connected to the web browser via a combination of HTML, JavaScript, and CGI accessed commands. PATROLWATCH does not require a web server on each agent, because it uses the PATROL protocol to communicate with the remote agents via the PATROL API.

10.15 PATROL Pathfinder

PATROL Pathfinder is an integration component that plugs into Microsoft Explorer on both Windows 95 and Windows NT to display the PATROL objects in this view. This allows a standard outline view of all PATROL objects that is familiar to Windows users. The objects are shown with their status and can be drilled into to launch the appropriate console or administrative tool in context.

PATROL Pathfinder is another component that relies on the PATROL API behind the scenes. The access to the Knowledge Module objects is via the API requests. This allows the objects to be shown in their correct statuses. Pathfinder can also display the objects from the various PATROL-DB database administration tools.

10.16 Architectural Strategies

For the remainder of this chapter, I examine some of the more philosophical aspects of the architecture. The PATROL architecture delivers on a number of strategies:

- Homogeneity
- Simplicity
- Centralization
- Portability
- Technology independence
- Openness
- Extensibility
- Interoperability

The qualities of homogeneity, simplicity, and centralization are so central to the PATROL concept that they need little introduction. PATROL is all about providing a simple, consistent, and centralized view of the enterprise. The basic strategy of PATROL is to map the complexity and heterogeneity of the existing diverse enterprise software packages back to a simple, central underlying data model. This model is displayed to the user in an iconic view allowing drill-down to the details. Other views include an outline indented semi-graphical view, or a fully text-based event view. For example, a machine icon is the top-most view, the Oracle server is the next level, and the Oracle users or tablespaces within that server are the next level.

Simplicity is achieved by holding the details of monitoring and administrative commands in the packaged Knowledge Modules and showing the user only the resulting data as part of the populated data model. The data model is also very simple, holding “instances” to represent the software objects that are discovered, such as database servers, and “parameters” to represent the measured values about these objects, such as cache hit rates. This model allows a simple, homogeneous view of all software objects monitored by PATROL. These are displayed in various views, depending upon the choice of console. Representations include graphical iconic hierarchies, textual event lists, or indented outline views.

This simple representation of monitored software entities is shown in a centralized view. A single desktop can watch the status of an entire enterprise. This central view can be part of the PATROL console (for UNIX or Windows NT), a web browser, or a variety of other third-party consoles or help desk components with which PATROL interoperates via the PATROLVIEW suite.

10.16.1 Portability

The PATROL infrastructure forms an abstraction layer above the various underlying components. This abstraction handles the details of the operating systems that run the applications. However, it also hides the details even further by abstracting the application-level details into the data model.

The first level of portability is the operating system independence of PATROL. This is achieved through the use of an intelligent agent on each operating system that supports the same basic feature set in a portable way. The PATROL Script Language (PSL) is a platform-independent programming language that runs identically on all agent platforms. PSL abstracts away a lot of the operating system dependencies and moves detailed processing onto the PATROL layer. Knowledge Modules benefit from the portable nature of PSL, allowing the same application to be monitored easily on different operating system platforms. For example, Oracle is supported by a Knowledge Module across UNIX, OS/2, VMS, Windows NT, and MVS in this way.

The next conceptual level of portability is the abstraction of the application layer itself. This means that PATROL brings about an abstract model over the underlying structure of a database or application. For example, the various database Knowledge Modules build a logical model of the database server, tablespaces, users, and tables. This logical view is used in the iconic graphical displays, event processing, and in user-oriented tasks that can be run in an ad hoc manner. This brings an abstract layer of processing to the database. It also brings a homogeneous model across all the different RDBMS components that are monitored on the enterprise.

10.16.2 Interoperability with Frameworks and Help Desks

The strategy taken by PATROL has been to interoperate with as many other complementary systems and network management consoles as possible. This means that the investment in an existing management infrastructure for networks and systems is not discarded. PATROL simply adds to these infrastructures to bring them up to the higher level of application management. This

strategy has spawned the entire suite of PATROLVIEW products, which are integration modules with a variety of third-party products. For example, some of the products with which PATROL will interact via PATROLVIEW include:

- HP OpenView IT/Operations
- HP OpenView Network Node Manager
- Tivoli TME/TEC
- CA-Unicenter TNG
- Remedy AR System
- Solstice Domain Manager
- Cabletron SPECTRUM

This PATROLVIEW suite allows PATROL's database and application management to be plugged into existing management schemes. PATROL complements the above products in that it does not provide network or device management. Some overlap exists in the operating systems monitoring features in the PATROL Knowledge Modules for Unix, Windows NT, VMS, and MVS. However, much complementary functionality is available for databases and applications.

PATROL does not offer framework services such as software distribution, user administration, or job scheduling. Instead, it focuses on the various aspects of application management that separate the discipline from more generic systems and network management issues. Although PATROL can clearly stand on its own to build an application management infrastructure, it can also work as part of a more general framework solution. Hence, PATROL is framework-agnostic through integration, and will work either with or without any existing management infrastructure.

10.16.3 Technology Tolerance without Dependency

Another form of the pragmatic strategy involves the support for particular technologies by PATROL. The strategy is to tolerate or support technology areas without having a dependency upon them. A good example is SNMP, which is supported by the PATROL agent, but is not the basis of the PATROL network transport. PATROL can be used entirely without any SNMP support, or alternatively, can be plugged into an existing SNMP regime.

Another example is PATROLWATCH for Web Browsers, which delivers web browser views of PATROL agent data. This facilitates integration into internet technology, without forcing PATROL to use this technology as the basis.

Simplicity is the underlying idea behind the technologies used by PATROL. The network transport layer is based on TCP/IP (via TCP or UDP with added reliability layers), and does not require any of the more complicated distributed data models. This usage is facilitated via the simplicity of the underlying data model for PATROL Knowledge Modules. This model works well in representing all of the required dependencies of objects within the software world.

10.16.4 Openness and Extensibility

The PATROL architecture is such that it is fully open for integration or extension. The most obvious example of openness is the published PATROL API for accessing data from the agents generated by Knowledge Modules. However, the API is not the only example. The Knowledge Modules themselves, via PSL, are open and can be extended to meet differing requirements at user sites. Tools also exist that open up various issues such as the historical database of parameter values, the agent and console configurations, and other areas. These facts all mean that PATROL is easy to bend to fit into any required monitoring or administration paradigm.

Extensibility is not limited to simple customizations to meet particular requirements. The PATROL Knowledge Module Toolkit (SDK) allows development of new Knowledge Modules to monitor in-house developed applications, or any other software application. The value of the PATROL infrastructure can be applied to any application, regardless of whether it is a commercial off-the-shelf product, or an in-house project.

Case Study: BMC's Service Assurance Center

The introduction of application management can offer astounding results. BMC's Service Assurance Center is a recent initiative of the IS department of BMC Software that demonstrates the advantages of using application management in a real-world corporate environment. The deployment of PATROL has resulted in significant improvements to the service levels offered by the IS department. Paradoxically, it has done this while at the same time actually reducing the levels of personnel power focused directly on day-to-day achievement of availability.

BMC Software, apart from being the owner of PATROL, is also a modern corporation in which the IS department has responsibilities similar to those in any other corporation. It has internal customers who complain just as much as anywhere else. The main requirement is to ensure the availability of the business-critical applications. The list of mission-critical applications inside BMC Software includes:

- WWW—external BMC web site critical to marketing and sales
- CWW—intranet BMC web site critical because of internal information sharing
- Vantive—customer support service desk and case tracking automation software
- Siebel Sales Enterprise—sales force automation software
- HP OpenMail—corporate email software
- Oracle Applications—corporate financial and HR software

Being in the software business means that some of these applications such as Vantive and Siebel are critical to BMC's business of development and sales of software. Other applications such as email and web servers are more general in their scope and are mission-critical in most industries.

11.1 Kickstarting Application Management

The first step toward achieving improvements through application management was an organizational rather than a technical step. The Service Assurance Center is not only a physical place with centralized monitoring screens, but also a new independent group within the company. This group started with only a few people but has since expanded because of its success. Most of the new people transferred from other IS groups as stresses on their time diminished because application management began to pay off.

The Service Assurance Center group is one that is effectively tasked with application management across all the technology layers and throughout all domestic and international sites. This group differs from the previous technology-oriented groups. These each have responsibility for a different area: application administrators, database administrators, system administrators, and network administrators. Instead, the Service Assurance Center group is tasked centrally with the overall achievement of service levels through application management. This group is also granted direct access to personnel from the different technology level groups. This cross-discipline responsibility avoids unproductive finger-pointing between groups during crunch times such as downtime diagnosis. The application management group also has clear accountability with respect to application availability and service levels. When a service is down, clear pointing is directed at the responsible group. This makes for a high-pressure and high-value group that should contain the best staff. The focus of the group on the service rather than the technology shows alignment of personnel individual goals directly with the true business goals of the IS department.

Several technical steps occurred in the introduction of application management. The first step was the deployment of PATROL Agents on every server. This led to the immediate use of off-the-shelf Knowledge Modules, particularly the PATROL KMs for Unix and Oracle, for the monitoring of server machines and databases. The second step was the focus on the application level. The web servers were covered by the PATROL KM for Internet Servers. However, at the start of the project, not all the other applications were covered. Off-the-shelf support was only available for Oracle Financials, with no support for Vantive, Siebel, and OpenMail. The development of custom PATROL KMs for these applications became a key part of the use of PATROL.

Custom Knowledge Module development has a number of distinct phases. The first level is monitoring and alerting, and the second level is recovery and correction. This second level can be a risky process. How much should one trust an automation system? However, growing pains were minimized because the Service Assurance Center group used the other groups as the resources to help define the corrective actions. The expertise in the heads of other staff and existing scripts was leveraged into the recovery actions.

Although PATROL is the highest-level monitoring tool, it is not the only one used in the Service Assurance Center. In particular, network management is achieved through various other tools, including COMMAND/POST, HP OpenView, CISCO Works, 3COM Transcend, and

Concord Network Health. These show how application management complements rather than replaces network management.

11.2 OpenMail

HP OpenMail is the email standard used by BMC today. Unfortunately, the email system was not a very reliably deployed system at the start of the project. The severity of the problems depended on the level of who was on the phone complaining about the lack of email service. Availability levels were hit or miss. Outages were common, and sometimes took a long time to diagnose and required experienced administrators to do so. Email was not a pleasant situation for BMC IS.

PATROL was the basic technique chosen to resolve the problem. The development of a custom PATROL KM for OpenMail (for internal company use) automated a great many of the tasks that mail administrators had previously shouldered. The aim was to embed the expertise of the OpenMail administrators into an automated recovery system.

The foremost concern was the monitoring of the full OpenMail server software. The PATROL KM for OPENMAIL_DAEMONS was created to monitor each daemon. This simple feature gave early warning about major failures. The next level is that recovery actions were added to recover, kill dependent daemons, and auto-restart the failed daemons.

One of the most common causes of problems with mail delivery was corrupted messages appearing on message queues. Of the 30,000 messages handled each day, usually at least one would cause such a problem, particularly in the internet mail gateway. This corrupt message would cause queues to back up and undelivered mail to accumulate. The first step to improving recovery from this situation was to monitor queue lengths to get early warning about the problem and diagnose which queue was affected. This gave early alerting and notification, but did not auto-correct the problem.

The next step was auto-correction by removing corrupted messages from the queues into a separate place for later handling. Corrupt messages are detectable via OpenMail utilities that scan header fields. These utilities are auto-launched by the PATROL KM to detect corrupt messages and move them. Still, the same number of failures occur due to corrupt messages, but they are resolved in a matter of seconds rather than the previous situation of typically 30 minutes or more. Users can no longer perceive any noticeable delays in message delivery. In fact, users now typically think any outage was a typing mistake on their part because it works the second time they try to connect. This perception means far fewer help desk calls about email problems.

These features are just two of the main components of the PATROL KM for OpenMail. In fact, literally dozens of KM classes now exist for different aspects of OpenMail. Notably, the PATROL KM for OPENMAIL_ACCELERATOR monitors system CPU utilization of OpenMail processes to detect performance problems.

The success of the PATROL KM for OpenMail is seen in that the parameter that measures its availability levels shows it in the range of 99%. Outages and problems occur, but these are rarely perceived by users. The automation of OpenMail monitoring also achieved numerous side

benefits. The burden on OpenMail administrators reduced significantly, and the effective number of people assigned to this task has dropped by 50%. This freed administrators to other tasks, including further exploitation of PATROL in other areas. Another interesting side benefit of the reduction in fire-fighting is the extra time available to investigate full solutions to the OpenMail problem. For example, patches from HP helped address certain problems, which administrators had previously not had enough time to investigate or test.

Another rather amusing outcome is that dealing with HP support about OpenMail fixes has even become a little more difficult, simply because the outages are not so easy to demonstrate. Administrators have to dig through the history collected by PATROL to provide evidence of outages.

11.3 Vantive

Vantive is a help desk and call tracking application that is used by BMC for customer support to track calls, by product development to track bugs and requests, and by IS as a help desk. Vantive is mission-critical to BMC support for prompt call handling and resolution.

One of the largest concerns for Vantive early in the project was performance. Two main types of users were found: interactive and batch users. Performance problems occurred when interactive user jobs competed with longer batch jobs for CPU time. Interactive users had small requests and expected snappy response time, but the large jobs often prevented this. Owners of longer batch jobs, typically in product support, would not react well to IS requests that their job be delayed in favor of interactive users.

The PATROL KM for VANTIVE_ACCELERATOR was developed for in-house use to solve the conundrum. This KM monitors all the Vantive jobs and automatically adjusts the priority of jobs according to various criteria including the length of time that the job has been executing. This improved the speed of interactive jobs and achieved consistency in response times for both interactive and batch jobs. The achievement of *consistency* of response time may have been more important to the user community than the actual improvement.

User level monitoring is a key component of other aspects of the PATROL KM for Vantive. Measurement and alerting occurs on issues such as response time and application-level errors. Whereas previously the help desk waited for users to call with problems, they can now see the red icons and receive notifications. The SQL text is also collected for problem users for faster problem diagnosis.

Chargeback data gathering is another example of per-user monitoring. The metrics used for process acceleration and response monitoring also record data that can be used to feed chargeback reports. The kinds of data that are available include CPU usage, memory usage, job counts, transaction times, and other per-user statistics. Although BMC IS does not currently make internal use of formal chargeback reports, this capability is just waiting for the CIO to ask for it!

11.4 Response Time Measurement

An interesting two-pronged approach for response time measurement turned out to be very effective. BMC IS is responsible for a global network of users throughout the U.S., Europe, and Asia/Pacific. Servers are in each area, of course, but international users also connect back to the U.S. for some applications. BMC needed to measure the response time for each of these areas. The two-pronged approach involved both an “out” and an “in” response measurement strategy.

The first method was to watch user connections at the server and try to extrapolate that “out” back to the client. This involved gathering network information about each client user from the application server. For example, the VANTIVE_NETWORK application gathers the list of IP addresses of user connections and performs monitoring of low-level network response time such as bandwidth usage and data loss (particularly with Frame Relay). However, this is only local network traffic information. One method it uses to get information about the performance along the path back to the client is the UNIX `traceroute` command, which is a good measurement tool, provided that the polling cycle is long enough so as not to flood the network with these tracing messages. This method is also used immediately for diagnosis when an alert is generated on the other response parameters and gathers details of response or connection failure on the route.

The second method was to work from the client-side “in” to the servers. BMC IS decided to “sacrifice” a few PCs in the name of service assurance. Instead of any server-side simulation, real client-side data was generated by placing a PC on the corporate sites across the WAN. Knowledge Modules were developed that launched simulation scripts for the various applications. For example, the PATROL KM for VANTIVE_RESPONSE performs a dummy login to Vantive, opens an inbox, creates a new case, and logs out. The response time for this activity is measured in a KM parameter. This parameter can be viewed locally by IS in the Service Assurance Center by connecting to that remote agent. Alerts on this parameter are raised through the notification methods either locally or at the remote site.

Actually, these remote PCs are not really sacrificed, since the load from doing these checks every 5 or 10 minutes is not high. Most of the time, these KMs are waiting for the response time. Hence, these PCs run multiple Knowledge Modules and are used for other tasks.

11.5 License Metering

Application management is not only about real-time problem resolution. The measurement of metrics also offers a few notable side benefits. User monitoring leads to the possibility of license management by comparing user connection counts with actual licenses purchased. This strategy is used by the Service Assurance Center in the off-the-shelf and custom application Knowledge Modules.

The basic method is to measure user counts via their concurrent connections. This gives real data for decisions on license purchases and negotiation with vendors. This is measurement of the situation without intervention.

The second phase of license management is intervention to reduce license count usage, ensure that licenses are available, and prevent license overruns on soft-fail applications. A good example of this case is the PATROL KM for Vantive that monitors the number of users, compares it with the number of available concurrent licenses, and alerts when the number approaches the maximum. The recovery action on this alert finds the user with the highest idle time and automatically logs them off. This ensures that users can almost always get a license and forces compliance with the license count.

11.6 Summary

BMC IS is a case study in the return on investment from the intelligent usage of PATROL for application-level management. Positive returns were achieved through a combination of off-the-shelf components for basic server and database event monitoring, and the development of custom Knowledge Modules for mission-critical applications. Here is a summary of some of the benefits that arose:

- Database and application license metering for usage in vendor negotiations
- More satisfied end users as application availability improved dramatically
- Rapid auto-correction of problems in OpenMail
- Auto-tuning of Vantive jobs to achieve consistent response times
- OpenMail administrators reduced by one-and-a-half and freed to other forward-looking tasks
- Help desk calls reduced by 85% through reduced problems and user information
- Help desk staff load reduced and available for other projects
- Less than expected growth in IS headcount needs

This does not include the intangible benefits from a reduced stress level. The reduction in fire-fighting improved the overall IS department. One outcome was increased delivery on other IS projects under development. Some of these current projects include testing the PATROL KM for Oracle Applications, the formalization of a Capacity Planning process, and the introduction of IVR voice automation into the help desk.

The process to achieve this result did not take a great deal of resources or time. The basic steps in the introduction of application management were:

1. Formation of an application-level group focused on service rather than technology
2. Off-the-shelf use of UNIX, Oracle, and Internet Servers Knowledge Modules
3. Custom Knowledge Module development for Vantive, Siebel, and OpenMail
4. Collaboration with administrator groups to define corrective actions

More information about the use of PATROL by BMC IS is available. An interesting video from BMC is titled "The CIO's Perspective: The Benefits of PATROL" and includes general

project discussions and live demonstrations of the custom Knowledge Modules. Also, consulting assistance, documentation, and software are available to achieve a similar Service Assurance Center with PATROL in other corporate IS sites.

Business Application Management

Many different technologies and applications are in use in a typical corporation. An important task is to monitor all of these to complete the application stack—from the operating system through the middleware through the database to the applications. PATROL provides Knowledge Modules for the monitoring of all the various components in the enterprise. Although categorizing individual products is hard because some of them offer so much functionality, the basic areas that are covered include:

- Enterprise Resource Planning (ERP) applications
- Internet servers and applications
- Groupware, email, and messaging applications
- Middleware and transaction processing systems

Currently, PATROL has Knowledge Modules for four key ERP application suites. Other ERP applications are expected to be covered via third parties through the PATROL Developer Network/PATROLMARK development partnerships. The current key ERPs covered are:

- SAP R/3
- Baan Applications
- PeopleSoft Applications and PeopleSoft HRMS
- Oracle Applications

From the non-ERP categories, PATROL monitors a number of commercial products. These Knowledge Modules monitor a critical layer of the application stack. Some of the applications covered by Knowledge Modules in these functional categories include:

- Lotus Notes and Lotus Domino
- Internet Servers
- Microsoft Exchange Server
- MQ Series
- TUXEDO
- Transaction Server—for Open Systems and OS/390

Other commercial products in these areas are likely to be supported via additional Knowledge Modules developed by partners in the PATROL Developer Network program.

12.1 ERP Applications

Modern businesses are increasingly relying on applications that control the critical processes that are part of general business management. This category of applications has become known as Enterprise Resource Planning (ERP) applications. The adoption rate of packaged ERP applications is clearly on the rise, with the replacement of home-grown solutions by commercial off-the-shelf solutions from major vendors such as SAP, Baan, PeopleSoft, and Oracle. The rise of these packaged ERP solutions occurs as organizations prefer to focus on their core business competencies rather than develop and maintain software. The maturity of the commercial offerings is such that the major vendors offer products that cross all the industry segments, rather than offering products specific to vertical industries.

ERP applications aim to centralize all of the major business functions into one application. These components touch on all aspects of the overall business and its need to manage information. Some of the typical ERP components include:

- Financials—accounts payable, accounts receivable, taxation, expenses, etc.
- Human resources—personnel management, payroll, application management, etc.
- Production management—inventory, distribution, suppliers, distributors, procurement

These various components are typical of business requirements regardless of the industry segment. However, not only is each industry unique, but each company also has its own practices. For this reason, the major vendors offer very open and customizable product architectures that permit the flexibility to achieve the desired changes in business processes.

The ERP application market, like any other market, is split into a number of major players and many smaller niche players. The top vendors offer cross-functional suites that address all the horizontal segments of functionality, and may also offer some components aimed at particular industries. Many of the smaller players concentrate on particular industries and their application requirements. At the top of the spectrum, the “top three” vendors that focus centrally on the ERP application market are:

- SAP R/3
- Baan Applications
- PeopleSoft Applications

The other major player is Oracle's applications division, which provides business applications that fit into the ERP segment. Oracle is significantly different to the other players in this group in that ERP applications are not its only focus, and are instead another layer of products on top of the Oracle database product line.

12.1.1 Why Use ERP Applications?

The demand for ERP applications has exploded recently. The top ERP vendors have recorded high growth in financial returns as more companies world-wide adopt their products for vital business functions.

The return on investment to corporations using ERP applications has demonstrated itself time and again in real world situations. Some of the payoffs occur horizontally across all industries, such as the removal of the development and maintenance costs of in-house developed solutions, or the simplification and increased productivity from using one single application suite.

Other returns are specific to particular verticals in the marketplace. A good example is the use of ERP applications to manage product inventory in the manufacturing industry. Full use of ERP applications can lead to a more effective sequence from raw material or component supplies to the distribution of the end products to customers. The gains occur in supply-chain management through faster turnover of inventory with less time that inventory sits on the factory floor due to just-in-time delivery. Another gain is that the overall time taken for manufacturing a product can be reduced on average through streamlined use of equipment and supplies.

Although ERP applications are valuable here on their own, the ultimate payoff occurs when the web meets ERP. The most desirable situation occurs when suppliers and customers can access the manufacturer's ERP and its information directly from their own ERP applications. The aim is a form of Electronic Data Interchange (EDI) without the intermediate step of re-entry of transactions, but instead the direct translation between the two ERP applications. EDI has long been an improvement over manual solutions via phone or fax, but direct ERP interaction is the next logical step.

The only shadow of doubt regarding returns from ERP applications has been the lengthy implementation cycles for the products. An ERP application usually requires a large commitment from many departments within the organization. It may also require significant changes in business processes to achieve the desired results. Longer cycles can result from changing internal business processes to match the desired processes to be supported by the ERP application, or the reverse option of customizing the software to match internal processes. Thus, implementation delays come from both internal and software issues. Consulting and programs such as Business Process Re-engineering (BPR) efforts are the only way to address internal issues. The fact that people issues are the most common cause of implementation delays or failures cannot be

addressed by the software itself. The key issues are the same for any project, including motivation, training, and competency.

The ERP vendors have begun to address the software implementation issues themselves with improvements in both external training and implementation services, and product feature enhancements. SAP has started the Accelerated SAP (ASAP) program to reduce R/3 implementation cycles. Baan has introduced Orgware as part of the Baan IV release, a suite of tools and procedures to assist in the deployment process for Baan applications. All vendors likely will focus efforts not only on feature enhancements at the top end, but also on reduction of the implementation costs because these costs cut only from the ROI from any ERP project.

12.1.2 Management of ERP Applications

The management of ERP applications is a crucial factor in their success. With vital business functions performed using ERP applications, any downtime or performance degradation is immediately a critical event. Application management provides the solution by ensuring high availability and good performance through continuous monitoring and proactive intervention.

ERP applications are perhaps the area of highest ROI on an application management project. The critical nature of the applications makes the insurance provided by application management very valuable. The application-level focus of true application management is an important aspect of application management. Be sure not to try to stretch traditional network and systems management to cover applications, because it will inevitably fall short. True application management means taking the ERP application as an entire entity and watching for availability or performance problems from the top level, breaking down through the underlying technology layers only for diagnostic purposes. The focus of management at the application level rather than at the technology level brings the ERP management function into close alignment with business goals.

The returns from complementing ERP application projects with application management are similar to the typical returns from application management, but are magnified by the more critical nature of the managed applications. The most basic return is the increased availability and reduced unplanned outages that are achieved through continuous monitoring, automatic corrective intervention, and early notification. These find problems early in the cycle, and give the administrators time to resolve the problem before its impact escalates.

Another key value of application management for ERP applications is in the improved productivity of administrative personnel. By automating tasks and providing packaged expertise about the application, the high-paid ERP experts can spend more of their time on business-oriented issues rather than fire-fighting problems that were detected too late in their cycle. Application management is the smart robot that watches the ERP application and notifies the administrator when a critical issue requires attention.

To realize these advantages, PATROL provides Knowledge Modules for all the major ERP application vendors listed earlier.

12.2 SAP R/3

A very visible player in the ERP market is German behemoth SAP AG. Founded in 1972, Walldorf-based SAP has become a worldwide leader in the market for packaged enterprise business applications with revenues into the billions and more than 11,000 employees. The origination of the SAP name was “SystemAnalyse und Programmentwicklung,” which is now “Systems, Applications, and Products in Data Processing.” The R/3 release brought SAP into the client/server applications market from its mainframe beginnings with the R/2 product.

The R/3 product provides applications for complex business processes. The implementation of SAP is typically combined with a BPR program to leverage the new technology with improved business processes. The key is to provide “return on information” through the use of SAP applications to control all business information processing. Literally hundreds of packaged business processes are provided with the R/3 product to realize the advantages.

R/3 offers a wide range of components for different business operations and units. Here is a summary of some of the business areas and the vertical industry packages supported by R/3:

- Aerospace and defense
- Insurance
- Banking
- Automotive
- Oil and gas
- Pharmaceuticals
- Chemicals
- Public sector
- Consumer products
- Retail
- Engineering and contracting
- Telecommunications
- Health care
- Utilities
- High technology and electronics

Across these industries, R/3 provides packages for human resources, financials, and logistics management, and all their various sub-categories. However, R/3 is not positioned as a tool for IT managers, but as part of shaping the business through BPR and technology combined.

SAP R/3 has become an industry in its own right. A myriad of partners are providing add-on or complementary products and services. SAP offers its own implementation aids through the Accelerated SAP (ASAP) program, and through partners in its TeamSAP program.

With thousands of deployed R/3 sites, many users exist. Annual SAPPHIRE user conferences take place in many countries worldwide. Dozens of books specifically on SAP R/3 development, usage, and implementation are available.

12.2.1 Managing SAP R/3

R/3 is a three-tier architecture with presentation, application, and database layers. Administration of SAP typically involves a Basis administrator at the application layer and a DBA for the DBMS layer. The basic method of managing R/3 is having a Basis administrator use SAP's Computer Center Management System (CCMS). Unfortunately, although CCMS is a monitoring tool, it requires manual checking of alerts rather than proactive notification. Using CCMS in this way effectively loads a Basis administrator with the responsibility for continuous monitoring using a tool designed more for configuration and diagnostic usage than for 24x7 hands-off monitoring. CCMS is powerful and provides a wealth of system information, but it is not a full monitoring framework. Although workable, this method of application management is not the most efficient deployment of expensive human resources.

A better solution is the use of an automated application management tool. For this reason, PATROL provides a Knowledge Module for the management of R/3 systems (see Figure 12-1). The Knowledge Module complements CCMS with full automation and continuous monitoring. In fact, the PATROL KM for R/3 intercepts all CCMS alerts so that they become proactive through notification or recovery actions in PATROL. CCMS data is also archived as history data to permit graphing and further analysis such as post-mortem diagnosis or capacity planning.



Figure 12-1 PATROL Knowledge Module for R/3

The PATROL KM for R/3 also adds a holistic view of the system and RDBMS underneath the R/3 system. This is a delivery on the promise of application-level management. Some of the key features of the KM include:

- Combined R/3 system, operating system, and RDBMS monitoring
- User monitoring

- Printer monitoring
- Work Process monitoring

The PATROL KM for R/3 incorporates monitoring statistics at all the application layers. Full support is available for UNIX and NT platforms and for RDBMS including Oracle, Informix, and Microsoft SQL Server. RDBMS monitoring is provided by the R/3 Knowledge Module itself, and more advanced monitoring is achieved through combined usage of the R/3 and RDBMS Knowledge Modules. Sites commonly use both the PATROL KM for R/3 and a PATROL KM for Oracle, Informix, or Microsoft SQL Server.

User monitoring occurs from three points of view. The first is problem detection through determining when a particular user is causing a problem by tying up one of the processes. The second is performance analysis by determining the per-user performance usage. The third is recording user requirement profiles to map against performance characteristics and determine future capacity needs of the R/3 system.

Printers can be a major headache in R/3 management, and PATROL provides the aspirin. The printer monitoring component of the PATROL KM for R/3 includes both alert detection and performance. PATROL detects basic printer problems automatically through automatic discovery of printers connected to the R/3 system and continuous monitoring of the printers themselves and the jobs queued to the printers (see Figure 12-2).

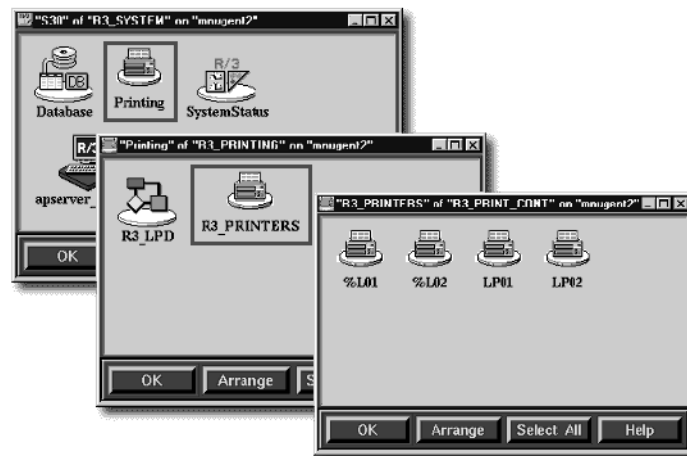


Figure 12-2 PATROL KM for SAP R/3 printer

The application layer of the PATROL KM for R/3 uses both direct R/3 access and operating system information. The top monitored statistics at the application layer for the R/3 Knowledge Module include:

- CCMS alerts
- Average response time
- CPU and memory top 10 hogs
- User activity
- Print queue length
- Buffer quality
- Buffer storage utilization
- Work-process response times
- Work-process event frequency
- Syslog activity
- Dispatcher queue capacities
- Transaction elapsed times
- Enqueue errors

The PATROL KM for R/3 incorporates operating system, database, and application level monitoring. The database layer is an important part of R/3 availability, and the Knowledge Module provides some significant monitoring. Be sure to monitor the PSAPBTAB, PSAPSTAB, PSAPCLU, PSAPPOOL, and PSAPPROT tablespaces, particularly their sizes, segments, and extents. This tablespace monitoring is provided by the database layer of the Knowledge Module. The top statistics for R/3 database layer monitoring include:

- Tablespace freespace
- Tablespace max extents
- Free space deficit
- Archive destination freespace
- Cache hit ratio
- Dictionary cache hit ratio
- Open cursors
- Transaction rate
- Processes used
- Active transactions
- Backup alarms

The architecture used by the PATROL KM for R/3 is interesting of its own accord. The KM is an example of an advanced multi-agent architecture that performs mapping of a multi-host physical structure into an application-level logical view. This is performed automatically by the KM and is shown to the user as a single icon per R/3 system that contains all the physical components that comprise this R/3 system, even if they are distributed across multiple hosts.

Another interesting aspect of the architecture is the manner of data collection from R/3 to perform the monitoring and performance analysis. PATROL does not use the R/3 MIB, because the MIB offers only basic information that is inadequate for the analysis required. Instead,

PATROL uses a data collection interface in conjunction with a set of ABAP functions that perform the data extraction. The API into R/3 is a critical part of this method. PATROL and SAP are actively working to improve the next version of the R/3 API to offer even more powerful management.

The primary benefit from the PATROL KM for R/3 is increased availability through early notification about problems. CCMS is passively requiring continual user attention to manually check all statuses, whereas PATROL is proactive and brings problems to your attention. The automation of what is predominantly a visual monitoring situation via CCMS frees the Basis administrator to spend less time watching the system and more time in other valuable activities. Rather than clicking through all the CCMS screens manually, the application management software can be left to do this, freeing the administrator for other work. The basic task of watching the system can even be delegated to a help desk or operator personnel, requiring Basis administrator attention only when PATROL detects a problem. In addition to these benefits from automation, the monitoring of the environment surrounding R/3 also catches more problems. The fact that CCMS is inside R/3 also means that problems with R/3 will often degrade CCMS as well. PATROL works on the outside, analyzing R/3 and all the RDBMS and operating system components. The monitoring of performance data and response time through all the application layers makes performance tuning possible with more detailed information. The archival of performance and system data also makes possible the use of analysis for system level reporting or capacity planning.

12.3 PeopleSoft Applications

Founded in 1987, PeopleSoft started with its human resources management products, as the company name suggests. Since then, PeopleSoft has become a major ERP applications player, with a suite of application components for various business units. In terms of revenue, in 1998, it was second only to SAP in the ERP applications market. With the release of PeopleSoft 7, the major areas of business processing coverage include:

- Human Resources
- Manufacturing
- Distribution
- Financials
- Retail
- Health care
- Federal Government
- Public Sector
- Higher Education

The release of PeopleSoft 7 adds various modules in different areas. For example, this version adds Time and Labor and Pension Administration to PeopleSoft HRMS. The acquisition of

Red Pepper Software by PeopleSoft added advanced supply-chain management technology to the portfolio.

PeopleSoft positions its ERP application suite at three levels of corporate size. The Enterprise Solutions are for corporations with large requirements for BPR and large changes. The Industry Solutions aim at particular vertical industries. The PeopleSoft Select initiative aims at the middle market segment of corporations wanting to make smaller business changes for a rapid return.

PeopleSoft provides its own Professional Services organization and also has the PeopleSoft Partners Program, through which third parties can provide the same services. The usual issues are covered, including BPR, implementation, and development for customization or integration. Development issues are handled by the use of PeopleSoft Tools components that provide RAD and reporting tools that abstract the lower level details. PeopleSoft Workflow is an open architecture that permits the adaptation of PeopleSoft applications to the requirements of business practices.

12.3.1 Managing PeopleSoft Applications

The need for managing PeopleSoft applications is clearly driven first to improve availability and second to improve performance. The PATROL KM for PeopleSoft Applications delivers both availability and performance improvements. It does so through the continuous monitoring of several key components of a PeopleSoft installation through all the different application layers:

- PeopleSoft application
- Database server
- TUXEDO transaction processing server
- Operating system

PATROL monitors all these components for downtime and offers both early notification and automated restart of these components. It does so using not only the PATROL KM for PeopleSoft Applications, but also via other Knowledge Modules—such as operating systems and RDBMS Knowledge Modules. Depending on the scenario, you may decide to use the PATROL KM for PeopleSoft Applications alone, or in conjunction with others. For example, your decision may differ depending on whether a two-tier or three-tier PeopleSoft architecture is in use.

By monitoring all the different levels, the PATROL KM for PeopleSoft Applications can detect problems both in PeopleSoft and in the surrounding environment (see Figure 12-3). The platform coverage also satisfies the needs of PeopleSoft monitoring with coverage across UNIX and Windows NT, in addition to the DBMS and TUXEDO monitoring components. Some of the key points in monitoring PeopleSoft applications include:

- User monitoring
- Response time monitoring

- Transaction performance and throughput
- Queue monitoring
- Batch process monitoring
- Process scheduler and job monitoring
- Table and tablespace monitoring

The Knowledge Module monitors all PeopleSoft users for important situations, and can diagnose which users are hogging resources. Ghost users with orphaned database connections are also detected.

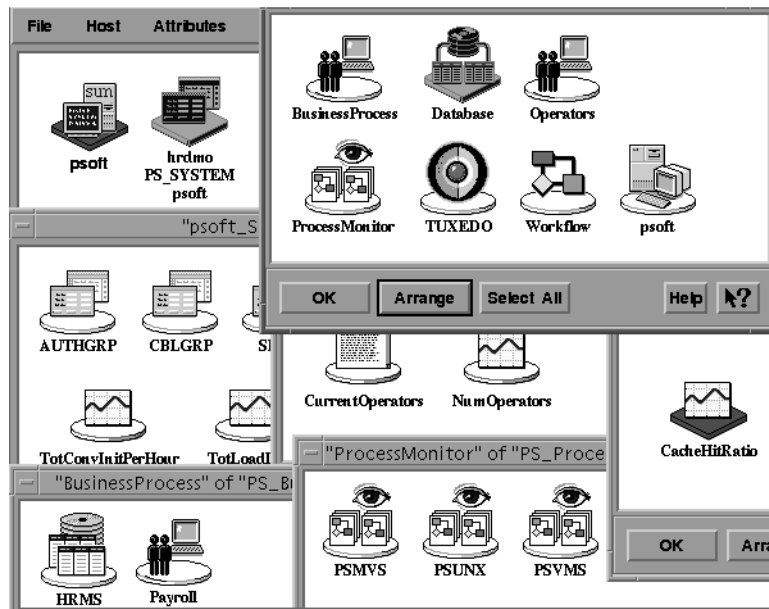


Figure 12-3 PATROL Knowledge Module for PeopleSoft Applications

An interesting note is how the application-level nature of the monitoring shines through in various features. For example, monitoring of aspects such as Pay Calculation and Pay Confirm occurs through the underlying mechanism of database table monitoring. Batch process monitoring becomes monitoring of application-level processes such as paysheet, time and labor, accrual, benefits, and COBRA.

This application-level management gives all the benefits to PeopleSoft administrators. Availability is improved through early problem detection and automated notification or even correction. Performance is monitored across all the levels to permit informed tuning. All the data measured by PATROL is archived, offering the opportunity for PeopleSoft-specific capacity planning analysis.

12.4 Baan Applications

Baan is an emerging major challenger in the independent ERP application vendors. Baan was founded in 1978 in the Netherlands and is now a major ERP vendor with revenue in the hundreds of millions of dollars. Baan completes the group often called the “top three” major independent ERP vendors.

The current product suite is the Baan IV release, which supersedes the previous releases under the name TRITON. Baan Applications is a suite of ERP applications that covers all the major horizontal business requirements and provides support for some of the vertical industries. Some of the major modules include:

- Manufacturing
- Distribution
- Financials
- Transportation
- Service
- Project
- Orgware

Some of the Baan components address horizontal management issues and others are aimed at particular verticals. Other components are more internally motivated to aid in the use of Baan applications.

The Orgware module is aimed at simplification of implementation and customization issues. It includes the Dynamic Enterprise Modeling (DEM) tools to help map enterprise business requirements to Baan components. Other components of Orgware include Enterprise Modeler, Enterprise Reference Models, Enterprise Performance Manager, and Enterprise Implementer.

The basis of Baan applications is a set of portable tools called Baan Tools, which provide an application development environment and its tools. These offer a portable shell called B-Shell, which incorporates a 4GL for application logic. This permits application programmers to write code that is abstracted from the systems layer.

Baan IV completes the integration of Baan Applications with the Hyperion finance and accounting package. This permits consolidation, budget, and reporting functionality from Baan through links to Hyperion.

Another recent development with Baan is the new Baan Synchronization module. Baan Synchronization provides supply-chain management for manufacturing and distribution. It synchronizes the actions of suppliers, manufacturers, and distributors to achieve faster manufacturing and turnaround cycles.

12.4.1 Managing Baan Applications

The growing importance of Baan applications to the business make managing them critical. The PATROL Knowledge Module for Baan Applications (Figure 12-4) is the PATROL solution.

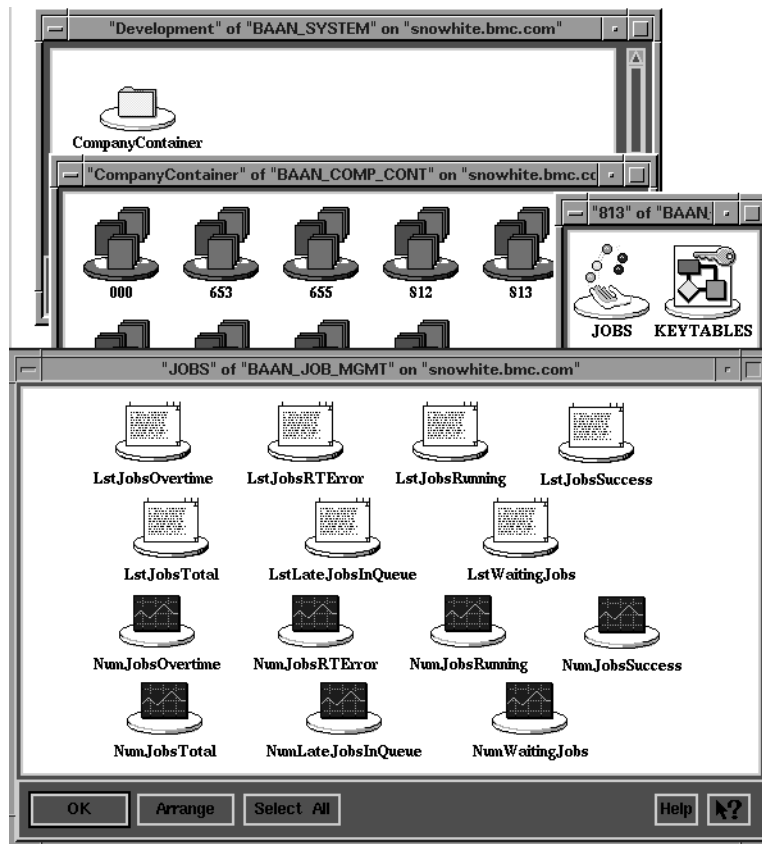


Figure 12-4 PATROL Knowledge Module for Baan Applications

This Knowledge Module provides cross-layer monitoring of all the major components of the Baan system, including:

- Baan application layer
- Database layer
- Operating system layer

Monitoring the Baan application layer includes functionality such as mapping database tables to Baan companies and monitoring Baan users. In fact, some of the key application-level monitoring facilities are:

- Server/daemon monitoring—license, database, printer
- Printer monitoring
- User monitoring
- Job monitoring
- License usage monitoring
- Shared memory monitoring

A lot of this monitoring is occurring via analysis of the underlying database. The database layer monitoring is provided for both Oracle and the proprietary Tbase database inherited from the TRITON product line. The PATROL KM for Baan Applications provides basic database monitoring features that can be further enhanced through the use of a separate RDBMS Knowledge Module such as the PATROL KM for Oracle.

The license monitoring features are notable. The PATROL KM for Baan Applications monitors seats licensed, registered, available, and in use. This is a powerful feature with respect to cost-effective usage of Baan applications. It also stores the historical data needed to perform capacity planning analysis through analysis of users, performance loads, and licensing counts.

The KM presents all the data in a neat application-centric view. The top view shows the application view and can be drilled into for analysis of components such as the databases, users, and printers.

12.5 Oracle Applications

Oracle has leveraged its penetration into the database market to also provide ERP applications that run on top of their databases. Like the other independent ERP vendors, this suite provides a set of business operations management components. It addresses numerous horizontal cross-industry business issues and some vertical industry-specific solutions. Some of the components of Oracle Applications include:

- Consumer Packaged Goods
- Energy
- Environmental Health and Safety
- Government
- Higher Education
- Pharmaceuticals
- Financial Services
- Automotive

The functional modules include the following areas:

- Human resources
- Financials

- Manufacturing and Distribution
- Workflow processes

All these areas are applications that run on top of the Oracle database technology. This arrangement is a combination of application logic on top of the database for storage, as with many ERP application vendors, but with the twist that the application and database vendor are the same company.

Oracle has made a major impact into the ERP industry with its Oracle Applications solution. Oracle has a users group for Oracle Applications that is separate from its database users group. Also, "PROFIT magazine" deals mostly with Oracle Applications.

12.5.1 Managing Oracle Applications

The management of Oracle Applications aims to increase availability and at the same time decrease the load on administrators. Availability improvement occurs through PATROL's early detection of problems through the PATROL KM for Oracle Applications (see Figure 12-5). Administrator workload is also eased through automated monitoring of the system, and through diagnostic aids when a problem is brought to the administrator's attention. Some of the key features of the PATROL KM for Oracle Applications include:

- Availability monitoring of key components
- Operating system level monitoring
- User monitoring
- Concurrent manager monitoring
- Process monitoring
- Job monitoring
- Transaction throughput monitoring

The basic feature is the monitoring of availability of all the components underlying the Oracle Applications infrastructure. PATROL automatically detects availability problems with the Concurrent Managers and the SQL*Listener process, not to mention monitoring of the underlying Oracle database server.

Operating system metrics are also applied directly. The CPU is monitored with regard to CPU utilization and memory usage. File systems are monitored to detect storage space problems. Swap space is also monitored for these reasons.

Monitoring of many of the underlying infrastructure components beneath the overall application occurs as listed above. Users are monitored to detect heavy resource usage. The count of users is also valuable for license validation purposes. Concurrent managers are monitored for their availability and for a total count. Processes are also counted and monitored to detect status problems. Long-running processes are detected and brought to administrator attention through whatever notification method is used.

Many of the features operate at the even higher application level. Specific monitoring of particular application modules occurs. Activity reports for financial modules are run, including general ledger, accounts receivable, and accounts payable. Transactions are monitored through the system in the particular modules.

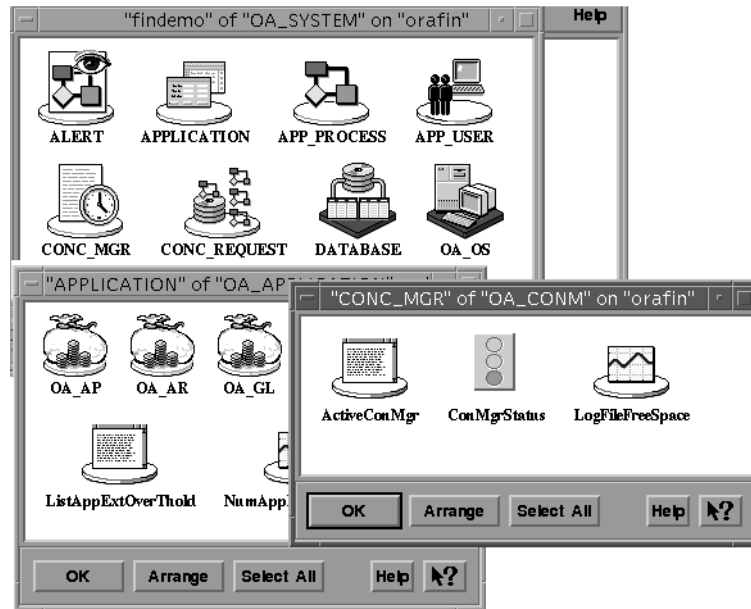


Figure 12-5 PATROL Knowledge Module for Oracle Applications

The Knowledge Module provides full coverage of the application modules across UNIX and Windows NT, and across Oracle versions. Although the first release of the Knowledge Module for monitoring Oracle Applications was focused on Oracle Financials, the current releases focus on the entire suite of Oracle applications. The PATROL KM for Oracle Applications can be used both with and without the PATROL KM for Oracle. They are complementary Knowledge Modules, and their use in combination provides fuller database server monitoring. However, the PATROL KM for Oracle Applications' monitoring capability is most important for the application for basic availability and space monitoring, such as free space and extents.

12.6 Internet Servers

The external internet server is a major corporate asset in these modern interconnected times. Not only is it a marketing asset, but it can also be a direct stream of revenue from online transactions with customers, suppliers, and distributors. Almost equally important is the smooth running of the intranet sites as organizations become more dependent on web-based resources for collaborative work and information sharing. Thus the internet and intranet servers are mission-critical assets and clear candidates for application management.

PATROL supports this importance with the monitoring and management provided in the PATROL KM for Internet Servers (see Figure 12-6). Continuous monitoring detects problems early and ensures that outages are avoided or minimized. Some of the key components monitored by the PATROL KM for Internet Servers include:

- Web server
- Proxy server
- Mail server
- FTP server
- News server

Multiple servers on the same host or different hosts are also handled for environments that use redundant servers. Local host monitoring occurs through the local agent, and remote monitoring occurs via the internet protocols. This setup is supported across all UNIX and Windows NT platforms.



Figure 12-6 PATROL Knowledge Module for Internet Servers

The PATROL KM for Internet Servers supports the Common Log Format (CLF) for web and proxy servers. Its power can therefore be extended to almost any internet server software available. Currently, the list of web servers that are supported includes:

- Netscape
- Cern
- NCSA
- Lotus Domino
- Microsoft IIS
- Apache

The availability and performance monitoring occurs from both the inside and the outside of the server. The inside analysis includes CPU usage tracking of the operating system processes and the analysis of the web server log files. The outside analysis means that the KM tests the server at regular intervals by sending it a request. This allows the KM to track internal estimates of performance and also simulate the real performance levels seen by external users making requests. These achieve continuous monitoring of web server availability and detection of significant performance degradation. Also, warnings occur for integrity failures such as bad links or requests for documents that were not found or not authorized.



Figure 12-7 PATROL KM for Internet Servers' availability monitoring parameters

Many parameters record both the performance of the web servers and the loads under which they are operating (see Figure 12-7). Performance is measured by parameters such as the response time, CPU usage, and memory usage. Load parameters record the number of requests, size of requests, and other workload statistics. By comparing these against historical maximums, the KM can detect when the web server is approaching saturation.

Analysis of the server log files is not limited to performance metrics. Also, administrative reports and iconic views show the hit rates at a granularity down to the individual web page. Mapping IP addresses to hostnames also helps simplify audits and access tracking.

The PATROL KM for Internet Servers also has proxy remote monitoring features. These are useful for monitoring web servers on Linux, FreeBSD, BSDI, or other platforms without native agents. Remote monitoring uses HTTP access and permits much of the performance, response time, and availability monitoring. Further remote capabilities are available if the log files are available via NFS mounting. Although full functionality is not available, the proxy features are still worthwhile to get the benefits of continuous monitoring.

12.7 Lotus Notes and Lotus Domino

Lotus Notes spans the groupware and middleware boundary. It has features like messaging and multi-user collaborative applications. This flexibility and multiple use makes Notes a mission-critical application in many sites.

The PATROL KM for Lotus Notes (Figure 12-8) monitors the Notes server and databases. Lotus Domino Internet server can also be monitored in collaboration with the PATROL KM for Internet Servers. They offer continuous monitoring of all the key aspects of Notes.

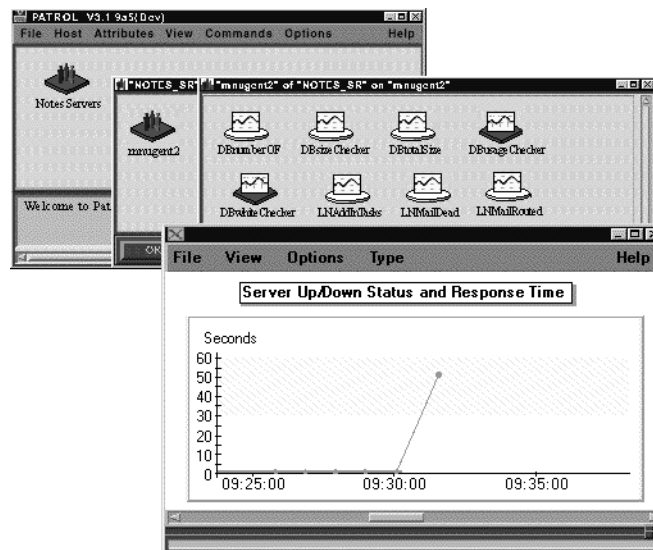


Figure 12-8 PATROL Knowledge Module for Lotus Notes

Multiple versions of Notes are monitored by the Knowledge Module. Separate add-in tasks exist for each major version, allowing the monitoring of older platforms such as the 3.3x, while still exploiting new features of the API found in Notes 4.0 and 4.5. Platform coverage includes UNIX, OS/2, and Windows NT.

The NOTES_SR application class monitors the server, continuously checking the availability and connectivity of the Notes server. The error log is analyzed to check for internally detected problems about which the administrator needs to be alerted. Monitoring of the key server tasks occurs, with warnings when one is missing and the capability to automatically restart it. Menu commands in this application class offer reports on a variety of measured attributes.

The NOTES_DB application class monitors the databases on the Notes server, allowing detailed analysis of user data and other databases. One special database that is monitored is the MAIL.BOX database, which is analyzed for dead or undelivered mail warnings.

Database Management

The management of databases is a critical part of application management because all the applications depend on data storage mechanisms. The management of databases becomes part of the management of the application stack for commercial and in-house applications. The databases are also sophisticated entities in their own right that warrant intelligent management and monitoring.

PATROL offers deep management of almost every database on the commercial market through its DBMS Knowledge Module suite. Currently, Knowledge Modules are available for various databases including:

- Oracle—including Oracle Symmetric Replication and Oracle Parallel Server
- Informix
- Sybase—including Sybase Replication Server and Sybase SQL Server
- Microsoft SQL Server
- Rdb
- DB2—Universal Database and OS/390
- CA-OpenIngres and CA-Ingres
- PROGRESS
- ADABAS

The coverage of operating system platforms and database versions is also very extensive. In addition, other databases are likely to be covered through the PATROL solutions available through the PATROL Developer Network (PDN). The basic PATROL product for these databases is the corresponding Knowledge Module. Major databases also have PATROL products in

the PATROL DB and PATROL Recovery Manager suites for advanced administrative functionality such as schema management or backup and recovery.

The database Knowledge Module is the base product in the PATROL suite for each database. Much commonality exists in the functionality provided by the various database Knowledge Modules. Typically, a database Knowledge Module provides several different types of functionality, including:

- Server daemon process monitoring
- Server error log scanning
- Parameters recording availability, performance, space, and capacity data
- Predefined alert thresholds to warn about impending database problems
- Recovery actions to gather more data for diagnosis and postmortem analysis
- SQL capture facilities for deep diagnosis
- Recovery actions to execute corrections (usually off by default)
- Ad hoc reports via menu commands
- Administrative commands via menu commands

The monitoring of availability, performance, and capacity/workload on the database is achieved via KM parameters. Menu commands provide administrative tools for tasks such as configuration, tuning, ad-hoc SQL commands, and launching DBMS-specific tools. Recovery actions provide both corrective actions and diagnostic aids such as reports and annotated data points. Corrective actions are often configured off and turned on as the administrator becomes more comfortable with the agent's level of intelligence.

Not all features of database management are provided directly inside the Knowledge Modules. Other features are provided in the basic PATROL agents and consoles and thereby apply to all Knowledge Modules without needing any special customizations. Some of these infrastructure services provided for all KMs include:

- Console iconic, outline, and text event views
- PATROLVIEW integrations with third-party consoles and help desks
- PATROLWATCH web browser views
- Notification via pager, email, or SNMP traps
- Integration via API, CLI, or SNMP
- Historical archiving of data for reporting and capacity planning

Notification is provided by agent services or add-on components that provide the service to all Knowledge Modules. Similarly, all Knowledge Modules work identically against the PATROL infrastructure including all PATROLVIEW integrations (Tivoli TME, HP OpenView, CA-Unicenter TNG, Remedy ARS, etc.), the PATROL API, the PATROL Console, and the PATROL CLI. So the actual Knowledge Module component provides only database-specific

intelligence, and other useful services are provided orthogonally to the Knowledge Modules that run in each agent.

13.1 Oracle

Oracle databases often serve as mission-critical components in an enterprise. The databases deserve monitoring in their own right and are often used as the server backend for ERP or in-house applications. This critical usage raises the need for monitoring of availability and performance characteristics.

The PATROL KM for Oracle is one of the premier PATROL Knowledge Module products (Figure 13-1). The Oracle components covered include the database server, Oracle Symmetric Replication, and Oracle Parallel Server. Oracle Applications are also monitored; see “Oracle Applications” in Chapter 12. The base PATROL KM for Oracle monitors the databases and the server. Separate KMs for Symmetric Replication and Parallel Server are add-on components to this basic Knowledge Module.

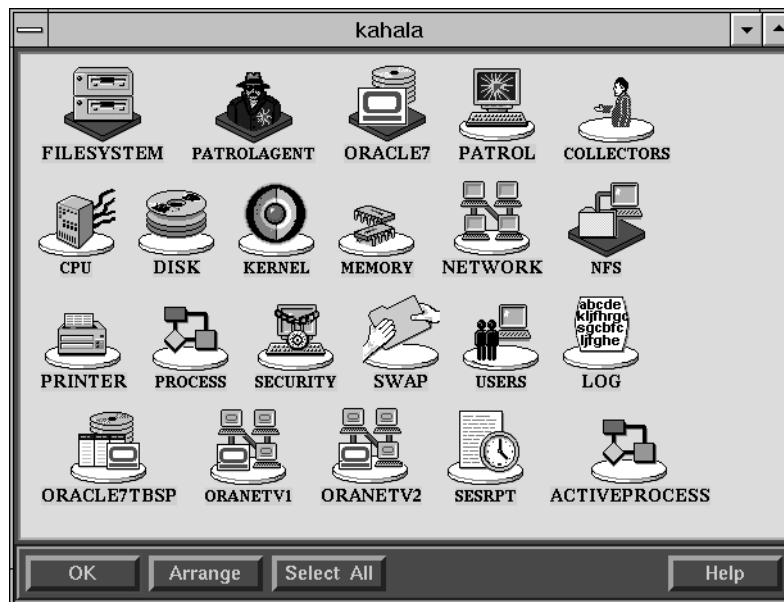


Figure 13-1 Host-level view of PATROL KM for Oracle with PATROL KM for Unix

The PATROL KM for Oracle offers a great many features in its monitoring of the server and its databases. Some of the key areas of monitoring functionality offered include:

- Database statistics
- Tablespace statistics
- User statistics

- Multi-threaded server statistics
- Transaction statistics
- SQL*Net statistics (versions 1 and 2)

Statistics cover a breadth of important areas including availability, performance, space usage, and capacity. Availability monitoring is the most crucial aspect of the Oracle KM. A number of parameters and discovery aspects detect availability problems. A downed database is detected in discovery via monitoring of the Oracle operating system processes and the SGA file. This distinguishes a crashed database from a normal shutdown to avoid false alarms. Of course, once the database has crashed is often too late, so various parameters in the KMs attempt to find problems earlier, particularly the parameters that measure space usage of the database. For example, monitoring tablespace extents and available space detects when the database may fail to allocate enough space for its next extent. Messages to the Oracle alert log are also analyzed to detect critical messages and raise a PATROL alert. Other parameters cover archive free space and background dump usage.

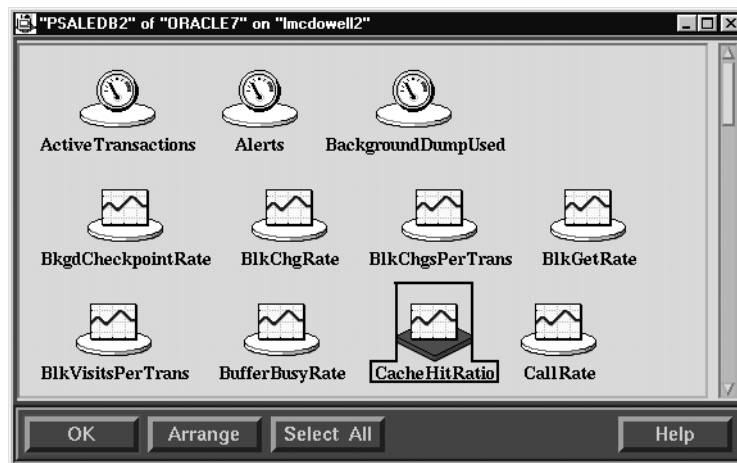


Figure 13-2 PATROL KM for Oracle parameters (subset)

Performance analysis is another key area of KM functionality. In fact, the majority of the parameters are aimed at performance measurement. Some of the performance monitoring parameters include cache hit ratio, call rate, chained rows, buffer busy rate, in-memory sorts, and many others. If Oracle has a cache hit ratio or other rate to measure, a parameter for it is probably in the PATROL KM for Oracle.

Diagnosis of performance problems is facilitated by the real-time capture of executing SQL statements. These are available immediately and also historically for postmortem performance analysis reports. This feature allows drill-down to see the SQL and its initiating user for the cause of a performance spike. These spikes are detected via the performance parameters and

the monitoring of user session performance. Another application of this feature is in the analysis of the performance impacts on new applications that use the database.

Space usage is monitored by many parameters in the Oracle KM. Areas covered include free space, tablespace extents, archive space, and others. Also, menu commands to run reports provide more detailed data than the parameters. The add-on companion product PATROL DB-Stats offers even more detailed space usage analysis.

Capacity and workload parameters are less critical in day-to-day operations but serve a useful purpose in historical analysis and trending. Capacity planning is one key activity that requires raw data. The Oracle KM records a variety of workload parameters including user sessions, transactions, and locks. In conjunction with performance parameters, these give a good profile of the strain of the database under current load, and further analysis of the data can help to predict future needs.

The PATROL KM for Oracle is broadly supported across a variety of environments. Some versions of the PATROL KM for Oracle support Oracle 6 through Oracle8. Some KM features work across all versions, and others are specific to new features of Oracle8. For example, special monitoring of new structures, such as partitioned tables, partitioned indexes, and index-only tables is in Oracle8.

The operating system platforms covered by the KM are basically the same as the agent platforms and include many UNIX versions, Windows NT, OS/2, OpenVMS, and MVS. Other Oracle platforms can be covered through the SQL*Net-based remote database monitoring, although this proxy monitoring is necessarily less powerful than direct monitoring with a local autonomous agent. Also, various add-on modules exist, including PATROL KMs for Oracle Parallel Server and Oracle Symmetric Replication, PATROL DB-Stats, and many other administrative products in the PATROL DB suite.

13.2 Sybase

PATROL supports the Sybase RDBMS through several Knowledge Modules and various PATROL administrative complementary tools. The Knowledge Modules are for monitoring Sybase SQL Server, Sybase Replication Server, and Sybase SQL Server Monitor. The PATROL KM for Sybase Replication Server covers both the Replication Server and the Log Transfer Manager (LTM). Various other Sybase tools for schema management, database administration, or backup and recovery are offered in the PATROL DB and PATROL Recovery Manager series.

The PATROL KM for Sybase SQL Server supports releases 4, 10, and 11 of Sybase. Differences occur in the handling of the different Sybase releases, with different KM functionality for different releases. The KM's support of release 4 is managed through a separate application class, whereas 10 and 11 share much of the same application classes.

Availability of the SQL Server is continuously monitored through process monitoring, error log message analysis, and internal connection status. The server and the databases carried by the server are monitored. Suspect or unusable databases are also brought to the attention of

the administrator. I/O errors are tracked and mapped to disks or filesystems in order to diagnose potential hardware problems that may adversely affect the data.

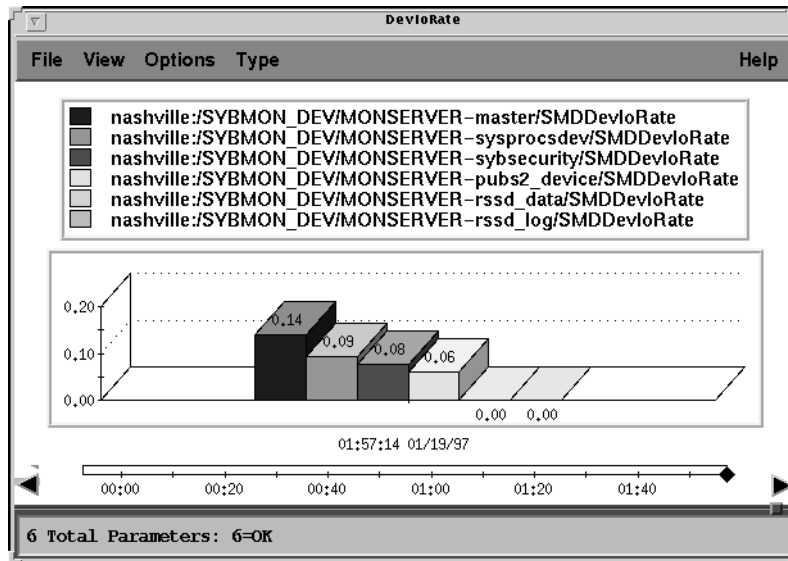


Figure 13-3 PATROL KM for Sybase SQL Server Parameters

Performance monitoring of the SQL Server occurs at various levels (see Figure 13-3). Firstly, numerous parameters record statistics about queues, resource contention, and response rates. Long-running transactions are detected that may indicate the need for further tuning. Idle processes are detected and can be removed as they are wasting resources. Space monitoring also detects conditions in which storage needs may soon exceed available storage.

The second form of performance analysis is the capture of SQL statements as they transition through the database (see Figure 13-4). Depending on the configuration and version of the SQL Server, this SQL is gathered either via the API or via an internal access method. These two features of the KM are named SQL Viewer and SQL Snapshot. Both gather and store the SQL scripts for immediate or later analysis, allowing the administrator to determine the user and originating node of an offending script that has caused a workload spike. Blocked processes are another useful area where SQL analysis is possible.

KM menu commands offer various administrative actions against the SQL Server, including both reports and commands that change the server or its databases. Reports include server configuration, memory allocation, space reports, resource hogs, uptime, and many more. Actions against the server include server start or shutdown, backup server start or shutdown, and the launching of SQL Server tools themselves.

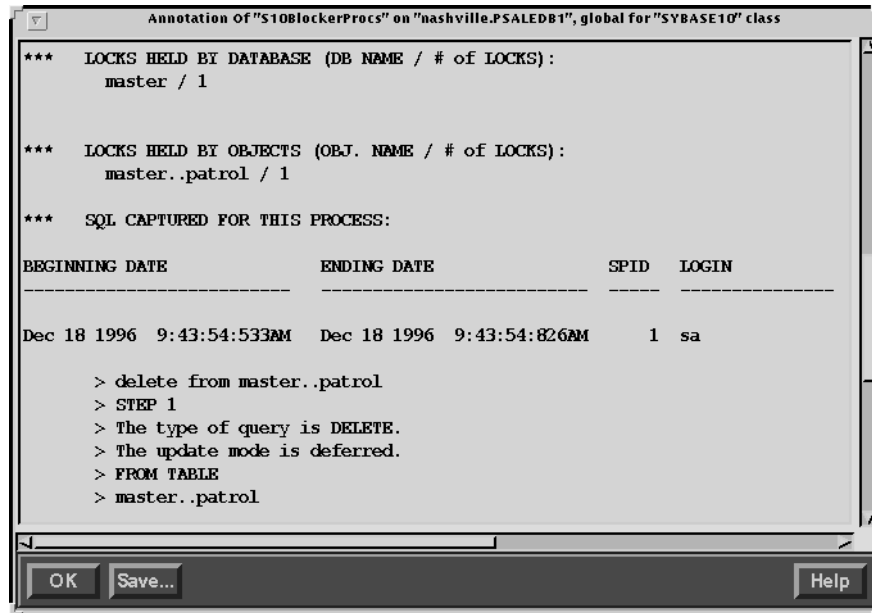


Figure 13-4 PATROL KM for Sybase annotation text showing captured SQL

13.3 Informix

The Informix RDBMS is supported by a range of PATROL tools. The basic database management tool is the PATROL KM for Informix, which monitors availability, performance, and load issues. A PATROL KM for INFORMIX XPO is also available. The PATROL DB series of products also offers advanced administration, such as backup and recovery, change management, and configuration, but these other products are not covered in this section.

The PATROL KM for Informix supports the most recent and several older versions of Informix, with coverage of Informix-OnLine versions 4 and 5 and Informix-Online Dynamic Server versions 6 and 7. Some KM features are the same across multiple versions, whereas other KM features can be exploited only in the latest Informix versions.

Remote administration capabilities are provided via KM menu commands. A variety of reports are available about the database server features, such as locks, users, queues, and configuration settings. You also have access to administrative tools such as tbcheck/oncheck, ad-hoc SQL commands, and Onmonitor/Tbmonitor. Some basic administrative facilities are provided, such as to load or unload tables, and to start up or shut down the database server (if the user has this privilege).

Problem detection and continuous monitoring of the database server is provided in the PATROL KM for Informix. 24x7 monitoring of the Informix server occurs via its external operating system interfaces and also via internal analysis. The Informix-OnLine log is analyzed for

critical errors when Informix detects internal problems. Raising these errors to administration attention helps with early correction and downtime prevention. Another interesting availability-related parameter is the monitoring of the number of days since the last backup. Keeping an eye on this statistic won't help keep the database up longer, but it will hopefully make recovery more effective and improve overall availability that way.

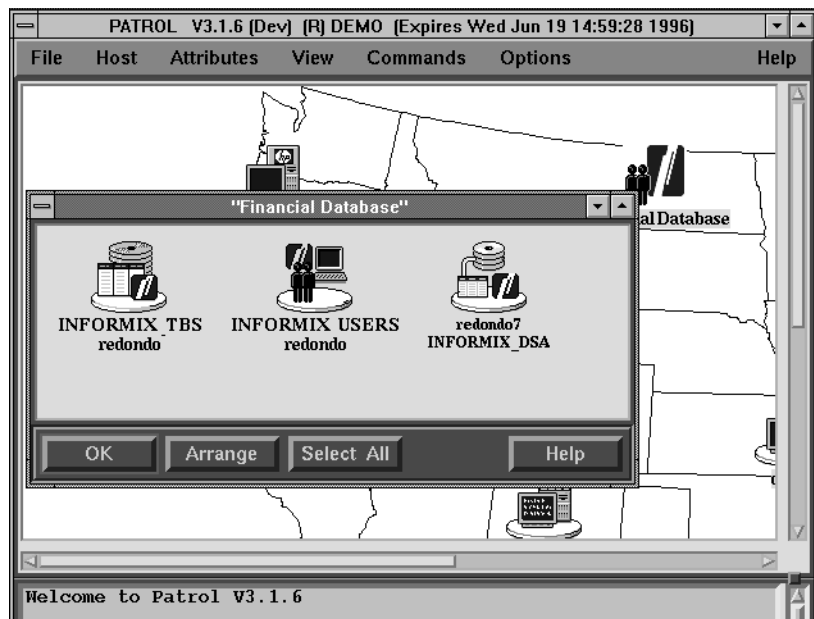


Figure 13-5 PATROL KM for Informix host level view

Space monitoring also improves database availability, since space usage problems are a common cause of downtime. The most critical aspects with regard to availability are monitoring of tablespace growth and that of the logical log. Analysis of the efficiency of access to this space is also useful for performance improvement. Ensuring adequate space and distributing data correctly to the available storage areas is important for performance tuning.

Performance monitoring is a key PATROL KM for Informix feature and is provided through a large number of parameters. Since the majority of performance problems are due to resource contention and associated queues, many Informix KM parameters analyze the wait times or queue lengths. For example, you can monitor lock waits, latch waits, deadlocks, buffer waits, and long transactions. Other forms of performance monitoring analyze basic performance aspects such as caches, memory efficiency, table scans, hot disk areas, and buffers. Some of the parameters on this long list include sequential table scans, cache hit rates, read-ahead pages, page-cleaner efficiency, and shared-memory buffers. Measuring these statistics helps the administrator tune the database to optimum efficiency.



Figure 13-6 PATROL KM for Informix user monitoring view

One powerful performance diagnosis tool is the real-time capturing of SQL commands for each user (see Figure 13-6). This allows the administrator to diagnose the cause of sudden performance spikes. Capturing SQL is particularly useful in a dynamic environment where end users have the power to build front-ends via PowerBuilder or Visual Basic that launch SQL scripts against the database. The SQL capture facility narrows it down to the culprit. Since the SQL is available both in real-time and historically via annotated data points, diagnosis can take place either immediately during the problem or in a postmortem analysis session. This can be hours, days, or even weeks later, which is useful for diagnosis of transient or intermittent performance problems.

13.4 DB2

The IBM DB2 RDBMS is one of the longest-standing database systems in the world, containing the mission-critical data in many enterprises. PATROL supports that data through two Knowledge Module products: one for DB2 Universal Database (UDB) and another for DB2 for OS/390 mainframes.

The PATROL KM for DB2 offers monitoring of several classes of DB2 attributes, including UDB (see Figure 13-7). Naturally, the server itself is monitored via availability and performance statistics. The resident databases are separately monitored with individual statistics and reports. Agents are also monitored via separated object creation. DB2 applications are also monitored individually.

A key feature of the KM is the SQL Viewer. As with many PATROL database KMs, this is a facility to capture SQL text as it executes in the server. This allows real-time and historical archiving of SQL statements. This helps diagnose sudden performance spikes by mapping it to the current SQL activity.



Figure 13-7 PATROL Knowledge Module for DB2 Universal Database

The PATROL KM for DB2 for OS/390 offers similar functionality (see Figure 13-8). It provides monitoring classes for DB2 systems, DB2 subsystems, DB2 applications, page sets, and buffer pools. The DB2 system class monitors CPU utilization, I/O waits, lock waits, and other performance statistics. Reports cover areas such as messages, buffer pools, stored procedures, logs, and many others. The DB2 application class offers similar performance measurement but specific to that application.

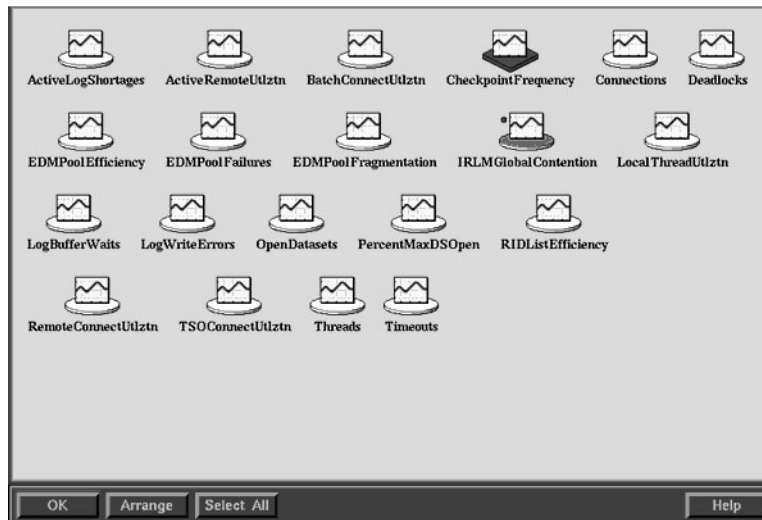


Figure 13-8 PATROL KM for DB2 for OS/390 parameter view

Implementing PATROL

Implementation is the art of converting theory into practice. This is the actual use of software to achieve the rewards of application management. It encompasses issues such as deployment, distribution, configuration, policies, procedures, and documentation.

In this chapter, I examine both the theoretical aspects of application management implementation, and the practical decisions required when doing that with PATROL. The basic issues with PATROL implementation are also covered in detail by the “PATROL Implementation Basics Manual,” which is part of the standard PATROL documentation available from BMC Software. More advanced implementation methodologies are typically provided by implementation specialists, including those from BMC Software and partners in the BMC Authorized Service Provider (ASP) program.

14.1 Goals of an Implementation

The basic goal of an implementation is to achieve the advantages of application management as quickly as possible. These advantages are quite basic to the needs of the IT management. Let us try to categorize the different basic needs as they relate to application management. The basic aims include:

- Detection
- Correction
- Notification
- Diagnosis
- Measurement
- Prediction

The detection of problems with the database or application is a clear requirement for application management. PATROL achieves this goal by measured metrics with thresholds, and also via more complicated detection of inter-dependent problems. Parameters with thresholds allow the alerting on basic problems, such as performance or load above which attention is needed. The more detailed alert generation involves analyzing the components of the application to detect anomalous situations. These situations might be self-detected by the application itself or specially detected by the Knowledge Module based on an understanding of the application's architecture. Self-detected application errors involve issues such as monitoring its own error logs and event interfaces and bringing these various methods of error reporting back into the PATROL standard context.

The different types of problems that the application management solution should detect include those related to availability, performance, and capacity. Availability problems are the most critical ones that require detection. The unavailability of an application must be very quickly detected and resolved. Performance problems are also critical because a poorly performing application can cause lost productivity and often degrade into lack of availability. Capacity data such as load and responsiveness is desirable for reports such as capacity planning.

Once the application management solution has detected a problem, what should it do about it? The most obvious idea is to try to correct it. This is the ultimate in reducing resolution times. PATROL offers this via recovery actions in the Knowledge Modules. These respond with built-in corrective commands, either factory pre-set or user-customized, to resolve the problems for which alerts are generated.

The alternative to auto-correction of errors is to tell someone about it, either because to automatically determine the right corrective course of action is not possible, or because the desired functionality is not to use automated corrections. You may also want to send notification that a problem was auto-corrected and no action is needed. Notification is the method by which people are alerted to the detected problem. Notification methods include screen alerts, pager, email, or SNMP traps to some other notifying tool, telephony, or other human interfaces. PATROL supports all of these, either as part of the product or via integration with third-party tools. Notification can include alerts about problems and messages to indicate that the problem has gone away or been resolved.

Diagnosis is the next level beyond notification. After alerting the user to a problem via some notification method, the application management tool should help solve the problem. This involves information to help diagnose the problem and tools to help correct it. PATROL offers diagnosis information via expert advice in events and online help text. The current situation can be examined via the real-time dashboard iconic view of the status of databases and applications, and the menu commands that run reports to provide more detail about the current situation. Also, some information that is critical for post-mortem diagnosis is found in the parameter history and the annotated data points. These contain information about the state of affairs at the time of an alert, such as the SQL scripts that were executing then. The corrective tools include a

variety of application-specific administrative commands that are in the menu commands of Knowledge Modules.

Measurement takes application management to the next level of maturity. The only way to know whether the application management methodology is effective is to measure its effectiveness. This involves measuring service levels, problem counts, and problem resolution times.

Prediction is the next level, where these measurements are projected into the future to determine what needs to be done to handle future capacity. PATROL provides all the raw data for this measurement and prediction task in its historical databases of measured values and generated events. Creation of reports from this data can involve scripts to analyze the data or integrations with other reporting tools by uploading the PATROL data.

14.2 Implementation Planning

An important aspect of implementation is getting the returns with as little time, cost, and effort as possible. In this context, application management is just like any other systems project. PATROL is a good vehicle to reach these goals in that it allows rapid implementation. However, PATROL's successful implementation at a technical level requires that the application management project itself is well managed. The concerns here are similar to any other project:

- Defining the scope of the project and who is affected
- Ensuring that adequate resources are assigned to the project
- Assigning staff to tasks and responsibilities
- Adequate planning for the project
- Adequate documentation about the project's goals and methodology
- Justifying its cost to upper management
- Running a successful pilot plan to validate expectations and estimates
- Training of all personnel planned according to their job roles
- Establishing credibility to rise above inter-departmental politics to a common goal

Good project management should address all these goals. The place of this book is not to preach about the value and methodology of good project management. This is certainly a topic well understood by IT project architects and managers. Instead, this chapter now examines how some of these high-level project requirements map onto the technical aspects of PATROL. Some of the particular aspects for use of PATROL include:

- Coverage—ensuring that all the agent platforms are covered, as are all of the operating systems, databases, and applications via Knowledge Modules.
- Development—whatever applications are not covered, such as in-house applications, will require some development resources for either non-intrusive or intrusive instrumentation.
- Console usage—determining who needs to use what consoles for what job roles.

- Integration—what off-the-shelf PATROLVIEW components or development are required.
- Event notification—how administrators are to be informed about detected problems, such as via a help desk integration, or perhaps a direct page from PATROL.
- Home remote usage—how pages are to be handled by personnel when at home.
- Data collection—how much data on what metrics is to be collected.
- Corrections—what automated corrective actions are to be enabled, or perhaps added via development customization.
- Reporting and capacity planning—how this collected data is to be used for management reporting of service levels, resolution times, and future capacity needs.
- Documentation—whether to use hardcopy PATROL documentation, use CD-ROM documentation, or place it onto an intranet web site.

Some of these issues are practical issues related to the project's goals. Others are technical aspects whose scopes depend on how you plan to use PATROL's feature set.

14.3 Coverage Inventory

An important task in planning an application management project is ensuring that every base is covered. The benefits of application management should be gained from all the areas possible. This guideline applies not only to the new open systems project, but also to the older legacy equipment and their applications. Defining the scope of the application management project is an important aspect of its planning. Some of the layers of infrastructure that you need to inventory for the application management project include:

- Server hardware and operating systems
- Databases
- Middleware applications
- Groupware and email applications
- Internet applications
- ERP applications
- In-house applications

A hardware inventory is necessary to determine to what platforms the application management benefits will be applied. Performing this inventory may be easy in the case where you are adding application management to the rollout of a new software project, when you can choose software which can be managed. In the case where application management is being injected into an existing environment to make it less painful to manage, the inventory process is more involved. In terms of hardware and operating system coverage, PATROL has agents to cover a huge range of areas including UNIX, Windows NT, OS/2, OpenVMS, and MVS. Solutions also exist for NetWare, AS/400, Tandem, and others.

Performing a hardware inventory is only half the battle, and perhaps the least important half. After all, this entire book is saying that the application is what is really important. Making an inventory of the software applications that are to be monitored is critical to application management planning. This includes software at all levels, including operating systems, databases, middleware, and applications. And don't forget about the in-house applications, because these are often more important business assets than the commercial off-the-shelf applications. PATROL has a large range of Knowledge Modules to cover the various software layers, and chances are that the commercial software will be covered by PATROL without the need for development. However, the in-house applications are not in the same boat, and require development effort.

14.4 Development Planning

A development effort may be desirable in a PATROL deployment for various reasons. It may be necessary to extend PATROL or to customize an existing component to meet local user needs. This issue applies to both Knowledge Modules and PATROLVIEW integration components that may be required. Hence, the two basic reasons for development are:

- Integration of PATROL with other third-party tools
- Knowledge Module development to extend monitoring capabilities

Some examples of development for integration include:

- Integration via the PATROL API, PET, or PATROL Toolkit for Console Integration to send events into a non-supported enterprise console or help desk
- SNMP trap integration into a non-supported enterprise console or help desk
- Customization to integrate an already used pager notification solution into PATROL
- Customization to send SNMP traps on PATROL alerts according to the required format

Knowledge Modules represent the other side in terms of monitoring applications. Some of the most common monitoring development items as part of a PATROL project include:

- Knowledge Module development for monitoring an in-house application
- Customizing existing off-the-shelf Knowledge Modules to measure particular statistics or detect localized problems that are known to be common in the particular environment.
- Customizing Knowledge Modules to merge in existing homegrown management scripts written in some other language: Perl, shell script, C/C++, TCL, VB, SQL, etc
- Customizing Knowledge Modules to add corrective scenarios under the guidance of specialists in the various specialist IT management groups

The level of difficulty of these development efforts naturally depends on the particular requirements. An in-house application might be simple enough to permit a new Knowledge

Module to be written in a day, or might be sophisticated enough to require several weeks of not only development, but also research about the application itself.

Development effort allows full customization to meet the local needs, but also can elongate the implementation cycle. This lengthening delays the benefits of application management. For this reason, you want to roll out packaged solutions earlier in the project without waiting for development completion. One typical set of implementation phases with regard to monitoring of applications might well be:

- Phase I: Packaged Knowledge Modules—monitor the commercial applications to get immediate benefit at all the underlying infrastructure layers.
- Phase II: Develop Knowledge Modules for in-house applications—build non-intrusive Knowledge Modules to monitor the in-house applications.
- Phase III: Intrusive instrumentation of in-house applications—allow better management as an overall long-term goal of the development effort.

All this discussion of development presupposes trained development staff. The learning of PSL to build a Knowledge Module, or the PATROL API is a new development training item. Various courses are available from BMC Software, partners, and others. However, the different areas of PATROL are not that different from other programming tasks, and a competent developer can quickly become proficient without needing hands-on training. PATROL is an environment that quickly rewards attempts at prototyping for those who are willing to try.

Proactive notification about problems is a key tenet of achieving value from application management tools. Early detection is not very valuable unless the problem is auto-corrected or someone is informed. Not all problems can be auto-corrected, leaving pager and email notification of administrators as a key facet.

14.5 Unsupported Agent Platforms

Although PATROL is very widespread throughout the various hardware and operating system platforms, some may not be supported. For example, various legacy platforms are not supported. This is not necessarily the end of the story for PATROL. Pressuring your sales representative for an agent port to this platform is not the only option. Knowledge Module development can achieve many of the benefits of application management without a native agent on the platform. Some of the options include:

- Remote monitoring features of existing Knowledge Modules
- Pinging Knowledge Modules
- Proxy Knowledge Module architectures
- SNMP-based monitoring (including instrumentation)

Some Knowledge Modules have remote monitoring capabilities that do not require a native agent on the platform. For example, the PATROL KM for Oracle has remote monitoring via SQL*Net, which can permit Oracle monitoring on unsupported host platforms. Many other database Knowledge Modules have remote access capabilities. The PATROL KM for Internet Servers supports remote monitoring via HTTP, which can be used to monitor web servers on Linux, FreeBSD, or other unsupported platforms.

Another simple solution is automatic monitoring that the server remains available. Various implementations of a “ping” type Knowledge Module have been developed. The Knowledge Module runs on a machine for which an agent is supported and polls the server’s availability remotely.

SNMP monitoring is a special case of a proxy Knowledge Module. This is possible if the server platform has native SNMP support, or if SNMP is added by instrumentation via the PATROL SNMP Toolkit. The PSL SNMP functions for MIB browsing and trap receiving can be used to remotely monitor the server or the application on the unsupported server platform.

A longer development cycle is the development of a Knowledge Module that uses a proxy architecture. For example, such a Knowledge Module can launch telnet to perform remote commands on the unsupported platform. An example of such an approach is the PATROL KM for AS/400. The details of proxy Knowledge Module design are discussed in “Proxy Architectures” in Chapter 27.

14.6 PATROL Developer Network

The PATROL Developer Network (PDN) is an important consideration in any development planning. The PDN is an official PATROL program that supports developers in any of the various development activities pertaining to PATROL.

The first reason to investigate the PDN when performing development work is to avoid duplicating effort. Quite possibly, some other entity has already built the component that you are considering. The PDN is the place to research whether such a component already exists. This comment applies to any type of PATROL-related add-on product, including Knowledge Modules or PATROLVIEW-like integration components. The PATROLMARK Solutions Catalog can be found at <http://www.bmc.com/products/catalog/>.

The second reason to investigate the PDN is to gain the advantages of the support it provides to developers. Various useful add-on development tools and documentation are available, as well as web-based support for the PDN Toolkits. For more information, the PDN can be found on the web at <http://www.bmc.com/developers>.

14.7 Console Usage Options

PATROL provides a great range of viewing components to examine the data and events at PATROL agents. These can all connect to the agents to view the status as computed by the various Knowledge Modules. You can use any, all, or some of these console or client components without any limitation on the configuration:

- PATROL Console for Windows NT (Operator and Developer)
- PATROL Console for Unix (Operator and Developer)
- PATROLVIEW integrated consoles (e.g., HP OpenView IT/O, IBM NetView, Tivoli TEC)
- PATROLVIEW integrated help desks (e.g., Remedy ARS)
- PATROLWATCH for Web Browsers (Netscape Navigator, Microsoft IE)
- PATROLWATCH for Windows event viewer
- PATROL CLI text-based interface
- PATROL Event Manager standalone console
- PATROL Pathfinder

The autonomous agent architecture permits all or none of these consoles to be used. You can choose to use no consoles, and leave the agents to run in the dark, only sending pages when a problem arises. The consoles can be started up at any time to examine the situation or perform configuration or administrative actions. Multiple consoles can connect to any of the agents without any restriction on the configuration.

The different consoles also need not see the same view. Consoles can share desktops via procedures to permit consistent views, or each operator can have an entirely different view. Different departments can watch different Knowledge Modules and not get distracted by objects from other classes.

Agent connections are also entirely changeable. The various consoles can either watch the same set of agents, or watch entirely disparate and unrelated sets of agents. One scheme might have a single console watching every agent in the enterprise. A second fallback console might even be watching exactly the same large number of agents.

Another might have different groups having a console each and watching only those agents of interest to them. Yet another scheme could be to merge this idea and have groups watching subsets of agents and one central console watching all the agents to act as the first level of support for alerts generated by the agents.

An important wrinkle in the console issue is the use of development features. To separate a development environment from the production servers is desirable for obvious reasons. When building Knowledge Modules via the PATROL Developer Console, this should be on an entirely separate console view. The console should attach only to the test agents. Development and production environments should never be mixed.

14.7.1 Web Browser Access

A standard part of the PATROL architecture is a console viewer component called PATROLWATCH for Web Browsers. This is a web-based interface into PATROL agents. It can be used to examine the current statuses, view objects, and examine events. This is a nice interface for operators who prefer a browser style of interface. PATROLWATCH supports both Netscape Navigator and Microsoft Internet Explorer browsers (see it in Figure 14-1).



Figure 14-1 PATROLWATCH for Web Browsers view

Note that PATROLWATCH is not a web server monitoring tool. It is purely a view into PATROL Knowledge Modules. However, PATROL does have a Knowledge Module for web server monitoring. This is performed by a PATROL KM for Internet Servers, which has support for a variety of web servers, including Microsoft IIS, Netscape's server, Apache, and others. PATROLWATCH is a viewing tool for any application monitored by a Knowledge Module, which naturally includes internet servers.

14.7.2 Home Remote Access

Several features of PATROL facilitate problem diagnosis from home. The typical scenario is the beeper going off at home and hence the need to log in. The components that might be used in this way include:

- CLI—text-based interface to PATROL
- PATROLWATCH for Web Browsers

PATROLWATCH permits the use of a web browser. This might be easier over a home connection. Even more basic is the PATROL CLI. This is a text-based interactive interface to PATROL that makes easy analyzing the environment from a home PC even using a basic text modem connection.

Of course, any of the PATROL consoles can be used if the remote access method is to use a machine connected over an ISDN line. For example, a Windows NT machine or UNIX via an X emulator, or perhaps a personal UNIX machine can run the console just as well over an ISDN line. However, the other features of PATROL allow using a low-end PC as a less costly home access solution and still having some ability to use PATROL from home.

14.8 Integration Planning

PATROL is an open tool that can be integrated with many different third-party tools. This is an important part of implementation planning, to ensure proper usage of existing resources and infrastructure component. Some of the various types of add-ons for which integration can be considered include:

- Network management consoles
- Help desks
- Enterprise event consoles

To document these requirements is an important step in implementation planning. The next step is to investigate what off-the-shelf components exist for the integration. Several PATROLVIEW integrations with existing consoles and help desks are available.

Another place to research is the PATROL Developer Network (PDN), because the integration may be available from a third party even if it is not available with PATROL directly. BMC partners are producing integrations into various management platforms.

The PDN is involved with all kinds of PATROL development activity, including integrations of all kinds. This fact makes the PDN an important resource if the required integration component does not yet exist and must be built. The PDN has various toolkits (such as the PATROL Toolkit for Console Integration) for console-like integration via the API, and naturally much about Knowledge Module development (e.g. the PATROL Knowledge Module Toolkit). So if the component has not already been developed by BMC or a partner through the PDN, it can usually be built with the toolkits.

14.9 Pager and Email Notification

Notification is the critical aspect of determining whom to tell about problems that occur. This issue largely depends on how application management is to be used within an organization. Departmental usage might make you want to hook PATROL alerts up to the pagers for the DBAs.

Enterprise-wide usage might require all alerts to go to a consolidated service desk according to the normal policy for problem management. Application management alerts might be handled just like network management alerts via a central help desk, or might be handled entirely differently because of the different groups involved. Who gets what alerting method is therefore as much an organizational issue as a technical one.

Technically, you have various possible alerting methods. The most basic is the flashing red icons in the various console views, although this is rarely considered a “notification” method. More typical concerns are issues such as:

- Pager notification
- Email notification
- SNMP trap sending

Paging or email notification functionality is not provided directly by the PATROL agent or console. However, various methods can achieve this. Add-on notification components are available to perform email and pager notification directly from PATROL. The use of such an off-the-shelf component is the quickest way to get up to speed with automated paging on PATROL alerts.

The main technical choice to make is where to do the alerting. It can come directly from PATROL or from another enterprise console component. This issue is often tied with the organizational issue. If alerts are sent directly to a specific group, then PATROL can directly access the paging software.

However, if all alerts are centralized at a service desk with 24x7 operators, then PATROL can forward its events to some other console or help desk, which does the notification by whatever method is already in use. Technically, this forwarding of events to another event management console is facilitated via the PATROVIEW suite of off-the-shelf integrations, SNMP trap sending, or sometimes the execution of a command-line tool to send events into that framework.

The first option is a specialist product specifically designed for pager and email notification of PATROL. This is the quickest way to get up and running with paging from PATROL. No development effort is involved because all the work has been done. A number of such add-ons have PATROL-specific notification support:

- PATROL Alarm Manager—from BMC Software (www.bmc.com) or Insight Innovations (www.inin.com.au). See Figure 14-2.
- CuraMessage—from CuraSoft (www.curasoft.com)

These are commercially available and supported products that specifically aim to address notification needs. The interested reader is referred to the product documentation for these products at the web sites.



Figure 14-2 PATROL Alarm Manager

The other main alternative exists if an enterprise console, help desk, or framework is already part of the system monitoring process. These products often have email or pager notification features.

In fact, these notification features might already be in use for network management alerts or other alerts. By integrating PATROL with these existing procedures, PATROL notification can occur consistently within the organization. PATROL alerts can be forwarded to these third-party console or help desk products in a few basic ways:

- PATROLVIEW off-the-shelf integrations with a variety of enterprise consoles, help desks, and frameworks (e.g., Tivoli TEC, HP OpenView, CA-Unicenter, etc.).
- SNMP trap sending integrations—have the PATROL agent send SNMP traps to the enterprise console or framework. These can be used in cases where no PATROLVIEW integration exists, or in some cases instead of using PATROLVIEW.
- API-based integration—this method is effectively to build your own PATROLVIEW-style integration with a console, framework, or help desk via the PATROL API. This is a more involved development effort than SNMP trap integration, but can be feasible.

These different methods have various advantages and disadvantages. The use of off-the-shelf products such as PATROL Alarm Manager or a PATROLVIEW integration are the simplest ways to get to the required notification architecture, depending on whether PATROL or a framework is to do the notification.

14.10 Documentation Issues

The documentation is often an unaddressed issue of a project. The product documentation is an important part of using PATROL in an environment with many users. Some possible interesting issues here include those regarding paper documentation and also online intranet use of the product documentation:

- Paper documentation—ensuring adequate paper books available.
- CD-ROM documentation—making the PATROL documentation CDROM available.
- Online RTF documentation—placing PATROL documentation RTF files online.
- Online HTML documentation—converting PATROL PDF files into HTML.

Documentation does not have to be a boring aspect of the project. Establishing your own intranet site to push the application management project can also have a pragmatic side of easing the distribution of documentation and training materials.

14.11 Service Level Reporting

The PATROL Agent and Knowledge Modules generate a wealth of monitoring data on availability and performance. This data is the basic ingredient for simple trending reports and formalized measurement of Service Level Agreements. However, today no built-in reporting engine in the PATROL architecture is powerful enough for advanced service level reports. Implementations have a choice between custom reporting methods and various add-on reporting components. Here is a list summarizing the various approaches to the different reporting requirements:

- Console Charting Server—visual ad hoc trend analysis
- Knowledge Module built-in reports—mostly diagnostic reports
- History Loader—analyze parameter history data
- Integration with BEST/1-Visualizer
- Add-on service level reporting components
- Reporting Knowledge Modules using `dump_hist` to gather history data

Depending on the implementation, any of these methods can be used to extract useful summary reports from PATROL's data.

14.11.1 Service Level Reporting Add-On Components

The simplest method to add monitoring and reporting on service levels is to use an add-on component from the PATROLMARK Solutions Catalog. Here is a list of some of the products that might apply:

- SLAManager from Proxima Technology (www.proxima.co.uk)
- BusinessBridge Integration Module (www.systar.com)

Another possibility is to send PATROL data to a more generic component that already has service level management functionality. Some of the options include:

- Boole & Babbage COMMAND/POST
- MicroMuse NetCool/Omnibus

The products can have different architectures and functional features. Some are built specifically within the PATROL infrastructure, and others are integrations of PATROL data into existing reporting or analysis tools. Generally, these add-ons offer functionality that can be used for better reporting on service levels and longer-term trends. Some also offer monitoring or alerting on the basis of service level objects that are composites of other lower-level metrics.

These tools should be examined in the context of the desired form of service level reporting. Also look at the most recent PATROLMARK Solutions Catalog, since more such reporting products could become available.

14.11.2 BEST/1-Visualizer

BEST/1 is a well-known suite of products for performance management and capacity planning. In March 1998, BMC Software acquired BGS Systems, the company that owned BEST/1. The aim is clearly to merge PATROL and BEST/1 into a superior product line. Although this statement has many implications, one of the simplest is that the BEST/1-Visualizer component is an option for reporting on service levels.

14.11.3 KM Reports

The various PATROL Knowledge Modules tend to offer a large range of short-term reports. These are most typically menu commands, text parameters, or recovery commands. These reports are useful for examining and summarizing the current state of the database or application. Clearly, this aim is different for service level reporting, which has a long-term trending focus. The KM reports are useful in the short-term for localized problem diagnosis and performance tuning.

Many of these reports have a dual nature in that they can be both manually and automatically triggered. A number of the Knowledge Modules run these reports as part of annotations when an alert is triggered. So an Annotated Data Point has the text from one of these reports that was executed immediately upon detection of the problem. The report is also stored in a persistent database with history data. This can be very useful for postmortem analysis of problems.

14.11.4 Console Charts

The PATROL Console provides a sophisticated charting engine to enhance the analysis of data. This allows for easy viewing and printing of useful graphs and charts including cross-correlated data points from important parameters. The first step in planning a reporting solution is to create a number of custom charts for the PATROL desktop. The views that can be shown include:

- Current and historical parameter value charts for all the chart forms
- Displays including line graphs, multiline graphs, bar charts, pie graphs, and others
- Cross-KM charts—e.g., mapping Oracle cache hits against CPU utilization parameters
- Cross-agent charts—e.g., showing the CPU utilization charts for multiple machines
- Cross-KM, cross-agent charts—combining any parameters from any agents in any way

These charts can be viewed in the console and also printed out. The annotated data points can also be seen from these views. This view can be useful to perform postmortem analysis of an unexpected peak or trough in the graph. Charts that are including the current values are also updated in real-time by new values from the agents.

User-created custom charts become part of the PATROL console desktop. These desktop entities can be shared through exporting and importing the desktop files, either entirely or in part via editing the text files. Custom charts are drilled into to pop up the chart. Real-time updating charts can be left popped open on the console desktop, and via a console preference; these desktop settings can be re-popped up even when restarting the console afresh.

14.11.5 PATROL History Loader KM

The PATROL History Loader KM is a data centralization tool for all the parameter values measured by the various Knowledge Modules. The PATROL architecture has a decentralized form of history data because each agent has its own history storage database. The History Loader KM takes the data stored in each agent's database and centralizes it to a single central agent. This central agent can then export the combined history data into a text file or a variety of supported commercial databases. Some of the currently supported databases for the History Loader KM include:

- Oracle
- Sybase
- Informix
- CA-Ingres
- Microsoft SQL Server

In this way, the History Loader KM can take PATROL data and export it to another database. The data can then be used by specialized reporting SQL scripts, or a third-party reporting tool can be used. This allows the full power of leveraging more advanced reporting tools that are on the market today, or alternatively, customization to provide user-specific reporting needs.

The History Loader KM is implemented as a Knowledge Module. The architecture of the History Loader KM has a KM component stored on each agent. This component gathers the data out of the agent's history database. Another component sends the history data values across the network using the PATROL inter-agent PSL methods. The central component on the main agent

that receives the various agent's data is also a KM type object. This KM component takes the data it receives and then formats it for export to a file or to a commercial database.

The History Loader KM has had its share of problems. Recent improvements have addressed problems such as performance and configuration difficulty. However, at the time of writing, it still lacks a packaged set of SQL queries for common reports. Therefore, it is a solution that still requires quite a lot of custom work to get to the end result.

14.11.6 Reporting Knowledge Modules

One method of customizing PATROL to perform particular reports is to build a Knowledge Module. The `dump_hist` and `dump_events` executables allow access to the agent's databases via command-line tools. These tools export the data in an ASCII form that can be automatically analyzed to generate the desired results. Some of the formats of `dump_hist` and `dump_events` are designed to be amenable to automated analysis.

One way to achieve this customization is clearly as a Knowledge Module. Since it involves analysis of each agent's local data stores, it makes sense within the agent itself. The Knowledge Module can either use the external `dump_hist` and `dump_events` executables, or use the PSL functions called "history" for parameter history and various functions such as "event_query" for event analysis. Unfortunately, the PSL `history` function does not have as many useful features as the `dump_hist` executable. However, the event manipulation PSL functions are quite powerful.

14.11.7 Custom Service Level Reporting

Typically, PATROL users have created customized solutions for the desired reporting levels. This situation may change as the add-on components in the PATROLMARK Solutions Catalog become more powerful. But at present, this is the usual state of affairs.

You have many different ways to create a custom reporting architecture. Components such as the History Loader KM and reporting Knowledge Modules have been part of such solutions. Some solutions involve pulling data from the PATROL Agent's SNMP MIB into another tool. Another example is the Service Assurance Center created by the BMC Software IS department, which uses custom code that outputs web-based documents showing trends.

14.12 Monitoring Business Processes

The greatest goal of application management is to extend beyond even the application-level monitoring into the business process level. This, more than anything else, bridges the chasm between the technology management issues and the business issues of a corporation. A single blinking icon showing the overall state of a business process as OK, warning, alarm, or offline is a clear demonstration of business-level monitoring.

The standard PATROL feature set for application management in the Knowledge Modules for the commercial or off-the-shelf applications takes PATROL to the application level. These provide application-level management of the application and its lower levels. However,

addition of configuration information about the surrounding business processes can take it to the next level. Some of this business process information is automatically discovered by the application-level Knowledge Modules but some must be supplied by configuration.

The PATROL console supports a simple drag-'n'-drop customization of the monitoring hierarchy. You can drag icons from any of the application layers into a logical view of a business process. This can give a single icon for the overall health of particular business processes or of particular business divisions: one icon for customer transactions, another for payroll, another for the web site, another for the data warehouse, and so on. Individual business processes are represented as a single logical whole, and can be drilled into to examine the lower levels of applications, databases, and servers that comprise the underlying technology for the business process. Regions of business can also be aggregated and managed as a whole, whether it be a physical or a logical division of the business into regions.

You have unlimited flexibility in the creation of these logical views. Different consoles can have different logical views. Multiple views are also available easily. Some of the logical views that are worth considering include:

- Business-level views—top-level views bridging applications and departments
- Application-level views—icons for each critical application
- Departmental views—icons indicating the status of each organizational department's resources
- Performance views—central combined views of all key performance metrics
- Capacity views—combining all capacity and load metrics
- Physical views—true host-level views of all icons; useful for technology access
- Geographical views—use a single icon and view to separate buildings, cities, states, or continents and show their logical status independently

The creation of these various views is easy via drag-and-drop for small sites and automatic console desktop file generation for large sites. Icons for low-level resources can also be contained within multiple business processes through the use of shortcuts that allow the object to participate in multiple logical views. Icons and shortcuts can be dragged individually from different servers, databases, or application objects, and can be used individually (e.g., a single CPU utilization metric for a server), or as a whole (e.g., an entire server or database object). These shortcuts also permit the combination of both physical-level, application-level, and business-level monitoring. A filesystem can be seen in both a physical host-based view, an SAP R/3 application-level view, and a business process view that summarizes all these layers.

14.13 Authorized Service Provider (ASP) Program

A good way to become an expert on PATROL implementation is to enter the Authorized Service Provider (ASP) program. This program offers training for individuals and organizations on the varied aspects of implementing PATROL. Also, a wide range of useful documentation on

PATROL implementation is available, starting with the “PATROL Implementation Basics Manual,” which is part of the PATROL standard product documentation. Also, a number of useful tools are available that can help facilitate the process in technical terms. More information on the ASP program is available on the BMC web site at www.bmc.com.

In-House Applications

One of the most valuable parts of an IT infrastructure is the self-written in-house application. These applications contain business logic that is very specific to the corporate business purpose. This makes these applications more valuable and their availability more critical. The management of these applications is therefore also critical. However, it cannot be achieved through off-the-shelf application management components.

The solution comes via PATROL's extensible architecture that allows an entirely new Knowledge Module to be created. This can be custom written to achieve the management and monitoring features required for the application. Before we look into the technical details of writing a Knowledge Module, let us examine the planning aspects. The architecture of the resulting Knowledge Module is very closely tied to a number of decisions regarding how to approach the management of the in-house application.

A major assumption this chapter makes about in-house application management is that a non-intrusive approach is chosen. This means that no changes are made to the application itself, and the management solution operates using whatever interfaces are available. The alternative is clearly an intrusive approach where the application itself is revised to contain instrumentation. The details of instrumentation for an intrusive approach are covered in Chapter 16.

15.1 Advantages of In-House Application Management

The addition of application management via a Knowledge Module brings the application management of the critical in-house applications into the generic application management infrastructure. This allows the leveraging of the generalized application management functionality rather than re-implementing it just for the application. Some of the functional areas that are leveraged include:

- Automatic 24x7 monitoring of the application for failures
- Proactive notification via flashing icons, email, pager, SNMP traps, or integration (framework events or help desk trouble tickets)
- Proactive scheduling of corrective actions
- Console views showing objects in iconic, text, or outline views
- Web browser display of the application via PATROLWATCH
- PATROVIEW integration with frameworks and help desks (e.g., Tivoli TME, HP OpenView, Remedy ARS, etc.)
- Historical storage of the data
- Porting efforts reduced by agent and PSL cross-platform infrastructure

No special development activity is required to achieve these features. These advanced application management facilities come as part of the overall use of PATROL. This fact in itself significantly reduces the development effort to add application management to an in-house application. Rather than focus on infrastructure, the Knowledge Module developer can focus on the features required for monitoring and administration of the application.

15.2 Monitoring Features

The first category of application management features is those that do not change the application but only monitor its status. Some of the major functionality categories in this area include:

- Discovery
- Availability monitoring
- Performance monitoring
- Capacity monitoring
- Alert generation
- Reports

Discovery refers to the detection of the application on the machine. The Knowledge Module needs to detect the application and create objects within the object hierarchy to represent the application components. The number of objects depends on the application. For example, a simple server component might have only one single object, whereas a complex application that spawns multiple processes to handle jobs could have multiple objects.

Availability monitoring involves determining the overall status of the discovered objects. This is mainly about detecting “crashes” of the application. For example, if the application is a daemon or server and this process is found to be missing, this could be a crash. It could also be a normal shutdown, and you need to have a method of determining whether a normal shutdown or a crash has occurred. The availability of an object is typically one of a small set of states such as up, down, or crashed, rather than a numerical measure.

Performance monitoring refers to numerical measurement of metrics about the performance of the application. One simple measure might be the CPU usage by the operating system process that is the server process. A more advanced measure would be end-to-end response time to show the time taken by transactions.

Capacity monitoring refers to measuring the workload on the system to complement performance measures. Example parameters might be the number of users or the number of transactions. The aim of capacity parameters is to record historical data about the load on the capacity of the application. These parameters are not likely to generate alerts about the current situation. Instead, these parameters are mapped against the performance data in long-term reports to analyze the response levels according to load, and thereby predict future capacity needs. This data is best uploaded into a generic reporting or capacity planning tool such as BEST/1 for more detailed analysis.

Alert generation is the issue of deciding when the Knowledge Module is to issue an alert. This is trivial for availability measures where it basically alerts when a server crashes or when some similar catastrophe is detected. Performance parameters can have alerts associated with peaks in load or response time. In some cases, you want to smooth out the peaks to alert only after a number of such threshold violations have occurred.

Reports are another form of read-only analysis of the application. One useful report is that of application configuration information such as its version, its install directory, when it was installed, what tuning parameters it has set, and so on. Other forms of reports can arise from the other features such as performance reports, incident reports about alerts, and capacity analysis reports.

15.3 Data Sources

The delivery of monitoring functionality requires data about the application. Assessing what data is available is an important part of planning in-house application management. Since we are assuming a non-cooperating application that has no special interfaces, one of the best places to find data is the footprints in the operating system left by the application. Some of the operating system data sources include:

- Files—static files, configuration files, temporary files, log files
- Processes
- Registry (Windows NT)
- SNMP MIB
- Performance database (Windows NT)
- Other Knowledge Modules

Static files are a good method of discovery. If the application is installed in a standard directory, such as `\myapp` on Windows NT or `/usr/local/myapp` on UNIX, the existence of this file is a key part of auto-discovery of the application components. The presence of this

file may indicate that the application is clearly running. Of course, if the application is not installed in a standard directory, this is not such a useful discovery hint. Searching the entire filesystem is a touch inefficient, and the solution is best left to configuration. However, even if files do not aid auto-discovery, some other data sources might.

Configuration files are a special type of static file. These files contain basic startup information about the application. They can be used to produce administrative reports about the current configuration of the application.

Temporary files might also be worth checking for in `\temp` or `/tmp`. These might help with auto-discovery or perhaps in some of the other monitoring characteristics. Perhaps the size, growth rate, or modification date of these files is indicative of the number, frequency, or currency of the transactions passing through.

Log files written by the application are often a gold mine of application information. Good applications usually write status or failure messages to a text log file. Analyzing this text is a powerful form of monitoring. The log file may contain configuration information such as the version of the application, when it was last started, and so on. Error messages written to the log file can be easily converted into application alerts if you know what messages to try to find. If the application writes enough information, such as timestamps, to the log file, sometimes you can gather performance statistics merely from the log file text. In some cases, these log files are written to the standard system directories, and in others they are required to determine the directory where the application is installed.

The registry on Windows NT can be a simple test for the presence of the application as part of discovery. It may also contain useful configuration settings such as application version or tuning settings.

On platforms other than Windows NT, you may find some equivalent of the registry. For example, if a particular package install method is used or a commercial distribution tool is used, this can contain useful information for discovery that can perhaps be extracted.

If the application is sophisticated enough to have SNMP support, its MIB can be queried for a variety of useful information. Firstly, the MIB can be used as a form of discovery simply by querying for a known OID whose presence guarantees that the application is running. Secondly, all the values that can be read from the MIB will typically map to particular parameter metrics. In fact, this reason is typically why an application is instrumented with a MIB in the first place: to expose this type of information. Thirdly, traps sent by the application can be intercepted by the Knowledge Module to generate alerts. Chapter 16 discusses how the PATROL SNMP Toolkit can be used to add this type of instrumentation to an application without SNMP support.

The Windows NT performance database is a good source of performance statistics. Those applications that do write values into this database are easily monitored. These values can clearly be used for performance or capacity parameters. They can also be used as a form of discovery to test whether the application is present. Unfortunately, no equivalent really exists on non-NT platforms.

Another form of existing data comes from the existing Knowledge Modules. For example, the operating system Knowledge Modules already have discovery and monitoring features for components such as CPU, processes, and printers etc. Similarly, for in-house applications running on databases, the various existing RDBMS Knowledge Modules are a good source of information about servers, tablespaces, users, and other database artifacts. All this existing Knowledge Module data is available for the in-house application Knowledge Module to use. It is internal to the agent upon which all the KMs execute and is therefore very efficient to access.

15.4 Administration Features

Monitoring is a passive checking of the status or behavior of the application. Administration is the other side of the coin that involves altering the application. These administrative requirements take on two basic features:

- Interactive user commands for ad hoc administration
- Corrective actions to be used as automated recovery actions

A variety of different forms of administrative functionality exist. All the basic requirements of application administration can be codified in an application management tool. Some of the areas of administrative functionality include:

- Application startup and shutdown
- Configuration changes
- Performance tuning
- User administration
- Security changes
- Archiving or restoring data

All these different areas of administrative functionality can be captured if the interfaces permit it. The form of these interfaces is what we now examine.

15.4.1 Operating System Administration Interfaces

Administration of an application involves changing the behavior or configuration of that application. This is often more difficult to achieve than monitoring features. It requires that the application take heed of some type of access in order to change its own behavior. The set of operating system access points is similar to that for monitoring:

- Files—configuration files
- Processes
- Registry (Windows NT)
- SNMP MIB

Files can sometimes be used to affect the application. The most common is changing a configuration or startup file. This will typically require a restart of the application for the new configuration to take effect. Another form of administrative functionality involving files is to copy, remove, or archive application data files.

Processes can be used to alter the application. You can directly kill or blast a system process in order to shut it down. Processes can also be restarted in order to begin the application automatically, in the case of a recovery action. Some applications may support signals such as SIGUSR1 or SIGUSR2 with some functionality, such as re-reading the configuration files.

The registry values can be changed to alter the application configuration. This change might take effect immediately, or an application restart might be required.

An SNMP MIB will usually offer some writable variables. These can offer any form of change to the application behavior. This depends on how the application is instrumented with SNMP. However, this method is usually limited to basic changes to simple numerical values.

15.4.2 Advanced Administrative Interfaces

All these administrative methods are quite limited because very little cooperation comes from the application. It is a black box application that cannot easily be changed. For this reason, we see the majority of complex commercial applications having a variety of administrator interfaces. Typically, these more advanced applications offer some form of configuration GUI or command-line interface. Unfortunately, a GUI administration tool, such as a GUI client that performs configuration of a server, is of very little value for automated application management. A GUI cannot be spawned in batch mode. A command-line tool is far more valuable because it can be used more flexibly to achieve whatever results are desired. A GUI is a closed and fixed interface, but a command-line executable is an open, extensible interface.

15.5 Leveraging Existing Work

A Knowledge Module can make use of existing work a number of ways. A typical organization has already developed some monitoring or administrative commands or scripts. Similarly, PATROL already provides a variety of out-of-the-box monitoring and administrative solutions. Both of these areas of existing expertise can be leveraged for a new Knowledge Module for an in-house application.

Existing scripts are simple to add to a Knowledge Module. The languages for these scripts could be UNIX shell scripts (sh, csh, ksh, etc.), Perl, TCL, SQL, Visual Basic, or any executables written in other languages. These scripts can be run without change for administrative commands. For monitoring features such as parameter values or application statuses, some post-processing of the results in PSL may be necessary to map the output or termination status of the script into a PATROL parameter or status.

Existing Knowledge Modules are another source of data for an in-house application. Rather than re-work the analysis of filesystems, processes, or other low-level resources, the data from the operating system Knowledge Modules (e.g., PATROL KMs for Unix, Windows NT,

VMS, and MVS) can be gathered within the agent and applied to the new Knowledge Module. Similarly, the database KMs can be used to provide data for an in-house application based upon a particular database, or in fact, even if based upon numerous different types of databases.

15.6 Knowledge Module Development Activity

The development of a KM is analogous to any other software development process except that it involves operating at a high-level KM and PSL script language level. This allows rapid KM development, with a simple proof-of-concept KM often available within hours by experienced KM developers. For developers experienced in other areas but not in PATROL, they may take a few days or weeks to get up to speed.

An important point to note is that no matter who is doing the work, it often requires the input of someone with expertise about the monitored application. To achieve all you want from this form of in-house application management via a non-intrusive Knowledge Module, you need to know the in-house application very well. Some useful hints may come from the documentation. However, these are usually from an operational point of view, rather than an administrative point of view. Even an administrative point of view might not explain all the inner workings of the application, which you need to know in order to interpret the various data sources. Design documents are getting closer, as long as they aren't written on the back of a postage stamp, but often you need to engage the help of an in-house developer who knows the details. Fortunately, you can more easily get a developer to wax lyrical about the obscure features of the application than request that they revise the application to add instrumentation.

Generally, an application expert can more easily learn how to write a KM than a KM developer can learn an application. This statement is particularly true because KMs often must make use of undocumented functionality in error logs or via open interfaces. However, the most common form is a partnership between IT personnel who are PATROL-proficient and know what administrative features are desirable, and the application development team who know how to get the data. This partnership is an efficient use of the expertise of both areas.

15.6.1 Milestones in Knowledge Module Development

The development of a KM depends largely on what monitoring and management tasks are available and desirable for the application. A clear relationship exists between both the complexity of the monitored application and the openness of the application for management, in terms of development difficulty. Nevertheless, a natural separation of KM functionality into various general areas occurs:

- Discovery
- Parameters
- Alerts
- Menu Commands
- Recovery Actions

These are useful milestones in the KM development project plan. They also provide the opportunity for the provision of a separation of the KM into a “limited” KM with discovery and parameter monitoring capabilities only, and a more fully developed KM having all the alerting and management features.

15.7 PATROL Developer Network

The PATROL Developer Network (PDN) is a program founded by BMC Software. This program aids developers interested in a variety of extensions or integrations with PATROL. This is aimed at developers interested in writing Knowledge Modules for extended application management at all levels from the hardware through to high-level in-house applications, and also at developers building PATROLVIEW-like console integrations. Several toolkits are available as part of the PDN, including:

- PATROL Knowledge Module Toolkit—advanced tools, examples, and information for Knowledge Module development
- PATROL Toolkit for Console Integration—advanced tools for PATROLVIEW-like integrations using the API

In addition to development tools, a variety of useful documentation and educational material is available. More information on the PDN is available on the web via www.bmc.com/developers.

Application Instrumentation

The most valuable part of an enterprise software infrastructure is often the in-house business-critical application. The ultimate goal of application management is to extend the reach of monitoring, notification, and proactive problem correction to this business application. This is where the crucial business logic is contained, and the availability of these applications is the most critical.

The PATROL solution for the most immediate benefit from monitoring these applications is Knowledge Module development for in-house developed applications. This is a non-intrusive solution that does not involve instrumentation or any change to the application itself. However, instrumentation of the application to make it easier to manage is the next logical step in the evolution of an application management project. This chapter addresses the issue of when to instrument and how to make the best use of PATROL as part of an instrumentation plan.

16.1 Intrusive versus Non-Intrusive Management

You have two basic approaches to in-house application management depending on whether you want to modify the application. A non-intrusive approach does not require application modification and involves building a Knowledge Module using existing interfaces. The intrusive solution is to modify the application with instrumentation code in order to help build an even better Knowledge Module.

Non-intrusive Knowledge Modules are clearly the antithesis of instrumentation. They take the pragmatic approach of using whatever manageability features happen to already exist. Therefore, a Knowledge Module for an application can be deployed without changing the application, and without even shutting down the application.

The trade-offs are clear. Instrumentation offers the capabilities for more advanced application management but at the cost of some extra effort and a lengthier cycle to introduce application management.

A combined solution is also possible using both intrusive and non-intrusive management. When using PATROL, the solution is usually to build a Knowledge Module for the current release of the application that is already running in production, and then work on instrumentation to allow a better Knowledge Module for the next release of the application. This next release may already be part of the development plans for improvements to the business aspects of the application, particularly with the impetus behind the year 2000 revisions globally, and the European Monetary Union (EMU) in European countries. With PATROL, the solution to intrusive instrumentation can be either the enhancement of existing interfaces to allow a better Knowledge Module, SNMP-based instrumentation via the PATROL SNMP Toolkit to produce the Knowledge Module, or WBEM-based instrumentation.

16.1.1 Non-Intrusive Management

The simplest method of managing an application with PATROL is to use a non-intrusive method. This assumes no change to the current application. In a way, this is not really even instrumentation. A Knowledge Module can be developed to take as much advantage as possible of whatever management hooks exist for the application. The entire approach of Knowledge Module development is based around a non-intrusive approach of making the best use of whatever is already available (e.g., install files, processes, log files, and so on). Indeed, PATROL has been very successful in developing powerful Knowledge Modules for a wide variety of applications without adding special features to these applications.

Therefore, a non-intrusive approach to application management should be the first consideration. The use of a non-intrusive solution such as a Knowledge Module has various advantages, including:

1. Redesign, redevelopment, and retesting of the application is avoided.
2. Redeployment of the application in production is avoided.
3. The application is untouched, and management can be added “outside” of it, without even restarting the application.
4. Current and older versions of the application are not stranded without management.
5. Rollback of management solutions, should they cause problems, is easily possible, since the application is untouched.
6. Application management can be handled by the administration group (who best understand its importance), rather than the application developers.
7. Time required for instrumentation design and development can be refocused on more important development areas such as business logic, or resolving Year 2000 or EMU issues. Gartner Group estimates that instrumentation takes around 25% of application development time.

8. Existing application monitoring scripts can be reused, such as shell scripts, Perl, TCL, etc.
9. Return On Investment—the shorter cycle of non-intrusive management gives more rapid ROI on any investment in improving application availability via application management.

Clearly, the main disadvantage of a non-intrusive management approach is that you have to make do with whatever management hooks are available. Better functionality benefits system administrators through the rapid detection of problems with the application and capabilities for resolution of problems. Nevertheless, the existing management interfaces are often quite substantial and can often yield a decent Knowledge Module with significant management capabilities. The operating system has quite a wealth of information about some applications without the application developer having to lift a finger. Some of the sources of management or monitoring data can include:

1. Processes—is a system process up, zombied, or down? How much CPU and memory has it used?
2. Files—the presence of a file can have significance. The size of a database file can be a parameter. Archiving or removing a file could be a useful administration command.
3. Log Files—plain-text error log files can be of great value if the application has them.
4. Executables—do you have an executable to run that measures something or builds a report?

Hybrid solutions are also possible. The consideration of external executables also adds a new wrinkle to the choices for building the management solution. Rather than revise the entire application, can you build a separate executable just for management? For example, in a client-server application, can you write a “monitoring client” whose sole purpose is to connect to the server and gather monitoring information or send management commands? Maybe the client could issue dummy no-operation queries to the server just to measure response time. The dummy client could use the same security authentication methods as a normal client by configuring the Knowledge Module with the correct information such as a username and password to give to the server. Clearly, this issue depends on the design of the application itself, but does show that manageability can potentially be improved significantly without having to revise or re-deploy the central application server in some cases. Further discussion of this management client executable idea is given in the section “Command Line Tools” later in this chapter.

16.1.2 Intrusive Instrumentation

In a situation where you can revise an application, notably an in-house application, certain advantages exist to an intrusive management solution. The main advantage of moving to the more involved process of building an intrusive management solution is the improved manageability that can be achieved. Although “making do” with whatever already exists allows a faster solution, it is exactly that. A lot can be done without instrumentation, but some features cannot

always be achieved without instrumentation. In addition, the revision and re-deployment of the application might be a non-issue, if this has to occur for the usual maintenance reasons of enhancing the functionality of the application itself. And while enhancing the business functionality, why not enhance its manageability? The extra effort can pay off in a big way in terms of the application manageability returns.

The basic aim of instrumentation is to turn the application into a “white box” or “management-aware” application that cooperates with the application management software. The application needs to adapt itself to provide enough information and enough integration points to offer full manageability. A variety of methods can “open up” the application to allow more advanced management.

16.2 PATROL SNMP Toolkit

The PATROL SNMP toolkit is a component of PATROL specifically aimed at addressing the issue of intrusive application instrumentation. The developer defines what points to expose, and they are brought into PATROL via PSL SNMP functions. The five basic steps to instrumentation via the PATROL SNMP Toolkit are:

1. Define what to monitor (in MIB notation).
2. Convert this definition to code that has specific API calls (by running the MIB compiler).
3. Add these API calls to your application.
4. Link the application with the toolkit runtime library.
5. Generate the PATROL Knowledge Module (also from the MIB notation).

These steps take instrumentation from the low-level of defining C variables and callbacks for attributes all the way through into PATROL. The MIB notation, which is a well-defined plain-text format, includes definitions of named metrics and their associated C global variables and callback functions inside the application source code. The MIB compiler creates code that includes API calls to get and set these variables. The application must then use these API call stubs to expose the data and fill out the callbacks. The internal application data is externalized via the SNMP protocol by linking in the runtime libraries for the PATROL SNMP toolkit. These libraries bring in the SNMP support in such a way that the main application event loop need only regularly hand control to the toolkit API calls. This sequence of steps creates a full application that can respond to SNMP requests from the PATROL agent.

The use of SNMP allows two-way instrumentation with support for read and write operations. This allows monitoring and management. PSL SNMP get functions in parameters can read the instrumented application’s MIB for monitoring. PSL SNMP set functions in menu commands can change the application’s MIB to cause configuration or other administrative changes to the application. The PATROL SNMP Toolkit can generate the basic Knowledge Module, including the PSL calls, directly from the MIB notation. This KM can be used as is or extended via normal PSL and KM development techniques.

The PATROL SNMP Toolkit can be used either with a PATROL KM or independently in other SNMP management consoles. Since the PATROL SNMP Toolkit provides a full MIB for the application, this allows the instrumented application to be viewed in SNMP consoles such as Hewlett Packard's OpenView or IBM's NetView, just to name a couple.

An application instrumented via PATROL SNMP Toolkit can be seen in PATROL versions 3.0 onwards. Through the development of a simple KM using the SNMP features of PSL, all the features of the instrumentation are available in the KM model. Parameters and thresholds can be used to issue alarms based on these values. Alternatively, SNMP traps can be converted to PATROL alarms through the KM monitoring incoming SNMP traps. This KM can be automatically generated by the PATROL SNMP Toolkit and can also be manually extended through PSL programming if desired.

16.3 Knowledge Modules versus Instrumentation Standards

The approach taken by PATROL Knowledge Modules differs significantly from instrumentation standards. The standard form of Knowledge Module uses a non-intrusive approach that takes whatever interfaces and operating system data is available. This aims to make the best monitoring and management features possible with existing applications.

One of the key differentiators between the wide availability of Knowledge Modules to manage applications with PATROL and the apparently slow adoption rate of intrusive instrumentation standards is an organizational one. No need exists for intrusive instrumentation, nor for any revision of the application itself, when building a Knowledge Module. A Knowledge Module can be built as an add-on piece, external to the application, using whatever interfaces already exist. So an application vendor can assign a fully independent group within its organization to the task of building application management via a Knowledge Module. This offers the attractive option to deliver manageability without affecting vendor delivery plans for the revisions to the application.

By way of contrast, extending the application itself for better instrumentation is a lengthy process. Not only does it require an application revision, but the manageability advantages only appear to application customers as they migrate to the newer application revision. Via a non-intrusive Knowledge Module, all existing versions of the application can have manageability enhancements. The current and even past releases are handled via the same Knowledge Module through its use of whatever is available. Hence, the features of PATROL make it somewhat standards-agnostic. The use of the Knowledge Module as a method of non-intrusive application management effectively means that PATROL can manage any application regardless of its standards enablement.

However, a high probability exists that some form of instrumentation standard will prevail in the long term. Where does standardized instrumentation leave PATROL's Knowledge Modules in the long run? An instrumentation standard will allow any application management tool to more easily gather statistics. The focus will increasingly move onto higher-level functionality for application management. Some of the areas that arise beyond monitoring and administration

include predictive modeling, root cause analysis, expert advice, problem diagnosis expertise, more complex corrective actions, and other advanced areas.

16.4 Transferring Management Logic to PATROL

An important point when revising an application is not to waste effort in coding the application to have too much management functionality. This comment applies to the use of the PATROL SNMP Toolkit and to the other methods discussed throughout this chapter. Frequently, management logic can be transferred outside the application into the monitoring tool. Because of the power and flexibility of PATROL Knowledge Modules through the script language, PSL, much of the management and monitoring processing and analysis can be handled outside of the application.

The instrumentation of the application merely has to make sure that PATROL has a method to get the data it needs, and this method need not be complex at all. Some examples of monitoring-style computations that need not occur within the application, but can be handled by PATROL, include:

1. Historical storage of data—PATROL handles this automatically.
2. Averaging or other summaries of metrics—PATROL can handle these easily.
3. Alerts and thresholds—PATROL handles these, and the application need not decide itself what thresholds apply to measured values.
4. Recovery actions—corrective actions can be handled most flexibly within PATROL rather than hard-coding these and losing the flexibility to control these without application revisions.

Hard-coding management functionality such as this into the application is duplicated effort and also potentially forces further application revisions. For example, if the application makes its own decisions about when an alert should be issued, this can be changed only by revising the application (or re-configuring the application).

The preferred solution is for the application merely to report the measured statistic and allow the configuration of the PATROL KM thresholds to decide what is or is not an alarm. After all, the application developer's idea of when an alarm should occur might not correspond with the administrator's needs.

16.5 Log File Instrumentation Methods

The term “instrumentation” usually means the use of special APIs in the application, such as via the PATROL SNMP Toolkit. However, the idea of instrumentation is more general and includes any type of application changes that make the application easier to monitor and manage. A number of simple, pragmatic improvements to an application can yield significant gains in manageability. Sometimes this simplicity can be as good a solution as instrumentation toolkits. Some of the simpler ways for the application to provide data to PATROL include:

- Touching files
- Log file text

PATROL can easily detect the presence or absence of a file, and touching files can be used to communicate status with PATROL. Obviously, the contents of a file can offer more information than its mere presence. A plain-text log file containing useful information can offer an incredible wealth of information. Not only can PATROL easily read the contents of the file, but it can also “tail” the file via the PSL file operation functions (i.e., PSL `fopen`, `fseek`, `read`) or via an external system `tail` executable. Therefore, PATROL can know when new entries are added to a file, and this simple idea is a powerful feature.

16.5.1 Log File Information

The log file is such a simple and powerful method of communicating management status. A number of guidelines are available for maximizing the potential of this concept. Three basic considerations govern the use of log files with PATROL:

- Location—Where is the log file?
- Format—How do you analyze the log file contents?
- Information—What do the messages represent?

Clearly, the level of formality with which you can approach these issues varies. However, by addressing all of them properly, you can greatly increase the value that PATROL can add by log file monitoring.

16.5.2 Log File Location

Firstly, some general comments about how to make the log file easy for PATROL to find and process. If the Knowledge Module cannot find or access the log file, the entire issue is moot. Various considerations are related to making the log file available to PATROL:

- Standard directory—Allowing PATROL to easily find the log file in a known place can help, although if this is not possible, you can configure each PATROL instance with the location.
- Standard file name—Clearly, PATROL needs to know what filename to seek. If the filename changes, such as to contain the date, this can be a minor inconvenience, but can be resolved.
- Permissions—Ensure that the file permissions are adequate to permit read access.
- Shared locks—On platforms such as VMS and OS/2, ensure that the application has the log file opened to permit a shared read lock. Otherwise, PATROL cannot read the log file while the application executable is running.

The final fallback is to rely on configuration. Typically, if auto-discovery of the application fails and requires configuration, then the log file location will also need to be part of that KM configuration sequence.

16.5.3 Log File Formats

The choice of a logical format for the textual messages in the log file can make processing the information much easier. Some of the considerations to consider include:

- Prefix each message with a simple identifier of the type of message, such as whether it is configuration information, a status change, a major problem alert, and so on.
- Timestamp each message—knowing when a message was written to the log can be very useful for a number of analysis tasks.

Other general considerations that apply to the choice of log file formats include:

- Portability—If the application sends operating system messages, such as system error messages, from spawned child executables or even from C's `strerror`, these can differ across platforms. The log file format should be such that messages can be identified regardless of their particular contents. For example, all system error messages could have a prefix added, which could be as simple as “SYSTEM ERROR:” or such.
- Internationalization—If the application is to run under multiple locales, the messages might appear in a different language. This can create difficulties for the KM to recognize patterns, and needs to be considered. The solution might again be a simple prefix that is always in a particular language and therefore easy to identify. Alternatively, the KM code can be modified to speak multiple languages via PSL message catalogs (in the 3.2 agent).
- Migration—When revising the application in the future, care is required to ensure that the KM can still work correctly. If the log file format is to change radically, the KM will need revision, too. The best solution is to decide on a format for the log file and stick to it.

16.5.4 Log File Processing

What sort of messages should appear in the log file? Certainly, a huge range of possibilities is available. This is the domain of many of the application instrumentation standards that were examined. Following these standards is one approach; informal pragmatic choice of relevant data is another. Suggestions for categories of information to store in the log file can include:

- Initial configuration information
- Alerts for major problems
- Security problems
- Significant events or changes
- Measured values or parameters

The start of the log file typically contains startup configuration information. These are useful for the KM info box popups or other reports. This configuration information could include:

- Application version number
- Application start time
- User starting application
- Port number of communications
- Other configuration data—the set of useful information is typically application-specific

Alerts are another clear example of information that should be externalized. If the application detects an internal failure or problem of some sort, this information can be sent to the log file for analysis by the management tool. Some examples of problems that could be detected include:

- Child process died unexpectedly
- SQL script execution failed
- Internal assertion was triggered
- Memory allocation failed
- Lock request was denied
- Disk read or write failed
- Client connection to server lost abnormally
- Internal data consistency checks failed
- Transaction was aborted and rolled back

The application can also detect and log internal events that are potentially indicative of a security threat or intrusion. Security issues that could be monitored and logged include:

- Authentication failures of any kind
- Repeated client connection requests with failed passwords
- Changed file checksums or modification dates

All of these alerts or security issues potentially deserve an alarm from the PATROL Knowledge Module. When these messages are read by PATROL, an event can be logged in the PATROL event system, and an alarm can be generated to alert system administrators if needed. The KM can be configurable to decide which of these internal events deserve alarms and which are ignored.

Another level of events are those that are less clearly a problem. Logging all these events can provide the KM with useful monitoring information for calculation of complex parameters and for audit trails. Some examples of information events that can be useful include:

- Transaction started
- Transaction ended
- User connected to server
- User disconnected from server
- User added or deleted from configuration

If these events are logged with enough information, PATROL can generate useful statistics from them. The aim of doing so is not to fill up your disk with useless logged information but to generate just enough information for PATROL. For example, if the transaction start and completion messages contain the timestamp and a transaction id, PATROL can compute transaction completion times and monitor the average transaction times, the total number of transactions, and the transactions per minute rate. Similarly, user connection and disconnection messages can monitor the number of current users, the connection time for any particular user, and the average user connection time.

16.5.5 Measured Parameters from Log File Information

I have already discussed how the logging of events can yield quite valuable measured parameters such as transaction times or user connection times. These are parameters that can be externally computed by PATROL from a relatively small amount of information about internal events from the application. The next level of measured parameters are those that are internally computed.

The application typically has a large number of internal attributes that in general are a good indicator of the overall application status and performance. These values can be written to the log file for PATROL to pick up and apply thresholds and correlation against it. If these values are not changing frequently, they could be written to the log each time they change. An example might be the number of total users connected, which could be written each time a user connects or disconnects. A more frequently changing attribute counter might be written to the log only at regular intervals. By writing this value with a clear message and a timestamp, PATROL can easily retrieve the value and use it as a monitored external value.

An important point about logging measured parameters to the log file is whether to do this internally or to let PATROL do the work. For example, the transaction count could be measured by PATROL from logging the transactions, or the application could retain an internal transaction counter variable and log that value regularly. In this case, the decision is really a matter of style, but in other cases, you do not want to make the application do the work that PATROL could do externally.

16.5.6 Implementing Log File Monitoring

The simplest way to perform log file monitoring is to take one of the existing generic log monitoring Knowledge Modules: some are available from PDN partners, such as Proxima Technology (www.proxima.co.uk). Refer to the PATROLMARK Solutions Catalog for this product.

The alternative is to code your own log file monitor. The monitoring of a log file from a PATROL KM is an advanced PSL coding exercise. The aim is to “tail” the file by watching for new text lines added to the end of the file. You have two basic methods to choose from in the Knowledge Module:

1. PSL file operation functions—Use PSL `fopen` to open the file, PSL `fseek` to get to the end, and PSL `read` to read data from the file. The PSL `file` function can be used to check file sizes, modification dates, and other statistics that indicate that new data is available at the end of the file.
2. External tail process—On UNIX is a standard “tail” executable that can be launched by the PSL `popen` function. The “-f” option is used to have the `tail` command operate in an interactive manner waiting for new data at the end of the file. PSL `read` is then used to wait for new output from the `tail` executable. Similar executables exist on other platforms, or can be easily written if needed.

Choosing between the two operations is a matter of style. The PSL file operations are more efficient than launching a separate system process. However, the PSL file function implementation requires that the PSL parameter repeatedly poll the size of the file to see when new data is available. The external executable operation does not require polling, because PSL `read` on a pipe channel will block until new data is available and then awake immediately. PSL `read` cannot do so on an `fopen` channel.

One final caveat in the implementation using an external `tail` process is that the termination of this process must be handled. This handles the case where a system administrator accidentally kills the long-running `tail` process. The PSL code can detect that this as a PSL `read` failure and re-launch the process after the `tail` process dies.

16.6 Simple Administration Functionality

Administrative functionality refers to the ability of the management tool to make changes to the application. For example, administrative functionality includes application configuration, application performance tuning, and generating reports.

Unfortunately, adding administrative functionality to an application is more difficult to do than simply adding monitoring functionality. Clearly, you cannot just read a log file to change the way an application works. However, the simplest form of administrative functionality is that which can be done through the operating system services. The category of administrative functionality can be separated into those that are “read” operations such as reports and the “write” operations that change the application.

Some examples of “read” administrative commands that can be performed without any advanced internal application interfaces could include:

- Report listing the application log file contents
- Report listing application configuration settings
- Report listing files, such as all the temporary data files

Examples of the “write” administrative commands using only operating system services that aim to change the application could include:

- Restarting the application
- Killing processes
- Signaling processes
- Downing the application
- Archiving files
- Removing files
- Pinging hosts
- Editing an application configuration file

Certainly these relatively simple commands and reports can be achieved via operating system level facilities. Security limitations may also prohibit some features. For example, PATROL cannot kill a UNIX process that was launched by another user such as root, even if the PATROL agent is launched with setuid root.

Hence, for more advanced administrative features, you need to have some way to get into the application and tell the application to make changes to itself. This typically takes the form of a command line tool or, in more advanced applications, an explicit API for the application.

16.7 Command Line Tools

Operating system facilities and simple techniques such as log files can create a good monitoring and administration tool via a PATROL Knowledge Module. However, the next level of technology is to have a command line tool or executable that the KM can use to perform both monitoring and administration. This tool allows PATROL to take more control over the issue, rather than passive monitoring on a log file.

The best thing about a command line tool is that you might already have one for the application. In a client-server architecture application, the client is a potential monitoring tool. Using the client, or perhaps a slightly modified client, can allow you to perform monitoring of the server with powerful functionality. For example, if the server can accept configuration change requests from the server, this client can be used for administrative commands. If the client can issue some form of dummy transaction requests, they can be used for performance and response time measurement. Examples of this idea abound in commercial applications, such as “sql-plus” for Oracle, or using a Web client to access a HTTP server.

Another command line tool you might already have is one that analyzes the application’s data files. For example, one that reads and dumps the information from the application’s internal

databases might be something you already have for debugging. This executable is also potentially a monitoring tool but with the same one-way limitations of text log files.

16.7.1 Command Line Tool Architectures

A number of potential architectures involve the three components of the application, PATROL, and the command line tool. You have two basic methods whereby data can be gathered from the application by the command line tool: batch and interactive.

The batch method is a tool that connects to the application, performs the required query or data gathering, and then completes. This implies a polling architecture for the Knowledge Module. Whenever PATROL's Knowledge Module configuration settings determine that more data is needed, the KM launches this command line tool to get the data it needs. The problem with a polling architecture is that the Knowledge Module may not notice a problem until the next polling period, rather than immediately.

The interactive method is a tool that connects to the application forever and remains there, either gathering data as it is generated by the application or responding to commands from the KM. For example, the simple case is the tool that connects to the application and then returns data as generated by the application. The advantage over the batch command line executable is the non-polling event-driven notification about problems from the application. As soon as the application is aware of a problem, the attached command-line tool is immediately returning data, and the Knowledge Module can immediately raise an alarm in PATROL. This is almost no different from the log file tailing model, especially in that you can also achieve a non-polling architecture with log file tailing.

The more complex case for the interactive command is where the tool accepts commands from both PATROL and the application interactively. By accepting data from the application, as in the simple case, the event-driven model is still achieved. But by allowing PATROL to send commands down the pipe to this command tool, PATROL can gather data as needed. However, this need for PATROL to request data is not very useful for monitoring. It is more useful for administrative commands such as user-driven menu commands or automatic recovery actions launched from the KM.

The basis of an interactive command is the use of PSL `write` to send commands to the command line tool and of PSL `read` to receive the results. This use of PSL `read` on the interactive command line utility does not prevent the KM from polling the application as required, because PSL `write` can still send commands down that pipe. Multiple PSL processes can access the launched utility, and hence one process can use PSL `write` to send an action for the utility to perform. The PSL `read` on the utility must be able to recognize any output it receives as being from a user-initiated action, or from an internally generated application status change. However, this is a matter of formatting the information, which is similar to the issues discussed for the log file format.

The disadvantages of the interactive command line tool are quite clear. Firstly, it is probably more complicated to implement. Secondly, it is constantly consuming system resources by

having an extra process around. Thirdly, it is potentially tying up a connection to the application and perhaps a license.

16.7.2 Advanced Monitoring with Command Line Tools

A command line tool can be used for monitoring purposes. However, in general, the benefits of a command line tool over log file tailing are not that great when considering only one-way monitoring. For example, a non-polling event-driven method of raising alarms can be achieved by either the log file tailing method or via a command line tool. Nevertheless, a few extra possibilities exist for monitoring involving command line tools.

The model for log file tailing is, of necessity, one where the application logs all possible data that might ever be needed for monitoring. By using instead a command line tool to extract data, the KM can decide what data it will need. If particular monitoring features of a KM are turned off by configuration, this data is not collected. No unnecessary writing of information to be ignored.

Another advantage is that this gives the KM control over the polling period for repeating parameters. If the application is repeatedly sending a piece of data to the log file, PATROL has no control over the period of data collection. However, if the data is returned only on request to the command line tool, the PATROL KM can choose the polling collection period according to the KM configurations.

As you can see, the advantages of a command line tool for monitoring over the log file tailing technique do not relate to the quality of the data. Instead, they relate to control over the reporting of the data. Conceptually, the same data must be collected by the application and printed, either to a log file or to the command line tool. Since significant extra development effort occurs in adding a command line tool compared to changing the log file reporting, you may not want to move to a command line tool purely for monitoring purposes. However, the use of a command line tool has the potential to make great leaps ahead in the area of administrative commands.

16.7.3 Advanced Administration with Command Line Tools

The main technological advantage of command line tools is that they need not be passive. They need not just collect data but can also change the application internally. This is powerful for both ad-hoc menu commands and automated recovery actions. Basically, the command line tool is a way that the PATROL KM can send an action request to the application. Some of the administrative commands that might be considerations for control by this command line tool include:

- Configuration settings—by allowing dynamic control of configuration settings via the command line tool, the PATROL KM can change configurations, and even dynamically tune the application.
- Actions on users—for example, you might want to force disconnect a user.
- Actions on data or databases—for example, you might want to remove or archive data.

An interesting conclusion arises around the need for batch versus interactive architectures for command line tools. The interactive command's advantages of non-polling event-driven monitoring of failures can also be achieved by log file tailing. Hence, all of the menu commands and recovery actions can be achieved via a batch non-interactive command line tool. A more effective use of development resources may be to create only a batch command line tool rather than an interactive one. Monitoring can be done via log file tailing, and administrative commands via the batch command line tool.

16.8 External Application API

A similar concept to a command utility is to provide a programming API by which a C program or similar program can access information or send commands directly to the application. This application API is launched by the PATROL agent to get information from the application.

The issues of using an application API are very similar to those of a command line utility. After all, any command line utility is really just a front end for such an API. Anything that can be done with one can be done with the other. The API can have read features for monitoring and write features for administration functionality, exactly like a command-line executable can have read and write capabilities. As for the command line tool, an application API has few advantages over the log file method for read-only monitoring, but greatly improves the administrative capabilities.

The main advantage of an application API over a command line utility is power and flexibility. An external application API is typically used by the PATROL KM by building it into an external child executable that is managed via a global channel. This process can make full use of the application's API set, rather than trying to scrape the text output of a CLI.

16.9 PATROL API Usage

The last issue is the use of the published PATROL API for monitoring and administration. The PATROL API is a published C-based API set for accessing information from an agent or triggering activities within the agent. Refer to the PATROL API Reference Manual from BMC Software for detailed technical information.

In general, the PATROL API is a very advanced feature when used for KM development. It is useful mainly in cases where the application is a complicated multiple-machine architecture. The main strength of the PATROL API over other methods is the ability to merge data from multiple KMs on multiple machines. This capability is used in the various ERP KMs covering SAP R/3, Baan, PeopleSoft, and Oracle Applications. These all use the PATROL API because of the distributed nature of the ERP application. All the major database monitoring KMs do not currently use the PATROL API, because the database server is not distributed across many hosts.

The main function of the PATROL API is the creation of events within the PATROL data space. This would seem the ideal way to get a non-polling event-driven architecture: as soon as the application finds internal troubles, it raises an event with PATROL, and PATROL does

something about it. This is true, but you can achieve this non-polling architecture much more simply through either log file tailing or an interactive command line tool, as discussed earlier. Both of these simpler alternatives rely on the fact that the PSL `read` function has an event-driven feature when reading from a pipe to a child process spawned by the KM.

Hence, if the application vendor provides enough power in the application “hooks,” whereby a command line utility or application-specific API has the power to register for notification of specific events, the integration of the PATROL API is not necessary to achieve immediate action on application changes. However, if the application does not have this capability in its command line or API hooks and does not have enough information in the log file, but it does have the ability to launch child processes based on internal events, the use of a process with the PATROL API is indeed powerful. These are two alternatives, depending on the facilities provided in the application.

Nevertheless, here is a brief description of how to embed the PATROL API into the application. The PATROL API is a set of C/C++ libraries that can be linked either into the application itself or into a separate executable. The application could raise events in PATROL via the PATROL API, or the application could launch a child process that contains the PATROL API. In the latter case, to use the ready-made executable called “`PatrolCli`” (the command-line interface to PATROL) may be easier, rather than building yet another executable.

An important issue to understand is that use of the PATROL API links PATROL-specific code into the application itself. If future PATROL releases do not support this API adequately, the application might have to be revised, rather than just revising the Knowledge Module.

And a final note about the PATROL API is that it has a lot of power for monitoring of distributed applications, as exemplified by its use in the PATROL KM for R/3 and other ERP KMs. However, these more advanced KM architectures are covered in Chapter 27.

16.10 Comparing Intrusive Instrumentation Methods

This whole discussion has become somewhat involved because of the sheer number of alternatives. Let me try to summarize the various alternative methods and then cover their advantages and disadvantages. The basic methods we have examined are:

- PATROL SNMP Toolkit
- Log file improvements and a Knowledge Module to monitor it
- Application API or client executable with a Knowledge Module to use it
- PATROL API used by the application to send events to PATROL

PATROL SNMP Toolkit to add an SNMP agent and MIB to the application has the following features:

- Monitoring capabilities unlimited.
- Administration features via SNMP “set” capabilities (and operating system actions).

- Polling architecture of SNMP “get” on the MIB.
- Non-polling architecture possible via SNMP trap sending.
- Other SNMP consoles directly supported via application MIB.
- PATROL SNMP Toolkit libraries linked into application. However, PATROL upgrades will not necessitate application changes, because PATROL uses SNMP access. Changes are needed only due to revisions of the SNMP Toolkit such as future SNMP standards support, and this is probably an optional choice if these new features are desired.

Log file changes combined with a Knowledge Module to monitor the file have the following features:

- Monitoring capabilities unlimited.
- Administration limited to operating system features.
- Almost fully non-polling design; poll only to detect file writes.
- SNMP consoles supported only indirectly via PATROLVIEW or the PATROL MIB.
- No PATROL code directly linked into application. PATROL upgrades cannot require application changes; only Knowledge Module changes at worst.

Application API or client command-line tool to access application combined with a Knowledge Module that uses the API or client (either with log file monitoring still used for the monitoring or replacing it) has the following features:

- Monitoring capabilities unlimited.
- Administration unlimited; API or client can provide administrative features.
- Event-driven architecture if application client or API supports asynchronous registration for application events; identical to log files if not. Asynchronous registration for the client executable can be as simple as it blocking until the application tells it something has happened, and the Knowledge Module is event-driven because the PSL `read` call wakes.
- SNMP consoles supported only indirectly via PATROLVIEW or the PATROL MIB.
- No PATROL code directly linked into application. PATROL upgrades cannot require application changes; only client and Knowledge Module changes at worst.

PATROL API used by the application to send events into PATROL has the following features:

- Monitoring capabilities unlimited.
- Administration capabilities limited to operating system features (the PATROL API itself does not help add administrative capabilities; another method such as a client or application API is needed).
- Event-driven architecture without question; this is the main advantage.
- SNMP consoles supported only indirectly via PATROLVIEW or the PATROL MIB.

- PATROL API libraries directly linked into application; PATROL upgrades might require application changes if the API is not fully backward compatible.

16.10.1 Levels of Instrumentation

The majority of the common methods for application instrumentation have been addressed. These methods range in complexity from the non-intrusive hands-off approach to the full use of the PATROL API. The levels of application instrumentation can be roughly categorized into:

- Non-intrusive monitoring using whatever features already exist.
- Log file standardization and extensions.
- SNMP instrumentation via the PATROL SNMP Toolkit.
- Building a command-line utility for monitoring and management.
- Creating an external application API for monitoring and management.
- Using the PATROL API to generate PATROL events and alerts without polling.

The most extensive techniques are probably not necessary for most application management needs. The use of simple log file techniques can yield an incredibly powerful monitoring solution. Surprisingly, the tailing of a log file can be an effective method and also offer an event-driven non-polling architecture. However, the use of reading a log file is limited in the administrative functionality it can offer, leading to the need for a command utility or API usage.

The various levels of instrumentation offer a natural set of milestones in the KM development process. The first level is a KM written for the current application using whatever features are available. The second milestone is extending the log file for advanced monitoring or the use of the PATROL SNMP Toolkit for monitoring plus simple administrative commands. The third milestone is creation of an application command-line tool or API to allow more advanced administrative control. The last milestone, which is rarely needed, is to use the published PATROL API for complete two-way interaction between PATROL and the application.

Integration with PATROL

The open PATROL architecture allows various advanced uses of PATROL via several methods of integration. The technical foundation for PATROL integration is a number of points of integration to the PATROL agent. The open PATROL 3.0 architecture has been further extended by a number of new integration points in the PATROL 3.1 and 3.2 releases.

Integration is required for a number of different uses of PATROL. Some of these integrations have already become part of the PATROL product suite via PATROLVIEW or PATROL-WATCH products. Other integrations require effort for customization in PSL, C, or SNMP.

Integration with PATROL is desirable when PATROL does not provide all the functionality desired within the application management infrastructure. It is also necessary when PATROL must be used to complement existing management practices. A common example is an existing set of tools already used for network management, to which PATROL adds database and application management. Some of the common needs for integration with PATROL include:

- Network management console—view PATROL events or objects within its GUI
- Help desk—forward PATROL events to a help desk for processing
- Web browser—view PATROL events or objects through a web interface
- Monitor an application, packaged or in-house, by writing a Knowledge Module
- Systems management integration—e.g., job scheduler, security manager, administrative tools, etc.

However, these are not the only requirements for integration. As you shall now see, the requirements for PATROL integration can be split into two categories.

17.1 Dual Nature of Integration

Two main types of PATROL integration exist, depending on whether data or events flow into or out from PATROL. This flow has been categorized as downstream and upstream integration. The differentiation is related to where PATROL is placed in the information stream with respect to the other component with which PATROL is integrated.

Downstream integration places PATROL down the stream of information flow that pushes events into PATROL. Some examples of downstream integration include:

- Writing a Knowledge Module to analyze an application and bring its data into PATROL
- Instrumenting an application via PSL, the PATROL API, or the PATROL SNMP Toolkit
- Accessing with SNMP an application's data via its MIB
- Converting some other form of events into PATROL events via PSL or the API

Upstream integration refers to placing PATROL high up the stream of information. This involves flowing PATROL events or data down the stream of information into some other repository or tool. Some examples of upstream integration of PATROL include:

- Sending PATROL data to reporting or capacity planning tools
- Sending PATROL data to archival systems or storage tools
- Forwarding PATROL events to a network management console or helpdesk
- Viewing PATROL events or data in some other console or web browser
- Reading PATROL parameter data using SNMP via the PATROL Agent's MIB

The needs for integration depend on how PATROL is used in the application management architecture. PATROL can be used in either an upstream or downstream capacity. You can integrate PATROL in any way via custom integration. However, when possible, a faster approach is to make use of the off-the-shelf integration components with PATROL that cover many of the above areas.

17.2 PATROLVIEW Console or Help Desk Integrations

A variety of existing integration modules for PATROL form the PATROLVIEW suite. These integrations support a variety of enterprise consoles and help desks. Some examples include:

- HP OpenView IT/Operations
- CA-Unicenter TNG
- Tivoli TEC
- Remedy ARS

PATROL does not attempt to be everything you will ever need, but rather tries to make easy plugging PATROL into the problem management scenarios that might already exist. For

example, the various PATROLVIEW products allow PATROL to be added to an existing network management scheme, and thereby add database and application problem management capabilities. When considering issues such as sending PATROL alarms into another console, help desk, or trouble-ticketing systems, the first place to check is PATROLVIEW.

The method used by PATROLVIEW is not all that magical. Indeed, you certainly can duplicate the PATROLVIEW style relatively easily for other products for which it does not already exist. The current predominant method is the use of the PATROL API, via what is called the PATROL Event Translator (PET) component. The PET connects to multiple agents at the same time via the PATROL API, and provides a front-end method of integrating with another software component on the local side. This is the technology that is packaged as a toolkit in the PATROL Toolkit for Console Integration available from the PATROL Developer Network.

In earlier times, SNMP was used to produce PATROLVIEW integrations. Although this method is possible and sometimes faster to achieve, it has some limitations because of the SNMP data model. The PATROL API-based technology has largely superseded the SNMP architecture for off-the-shelf PATROLVIEW integration components.

17.3 Web Browser Integration via PATROLWATCH

Web integration is one area that is already provided off-the-shelf with PATROL. Integration with web browsers such as Netscape Navigator and Microsoft Internet Explorer is available via the PATROLWATCH for Web Browsers component.

The architecture used by PATROLWATCH is quite similar to PATROLVIEW. In a sense, a web browser is just another console, although a much more programmable one. PATROLWATCH works via the PATROL API in a manner similar to the PET component. On the console machine, PATROLWATCH connects to multiple agents via the PATROL API, and then converts that into HTML and JavaScript to allow displays. Therefore, PATROLWATCH does not require an HTTP server on the agent machines at all. Instead, it talks directly to the agents via the PATROL API and does not use HTTP in its architecture.

Another related Internet issue is monitoring the various internet services. The PATROL Knowledge Module for Internet Servers monitors web servers, mail servers, ftp servers, and other network daemons. This concept is different from PATROLWATCH, but another piece of the puzzle for internet management issues.

17.4 PATROL Developer Network

The PATROL Developer Network (PDN) is a formal initiative surrounding PATROL with development effort from third parties. It is tasked with any development activity related to PATROL integration. Therefore, some of the key components that may be produced for PATROL integration include:

- Knowledge Modules for application monitoring via PSL
- PATROLVIEW-like console or help desk integration components via the API

All forms of PATROL integration are covered by the PDN. This fact means two things to potential developers or integration experts. Firstly, the PDN is the place to get tools and support for any type of development activity. Secondly, the integration component or Knowledge Module may have already been built by a PDN member and is available for purchase.

The PDN is the place to check whether such an integration component already exists. The PDN-developed integration components are listed in a PATROLMARK catalog on BMC's web site. In any case, the PDN will have the latest information on what off-the-shelf components have already been built, which may save you some considerable development time in an application management project.

17.5 Integration Building Blocks

The integration with PATROL is achievable via a number of components or technology integration points. These underlying components are used by the many off-the-shelf PATROL components, and are also published and available for custom integration development work. The main components are listed below:

- PSL and Knowledge Module features
- Event Notification Commands
- Recovery Actions
- Console-side state change actions
- PATROL API for external API-based access
- PET and the PATROL Toolkit for Console Integration
- SNMP trap sending from the agent
- PATROL MIB for external SNMP access
- Command-Line Interface
- History Loader
- PATROL SNMP Toolkit
- PatrolLink

All these components serve different purposes. Any integration project is not likely to use all of these components in the same architecture.

17.5.1 PSL and Knowledge Modules

The PATROL Script Language (PSL) and other technology used in a Knowledge Module is not usually thought of as an integration point. However, it is really one of the strongest. PSL is a powerful, fully capable programming language that runs within the PATROL agent. This can be used to integrate data from the outside or send data from the inside. Thus, PSL can be part of a downstream or upstream integration sequence.

The main use of PSL is in the development of Knowledge Modules for application monitoring. This is a form of downstream integration, where PATROL grabs data and information from the application. Some of the common areas of Knowledge Module development include:

- Operating system monitoring
- Database monitoring
- Application monitoring (packaged and in-house)
- Middleware monitoring
- Groupware monitoring
- Internet server, email server, and firewall monitoring

Many examples of such Knowledge Modules are already part of the PATROL suite. This is the main area of value of PATROL, in that it has off-the-shelf Knowledge Module components that know what is important to gather or alarm about in a database or application. In this way, PSL is the primary integration point between the monitored database or application and the PATROL agent.

The traditional form of a Knowledge Module is a non-intrusive monitoring component. It does not require modification of the application to achieve monitoring, but instead relies on whatever interfaces are available or whatever footprints the application leaves in the operating system. However, since PSL is a procedural programming language, you can use PSL to build an instrumentation-based Knowledge Module. The instrumentation can be quite simple, such as better information written to an application log that is read via PSL, or it can be quite complex via the PATROL API in combination with PSL. In this way, you can achieve any level of application monitoring and administrative capabilities via different uses of PSL.

17.5.2 Other Uses of PSL

Knowledge Modules are not the only integration usage of PSL. It is a complex programming language that can be used to turn the agent into any part of a desired architecture. Here are some other examples of using PSL as an integration point:

- PSL can receive SNMP traps from some other software component or hardware device, and then process them or forward them along.
- PSL recovery actions can send pager, email, or SNMP trap notifications.
- PSL event notification commands can also send pager, email, or SNMP trap notifications.
- PSL recovery actions or event notification commands can be used for more generic notification methods, such as using the CLI of another framework to send it events, as an alternative to a PATROVIEW-style integration.

And these are not the limit. As a free-form programming language, PSL can be used in any way that the programmer can imagine.

17.5.3 Recovery Actions

Recovery actions were originally added to allow automated corrective actions to be performed by a Knowledge Module. They are certainly used for this, but have also found a use in notification style integrations. Recovery actions can forward PATROL alarms to pagers, email, SNMP traps, or some other integration framework via its CLI.

Recovery actions are added to each parameter for which alarms are critical. The PSL code that forms the recovery action is then used to send to the desired notification scheme. Typically, the PSL does some pre-processing such as determining context information, and then calls an external system executable. For example, it would build a message and then launch a paging or email client to send the message. In other cases, the PSL remote functions can be used to send the data or event to a centralizing agent. Another use is to have the PSL SNMP functions send an SNMP trap with the message.

17.5.4 Event Notification Commands

An event notification command is a PSL script that is executed whenever the agent creates an event. Although usable in many contexts, one of the main uses of these commands is the handling of state change events that are generated by PATROL alarms. This usage is similar to recovery actions but can be advantageous in terms of simpler configuration and a more general solution.

Event notification commands are PSL that typically build a message and then try to send it. As for recovery actions, the message is typically used to launch an executable that sends a page or an email. The remote PSL functions can also be used to send the message to another PATROL agent, and the PSL SNMP function can be used to emit an SNMP trap from the PATROL agent.

17.5.5 PATROL SNMP Toolkit

The PATROL SNMP Toolkit is an instrumentation toolkit for in-house applications. It allows the instrumentation of applications with information published through an SNMP MIB. This makes the PATROL SNMP Toolkit one possible method of achieving intrusive instrumentation-based application management for in-house developed applications. It is a development toolkit through which instrumentation can be added via some extra development work.

The usual scenario occurs for an in-house application written in C or C++. The instrumented version of the application links with the code libraries that are part of the PATROL SNMP Toolkit. This action adds an SNMP subagent to the application code, which can then communicate with an SNMP master agent on the same machine. A MIB is also generated to export an SNMP interface to internal application values, such as performance or availability metrics. Writable values of the MIB permit remote SNMP-based configuration of the in-house application. An SNMP master agent is available as part of the kit, or any other alternative SNMP master agent can be used, provided it supports the SMUX protocol.

Instrumentation is the sole reason that the toolkit can be used. This makes it an alternative to use of the PATROL API and a Knowledge Module, which is also capable of achieving an instrumented application. However, the toolkit clearly has much stronger SNMP support, which allows it be used with or without a PATROL agent. It can also be used in conjunction with a Knowledge Module, which must be written using the PSL SNMP functions to read the newly created MIB values. The PATROL SNMP Toolkit can automatically convert the MIB to a KM that uses the PSL SNMP functions to read application data into KM parameters. This conversion yields the combined benefits from an open SNMP interface and a continuously monitoring Knowledge Module.

17.5.6 PATROL MIB and Agent-Side SNMP Support

The PATROL agent has a fully featured SNMP subagent component. This allows the agent to act according to all the SNMP requirements. It can browse other MIBs via SNMP set/get/get-next requests, it can receive traps, and it can send traps. All these three forms of functionality are available via PSL.

In addition, the agent has its own MIB. This allows other non-PATROL applications to examine the data gathered by the agent and its KMs via SNMP get requests. The PATROL MIB is quite complex, with a number of tables allowing different forms of access to the KM data. Some of these tables are dynamic, since it does not know beforehand how many instances each KM will discover. Therefore, the OID for these tables is not fixed for a given object, but depends on the dynamic discovery of objects by KMs. Some of the other MIB tables are static, which means that provided you know the full name of the object to be examined, it can be found via the same OID on any occasion.

Be sure to understand the distinction between the PATROL SNMP Toolkit and the agent's built-in SNMP support. The purpose of these two SNMP components is quite different. The PATROL SNMP Toolkit is a kit specifically aimed at achieving SNMP-based application instrumentation of in-house developed applications. The PATROL agent's SNMP support is completely different, and is simply a part of the PATROL monitoring infrastructure. This SNMP support in the agent can be used for any issue, including integration via SNMP trap sending. In fact, the PATROL agent's SNMP support is actually based upon the PATROL SNMP Toolkit. Just as any other developer team would use the PATROL SNMP Toolkit, so too does the development of the PATROL agent achieve its SNMP features through use of the toolkit. Technically, they are therefore similar, but in their expected usage, they are quite different.

17.5.7 SNMP Monitoring via PSL

A particular example of the agent-side SNMP support is the PSL features. These include a full suite of capabilities including SNMP management requests (i.e., get, set, get-next, walk, etc.), SNMP trap receiving, and SNMP trap sending.

These full-featured capabilities are probably one of the most underutilized areas of PSL. A PATROL Knowledge Module can examine an SNMP MIB or register to receive traps from an

SNMP-enabled component. For example, via PSL, you can write a Knowledge Module that monitors any SNMP device. This could be a hardware network component such as a router or an application that is instrumented with an SNMP MIB.

A Knowledge Module can even act as an SNMP mid-level manager, since it can analyze values, receive traps, and then forward them along. PSL can be used to gather values, process them, and receive traps. According to the desired logic of the SNMP middle manager KM, it can then send SNMP traps or alternatively forward standard PATROL alert events to PATROL or to a PATROLVIEW-integrated enterprise management console or help desk.

17.5.8 Command Line Interface

The CLI is available via the “PatrolCli” executable that is part of the standard PATROL product. This executable is typically part of each agent package. The CLI has most of the capabilities of the PATROL API with both read and write capabilities. It can access the PATROL object hierarchy to read current values and statuses, and also the repository of agent events. Also, facilities to change aspects of the agent configuration are available, if permitted to do so, and also to generate events within the agent.

The CLI has both interactive and batch components. The interactive component is mainly useful for human interaction, such as when logging into a machine from home to see quickly what is happening. The batch components of the CLI allow it to be used in sophisticated scripts for various purposes. In this way, it is a form of integration point similar to the API, but does not require as sophisticated programming knowledge to use it.

17.5.9 PATROL History Loader KM

The PATROL History Loader KM is an architectural component that is layered on top of the agent via PSL and a Knowledge Module basis. Although implemented as a Knowledge Module, it is not really a true Knowledge Module in the sense that it does not monitor anything. Its functionality is to read each agent’s parameter history values, package them into files, and send them over the network to a central agent. This central agent receives historical values from multiple agents and merges these values together. This combined data is then exported to a text file or to a variety of SQL-based commercial database formats. All these components are implemented as Knowledge Module components. The data collector and propagator components sit on each agent, and the loader component sits on the central agent.

The PATROL History Loader KM is predominantly used for data archival and reporting. It allows data measured by the various Knowledge Modules to be centralized into a single commercial database. This database is then used to generate reports via one of the many database reporting tools. Thus management reports can be produced, and trending analysis is possible.

17.5.10 PatrolLink for PATROL Console Integration

The PATROL Console has some programmable capabilities. The PatrolLink command is useful to launch the PATROL Console in context, and is often used this way for some of the more

sophisticated PATROLVIEW integrations. However, PatrolLink is more sophisticated than just launching the console. A variety of commands affect the desktop of an already open console, or launch a new one. For example, you have control over containers and shortcuts, and the ability to launch graphs and charts.

PATROL 3.2 also has a simplified form of this in the PSL “desktop” function. This allows a PSL script in the agent to use a simple PSL command to control the console desktop. Previously, it would have been required to inelegantly launch the PatrolLink executable from the agent side.

17.6 Integrating Advanced Event Management Components

Several types of complementary advanced management functionality are available as third-party tools. Some of the most common forms include:

- Event correlation tools
- Root cause analysis tools
- Business process and high-level management views

Two possible types of requirements are: pulling PATROL events out of PATROL, and putting new higher-level or correlated events back into PATROL. In some cases, only the first type of management functionality is desired as the event correlation tool handles further event processing.

Pulling events out of PATROL agents is best achieved via the PATROL API. This allows registration for asynchronous notification from PATROL about new events. The advantage of this is that a scalable event-driven architecture is possible, rather than a less scalable polling architecture. This use of the API is best seen in the various PATROLVIEW products that pull events from PATROL and push them into a framework console or help desk. This use of the API also does not require any configuration changes to the PATROL agents.

The PATROL CLI can also be used in a way similar to the PATROL API. It is a client executable that wraps a lot of the API functionality. It can be used as a quicker way to get an integration working via scripts instead of C/C++ programming. Although it is slightly less efficient, this is only on the client side, since the same underlying agent API is used with the same agent efficiency as the PATROL API.

Another form of event-driven architecture is to modify the PATROL agents via configuration so that they send the events to the advanced console. One way to do this is via the agent’s SNMP traps, either built-in traps or via PSL customization. Another way is to use some form of client executable that registers the events into the other framework. These methods are quite efficient and scalable, but require some agent-side configuration to use SNMP traps, recovery actions, or event notification commands.

Some other options involve polling architectures. Although not scalable, they can be options for small-scale monitoring. Some of these integration methods include polling via the agent's SNMP MIB, the API, or CLI.

These are all methods of extracting PATROL events. The other side is what the event management tool does with the new events. An event correlation system may produce compound events, and a root cause analysis engine can produce diagnostic events that identify the root cause. In some cases, these new events will stay inside the advanced event management component, because it may handle the further processing of the event, such as notification methods. In other cases, to push the resulting event back into PATROL is desirable. The simplest method to achieve this is via PSL in a Knowledge Module, which analyzes the data coming out of the event management tool. Another alternative is to push the event into PATROL via a call to the PATROL API or CLI from the event management tool.

17.7 Integrating Distributed Systems Management Tools

Many forms of Distributed Systems Management (DSM) tools exist. The idea of proactive monitoring applies just as well to these tools as it does to business applications. Many of the various DSM tools can raise alerts that should be funneled into the PATROL event management system so that they are either handled directly or forwarded via PATROLVIEW to a help desk. Some of the DSM tool areas include:

- Job schedulers
- Security monitoring tools
- Backup and recovery tools
- System administration tools
- Database administration tools
- Software distribution tools

The basic method of monitoring any of these DSM tools is to place a Knowledge Module on the server side of these tools. A Knowledge Module can monitor the various tool components via PSL, such as the daemon processes and their log files. In a sense, these tools are just like any other form of distributed application that requires monitoring. However, some of these types of tools have particular aspects that distinguish them.

Integration of PATROL with a job scheduler involves a few special wrinkles. Firstly, the scheduler's basic daemon process must be monitored for availability on the server. Secondly, the status of jobs needs to be monitored according to what status the job scheduler considers them to be. For example, the scheduler might write an error to a log file if a job is failing.

Security monitoring tools are a good example of the value of a Knowledge Module. These tools are typically scheduled tasks to detect modified files, web documents, or other security incursions. Once these problems are detected, they need to be brought to the attention of the administrator. A neat way to achieve this result is via a Knowledge Module that processes the

security tool's output, log file data, or some other interface. It can then raise PATROL alerts, which are propagated according to the application management architecture. A good example of an integration between PATROL and a security monitoring/management tool is the PATROL-MARK product built by Axent (www.axent.com).

The PATROL suite contains a number of examples of Knowledge Module integration with DSM tools. The PATROL product set consists not only of application management tools, but also of backup and recovery, database administration, and performance optimization tools. The PATROL Recovery Manager, a backup and recovery tool, uses the PATROL agent to schedule its internal jobs and monitors its own components via a Knowledge Module. Similarly, the PATROL-DB suite of database administration components is monitored by a PATROL component for any dangerous failure on its server side.

Building Console Integrations

PATROL integrates with enterprise frameworks, help desks, and SNMP management consoles via the various PATROLVIEW components. These provide a suite of off-the-shelf integrations for many major third-party products. However, not all consoles are supported in this way, and the alternative is to build your own PATROLVIEW-like integration. This is the second major form of PATROL-related development that is supported by the PATROL Developer Network (PDN), after Knowledge Module development.

The first alternative to building your own is to examine whether another third-party is providing an integration component even if one is not directly available with PATROL. The PDN may already have a member company that is offering exactly the component you require.

Performing the development yourself is necessary if no other components exist. The PDN provides the tools you need to perform this integration in the PATROL Toolkit for Console Integration. Not all integrations are alike, but various common types of functionality are required, including:

- Event forwarding of PATROL events or alarms to the console
- Display of PATROL objects native in the third-party console displays
- Launching of the PATROL operator console

These are to some extent in order of development difficulty. Event forwarding from PATROL to another console or help desk is the basic offering of the PATROL Toolkit for Console Integration. It is also possible via the conversion to SNMP traps or trouble ticket formats. The representation of PATROL objects natively within another console infrastructure often involves more detailed usage of the console's internal API to create the internal objects. Launching the PATROL console furthermore requires the existence of the native objects and then the

integration of actions on these objects converted to PATROLINK commands that launch the PATROL console.

The desired functionality affects the technology used. Some of the basic PATROL integration points include:

- PATROL API
- PatrolLink
- PATROL Toolkit for Console Integration
- SNMP Traps
- Recovery Actions
- Event Notification Commands

A simple event forwarding integration typically uses recovery actions or event notification commands combined with SNMP traps or command-line executables to send data in a one-way stream to the console or help desk. A more advanced form of event forwarding integration avoids SNMP and instead uses the PATROL API or its generalization via the PATROL Toolkit for Console Integration.

The next level is the representation of PATROL objects within the new console view. This is usually more of an issue with the console's API set rather than the choice of PATROL integration method. However, usually the API or PATROL Toolkit for Console Integration technology is used here.

The functionality of PATROL console launching is a very specific integration point. The only possibility here is to use the PATROLINK command to start the console.

18.1 Event Forwarding Integrations

The PATROL Toolkit for Console Integration and its tools can be used for any type of integration. The toolkit aims at the required functionality of forwarding PATROL alarm events to another console. Also, a number of other alternatives do not require C/C++ coding, which would be needed for the API or the PATROL Toolkit for Console Integration. Event forwarding integration is possible using these various simple methods, because the third-party console does not have to maintain a list of PATROL objects and their states. Instead, it simply receives information about PATROL alerts as they occur. This is a one-way integration from PATROL to another console. The technologies that can be used to achieve this include:

- Recovery actions added to the most important parameters
- Event notification commands defined in the Standard Event Catalog

These two integration points can use a number of messaging methods by launching the appropriate client or PSL call. Some of the common methods include:

- SNMP traps sent via PSL SNMP function calls.
- Client executables launched from PSL that create third-party messages or events.

SNMP trap integrations can actually use agent built-in state change traps, in addition to recovery actions or event notifications. Chapter 19 addresses these SNMP integration methods.

18.2 PATROL API, PEM API, and PET

This area of PATROL is three-letter acronym heaven. You have PEM, PET, PEA, PTK, and API, just to name a few TLAs. These translated become PATROL Event Manager, PATROL Event Translator, PATROL Event Adapter, the PATROL API, and the PATROL Toolkit for Console Integration.

The API is the basic layer in all of these. The API is the main mechanism for pure programmatic interface with the PATROL agent. This is for developers who prefer C/C++ to scripting. Historically, it has been used for many PATROLVIEW integrations, as they moved away from SNMP. The API is also the underlying technology for the PATROL CLI and a number of inter-agent PSL functions. The API has been around since the 3.0 early releases, and was significantly enhanced in the 3.0.15 release.

The nomenclature regarding the PATROL API has become a little confusing over the years. Conceptually, at least three names mean basically the same thing:

- PEM API
- PATROL API
- PATROL Toolkit for Console Integration

The PATROL Event Manager API or PEM API is the older name because the API was originally based around events. As the API became more powerful (notably in version 3.0.15), it became the PATROL API. Finally, the release of the SDK via the PATROL Developer Network provides the “PATROL Toolkit for Console Integration,” which is a set of tools and documentation that aid the developer in building API-based console integrations using the PATROL API or using the PET.

So how does the PATROL Event Translator (PET) fit into all this? The PET is another layer on top of the PATROL API that handles connections with multiple agents. This forms the basis of most PATROLVIEW integrations, since they must talk to many agents. You would use the API directly rather than PET if you were doing single agent connections, or if the PET did not provide access to the data you need. You would use the PET if you were doing a PATROLVIEW-like integration that is pulling events from many agents. PET technology is exactly what the PATROL Toolkit for Console Integration makes available.

Another older component is the PATROL Event Adapter (PEA). This is merely a precursor to the PET that was used in an early Tivoli integration. It is superseded by PET and the PATROL Toolkit for Console Integration.

18.3 PATROL Toolkit for Console Integration

The PATROL Toolkit for Console Integration is a toolkit that aims to simplify a particular class of integrations. This class is integrations that involve event forwarding integrations. The aim is to make PATROLVIEW-like integrations with third-party enterprise network consoles or help desks.

The PATROL Toolkit for Console Integration architecture involves the use of the lower-level PATROL API to get events out of multiple agents. This handling of multiple agents is where the toolkit adds value over the API. It uses the PET technology to encapsulate many of the details of handling multiple API connections to agents. Configuration issues for these connections are handled by the PET configuration component, taking away a lot of the low-level grunt work involved in converting the raw PATROL API calls into a useful multi-agent event forwarding integration.

The PATROL Toolkit for Console Integration is a set of C libraries, header files, and the associated APIs. These API calls have the PTK prefix from an earlier project name: PATROL Toolkit. These PTK API calls are actually wrappers for both the PATROL API calls and the underlying PET technology. However, whether the PTK libraries permit direct calls to the underlying APIs is not clear.

The toolkit supports UNIX and Windows NT development platforms. The APIs can be used in both non-GUI base applications, Windows NT applications, or in UNIX Motif X Windows applications.

The initial 1.0 release of the PATROL Toolkit for Console Integration is a safe and fast event forwarding toolkit. It is not aimed at other functionality, and notably does not even have a method to send an event to the agent. Whether the raw API can be used directly from the PATROL Toolkit for Console Integration to achieve this need, is not clear. However, one-way event forwarding integrations do not require this functionality.

18.4 PATROL API

The PATROL API is the most fundamental PATROL integration point. It is for advanced developers and is a raw tool that is both dangerous and powerful, whereas the PATROL Toolkit for Console Integration is safer and hides the many development details.

Fundamentally, the PATROL API is a published, documented C function API set that allows local and remote access to the PATROL agent. It can be used in both C and C++ applications, but it does not use any of the C++ class features. The API supports both “read” and “write” actions against the PATROL agent. These effectively allow the API to be used in both upstream and downstream integrations. The API can be used to pull data from PATROL or push data or events into it.

The API can operate both locally and remotely. The PATROL UDP and TCP communications layer is underneath the API. This permits the API to connect to an agent remotely, much like a console does. You can remotely send or receive events, or perform other API actions. The functionality is identical for the “P_{emn}” set of API functions for local and remote actions.

The read actions allow features such as gathering parameter data for reporting, or waiting for alarm events for console or help desk integrations. A variety of “read” actions determine the state of an object or receive an alarm. The read operations can be performed in a synchronous or asynchronous manner. By registering to receive events in an asynchronous manner, the API can efficiently wait to receive an event from the agent.

The write operations are somewhat limited compared to what a Developer Console can perform, but can be used to perform agent configuration changes such as tuning the agent. The write operations can also be used by advanced instrumentation methods to pump events into PATROL in an event-driven manner using the “PemnSend” API. They allow more efficient Knowledge Modules that do not need to poll for results from the application, but can simply wait for an API connection that creates an event.

Support is available for asynchronous queries to the agents, where an API call registers to be informed by the agent of new events. The call to use is “PemnReceive.” This causes the program to wait for events coming down the communications line. This event sending method goes through the same persistent filters as with any events sent to clients of the agent.

Another component of the API is the methods for reading an event log. These are supported via the “Pemr” family of functions. This allows local access to the circular event log files, a programmatic equivalent to what the `dump_events` command offers.

One common pitfall in the usage of the API is that it is not thread safe itself. This means that if API functions are called from multiple threads, big problems occur. This problem appears most commonly when building Windows NT clients using the API. Currently, the only solution is to make all API calls from one thread, and perform thread synchronization via other threads making thread-safe requests to that “PATROL API” thread.

18.5 API Main Loop Integration Issues

The PATROL API takes control of the executable main loop at its call to `PemnLoop`. This characteristic can create problems in applications where the application code also needs control. The PATROL API requires main loop control to detect incoming messages from the agents on socket file descriptors. The API main loop places the executable in a sleeping state, typically via a `select` system call, or places its equivalent on platforms without such a system call. However, a common requirement for large integration components is that they also perform some other tasks, even if nothing is incoming from PATROL agents. Naturally, a third-party console has to perform some non-PATROL activity. This is clearly difficult to do if it gets only CPU time back from the PATROL API when a PATROL message arrives. The solution is to share the CPU between the PATROL API and the third-party console using a number of techniques:

1. Threads—separate PATROL API and other issues into separate threads.
2. Processes—separate system processes for the API and other components.
3. Replace the PATROL API main loop via handlers replacement.

The threading issue may run afoul of the already-mentioned thread-safe pitfall of the PATROL API. However, it can be workable if done safely. One thread must handle all the API calls, and other threads in the application must use mutexes to pass data into or out of the thread handling the PATROL API calls. This approach wraps the unsafe API libraries into a multi-threaded application. Future releases of the client-side PATROL API may be thread-safe.

The second alternative is to build the integration component using the PATROL API into a separate executable process, often possible for third-party consoles that provide sophisticated integration points.

The final alternative is to merge the PATROL API and other code into one executable by replacing the API main loop with your own. This is documented in Appendix B of the 3.2 PATROL API documentation, but is also discussed here briefly. To replace the main loop requires understanding of the one used by the PATROL API, which is the “handlers” library. It is so named because it is a set of handlers for file descriptor activity due to socket messages. The PATROL API ships with various sets of handlers libraries:

- `libxhandlers.a`—UNIX X-Windows program handlers
- `libhandlers.a`—standard handlers for UNIX
- `handlers.lib`—standard handlers for Windows NT

You would use `libxhandlers.a` as the main loop if your program is a UNIX X-Windows program. Otherwise, the standard `libhandlers.a` library is used for UNIX, and `handlers.lib` is used for a Windows NT executable. These are the libraries that must be replaced with your own when you are replacing the API’s main loop. This replacement requires the re-implementation of the handlers interface as defined in the “`handlers.h`” header file. The new self-implemented handlers routines are linked rather than using the PATROL `handlers.a` or `handlers.lib` library file.

Similar main loop integration issues also apply to the PATROL Toolkit for Console Integration and its `PTKLoop` function. However, whether this handler library replacement also works with PTK calls is unclear.

A good example of the need to merge main loops is a collector daemon that is attempting to collect events from not only PATROL events, but also other inputs from elsewhere. The problem is that `PemnOpen` and `PemnReceive` place the executable in a waiting state until action is on the communication file descriptors. This approach works fine for an application dedicated to receiving events, such as `PET` or `PATROLVIEW`, but is not so good if the event receiving is part of another executable that must do other work. The example of a multi-faceted event collector requires that the other application code also get control.

The problem is how to get the main loop of the other half of the executable some cycles when the API side is waiting. The PATROL API has no way to perform a polling check of the API’s file descriptors rather than allowing it to sit in a wait state. Hence, the best solution is to use a separate thread, but this potentially runs afoul of the already mentioned lack of thread-safe API calls.

C H A P T E R 1 9

SNMP Integration

Although SNMP is not the base technology for application management, it still has a great many uses in the discipline. SNMP is useful to send notifications to existing SNMP management consoles or for the management of SNMP-enabled devices. This chapter discusses a number of uses of the SNMP support in PATROL for use within an environment. The major topics covered include:

- SNMP trap sending—problem notification to other SNMP management consoles.
- SNMP trap receiving—forwarding or handling traps within the PATROL agent.
- SNMP data extraction—reading data from the PATROL MIB.

The PATROL agent has full SNMP support in that it can act as both an SNMP master agent or subagent. The SNMP support is so deep that it has PSL built-in functions for MIB access, trap receiving, and trap sending. This chapter shows how these capabilities can be used to plug PATROL into the desired management infrastructure.

19.1 SNMP Trap Integration

One of the most popular methods of notification about critical events detected by PATROL is sending SNMP traps to some form of event management console. PATROL can be used to send SNMP traps to a number of third-party products, including but not limited to the following:

- IBM NetView (for AIX RS/6000 and mainframe)
- HP OpenView IT/Operations
- CA-AgentWorks Domain Manager
- COMMAND/POST

- MicroMuse NetCool/Omnibus
- Other SNMP enterprise consoles
- Other help desk systems

In some of these cases, PATROLVIEW products are also available for integrating with these products. The choice between PATROLVIEW and SNMP trap integration is discussed on page 269.

19.2 Three Methods of SNMP Trap Sending

PATROL has so many SNMP integration points that you have three main ways to send SNMP traps from the PATROL agent:

1. Default agent built-in state change traps.
2. Event catalog—event `TRAP_SEND` and `NO_TRAP` settings in the event catalogs.
3. PSL—writing a PSL recovery action or editing the standard event catalog to have a PSL event notification command that uses the PSL SNMP built-in functions to send any kind of trap.

Actually, a fourth method of sending traps from the agent exists, which involves launching a child executable to launch a trap. For example, via PSL `system` or `execute` functions, you can run an “`snmpttrap`” executable program on the agent side. However, using the built-in PSL functions is significantly more efficient.

And note that a method of sending SNMP traps from the PATROL Console, rather than the agent, is also available. This is achieved by KM state change actions, and is discussed on page 267.

All these SNMP trap sending methods have various advantages and disadvantages that will be discussed below. Table 19-1 summarizes some of the issues.

Table 19-1 SNMP trap method comparison

SNMP Trap Feature	Agent Traps	PEM Traps	PSL Traps
Requires configuration of out-of-box install	Yes	Yes	Yes
Any trap format possible	No	No	Yes
Enterprise OID can be changed	No	No	Yes

Table 19-1 (cont'd) SNMP trap method comparison

SNMP Trap Feature	Agent Traps	PEM Traps	PSL Traps
Different OID possible for each KM class	No	No	Yes
Trap text can be configured/changed	No	Yes	Yes
Number of different trap formats possible	2 (version 3.0 & 3.1)	2 (version 3.0 & 3.1)	Unlimited
Methods of controlling format of these traps	Agent configuration settings.	Event catalog settings and agent configuration.	PSL coding (almost unlimited options)
Trap sending caused by what situations?	Parameter, instance, and application state changes.	Event generation of an event in the associated event class.	Any method of PSL execution.

19.3 Built-in Agent SNMP Traps

The PATROL agent has a number of internal SNMP traps that can be enabled. These traps occur when a parameter goes into an alert state due to an alarm threshold violation, or if an instance changes state. The agent sends these SNMP traps automatically.

In PATROL version 3.0, these traps occur on every state change. The traps occur when a parameter, instance, or application changes state. Therefore, when a parameter alarms, it sends as many as three traps (one for its own state, one for the instance state change, if required, and one for the application class state change, if required).

When a parameter returns to OK from an alarm state, it will also issue a trap indicating the new OK state. This too can lead to three traps in the PATROL 3.0 style as the OK state gets propagated up the levels.

Another issue is the initial startup state change trap on creation of an instance or the first valid value of a parameter. Typically, this is a state change from VOID to the new state, which can be OK, WARN, OFFLINE, or ALARM.

In PATROL version 3.1 and later agents, the times when the agent sends a built-in agent trap are more tightly controlled via settings in the standard event catalog. Since each event's trap properties can be controlled individually, you have more control over whether a particular situation causes an SNMP trap.

19.3.1 PATROL 3.0 and 3.1 Trap Formats

PATROL 3.1 saw the introduction of a new SNMP trap format, but backward compatibility with 3.0 was offered. Therefore, two SNMP trap formats are available. Note that no new 3.2 trap format is available and the choice between those from 3.0 and 3.1 remains for 3.2 agents.

The version 3.0 PATROL trap format looks quite simple with just the object pathname, and the new status of the object. Some example dummy traps might look like:

```
/appl/inst/param ALARM
/appl/inst WARN
/appl OK
```

Here is the text from some v3.0 traps from a regression testing KM:

```
/snmpptest/snmpptest OK
/snmpptest/snmpptest/consumer ALARM
```

The version 3.1 PATROL trap format contains more information than the 3.0 format. This information is read from the event catalog text fields for the event associated with the trap. This new format is part of the PEM SNMP integration support in 3.1 and later agents. Here is the text from some real-life agent v3.1 built-in traps from an SNMP regression testing KM. As you can see, this text is coming from the event description fields for the associated event classes, as specified in the event catalog file.

```
Alarm #2 of global parameter 'consumer' triggered on
'snmpptest.snmpptest'. 2 <= 2.00 <= 2
Alarm #1 of global parameter 'consumer' triggered on
'snmpptest.snmpptest'. 1 <= 1.00 <= 1
Alert on 'snmpptest.snmpptest' from global parameter 'consumer' cancelled; exception no longer exists.
```

The agent configuration settings for version 3.0 agent trap formats are:

- “/AgentSetup/pemIssueV30traps” set to “yes”
- “/AgentSetup/pemIssueV31traps” set to “no”

The agent configuration settings for version 3.1 agent trap formats are:

- “/AgentSetup/pemIssueV30traps” set to “no”
- “/AgentSetup/pemIssueV31traps” set to “yes”

Note that the results from setting both these configuration variables to the same value are unclear. I recommend setting both at the same time to consistent and opposite values. A single Boolean agent configuration variable would indeed make sense, but the PATROL agent has two.

The PATROL 3.2 agent is defined out-of-the-box to issue events in the 3.1 trap format. The chosen setting can be preset prior to agent distribution by editing the “config.default” agent configuration file. Alternatively, these settings can be changed after distribution via the pconfig, xpconfig or wpconfig tools.

19.4 PEM Built-in SNMP Trap Support

PATROL offers an SNMP enhancement to the PATROL Event Manager (PEM). The change allows the individual association of SNMP trap sending configuration settings with each event class. This is true for both the Standard Event Catalog and any KM-specific event catalog created for a new KM.

The main SNMP feature added to the PEM support is a setting to NO_TRAP or SEND_TRAP for each event class. This specifies whether the agent will emit a trap when this event is created. By choosing which event classes have traps enabled, you have finer control over the number of traps and the causes of traps.

You can control whether these traps occur in v3.0 or v3.1 PATROL traps formats. However, little other control is available over the other details of the format of PEM SNMP traps. For example, you have no control over the event-specific subid with the trap, or the enterprise id.

The 3.2 agent’s default settings for the SNMP PEM support in the file “stdEvents.ctg” is enabled only to SEND_TRAP for events 9 and 11. Event class 11 occurs when a parameter violates an alarm threshold (i.e., goes into warning or alarm), and event class 9 occurs when a parameter goes back to OK state from an alert state.

The PATROL Event Manager subsystem also has a second set of configuration settings, the agent configuration variables for the PEM. These include filters to suppress creation of events, SNMP trap sending, and various other features of the PEM SNMP functionality.

19.4.1 Agent Configuration for Built-in SNMP Traps

The 3.2 agent does not send SNMP traps for the built-in state change traps by default. Some agent configuration is necessary. The list of agent configuration variables related to SNMP trap sending includes those in Table 19-2 below.

Table 19-2 SNMP trap sending agent configuration variables

Agent Configuration Variable	3.2 Default	Purpose
“/snmp/piV1m_list”	“”	List of SNMP enterprise managers to which to send agent built-in traps.
“/AgentSetup/pemIssueV30traps”	“no”	Issue v3.0 traps?
“/AgentSetup/pemIssueV31traps”	“yes”	Issue v3.1 traps?

Table 19-2 (cont'd) SNMP trap sending agent configuration variables

Agent Configuration Variable	3.2 Default	Purpose
"/snmp/trap_port"	"162"	Port on which traps are sent.
"/AgentSetup/pemSnmSupport"	"yes"	Is the PEM SNMP trap sending enabled?
"/snmp/trapConfTable"	"no"	Issues traps to the managers of pre-configured list? (snmp_trap_register_im)
"/snmp/trapMibTable"	"yes"	Issue traps to the managers listed in piV1m_list configuration variable?
"/AgentSetup/pemPFSnmpStartTime"	""	Filter traps based on time (start time).
"/AgentSetup/pemPFSnmpEndTime"	""	Filter traps based on time (end time).
"/AgentSetup/pemPFSnmpStatusMask"	"O,A,C,E,D"	Filter traps based on event status (Open, Ack, Closed, etc.).
"/AgentSetup/pemPFSnmpTypeMask"	"I,S,E,W,A,R"	Filter traps based on event type (Info, State change, etc.).
"/AgentSetup/pemPFSnmpNseverity"	"1"	Filter traps based on event severity (minimum allowed).
"/AgentSetup/pemPFSnmpNode"	""	Filter traps based on hostname of node.
"/AgentSetup/pemPFSnmpOrigin"	""	Filter traps based on object.
"/AgentSetup/pemPFSnmpPattern"	""	Filter traps by pattern.
"/AgentSetup/pemPFSnmpEidRange"	""	Filter traps by event id range.
"/AgentSetup/pemPFSnmpEvClass"	""	Filter traps by event class.

And for completeness, listed in Table 19-3 is all the other SNMP agent configuration variables. Details about these agent configuration variables can be found in the documentation, or in the comments in the original “config.default” text file.

Table 19-3 Other SNMP agent configuration variables

Agent Configuration Variable	3.2 Default	Purpose
/AgentSetup/snmpConfigAccess	“0”	Can SNMP access write to the agent configuration database?
“/snmp/support”	“yes”	SNMP agent support enabled?
“/snmp/agent_auto_start”	“yes”	SNMP agent starts automatically?
“/snmp/mibFileName”	“patrol.mib”	Name of the MIB file
“/snmp/default_timeout”	“500”	Timeout
“/snmp/default_retries”	“3”	Retries
“/snmp/default_r_community”	“public”	Default read community
“/snmp/default_w_community”	“private”	Default write community
“/snmp/agent_r_community”	“public”	Agent read community
“/snmp/agent_w_community”	“private”	Agent write community
“/snmp/master_agent_port”	“1161”	Master agent port
“/snmp/sysName”	“unknown”	System name
“/snmp/sysContact”	“http:// www.bmc.com”	System contact details
“/snmp/sysLocation”	“BMC Software Inc.”	System location details

Table 19-3 (cont'd) Other SNMP agent configuration variables

Agent Configuration Variable	3.2 Default	Purpose
"/snmp/masterAgentStartLine"	"/snmpmagt \$PATROL_HOME/ lib/snmpmagt.cfg NOV&"	Command to start master agent
"/snmp/masterAgentWorkingDir"	"\$PATROL_HOME/ bin/"	Master agent working directory
"/snmp/masterAgentName"	"snmpmagt"	Master agent executable name
"/snmp/masterAgentDir"	"bin"	Master agent directory
"/snmp/masterAgentConfigName"	"snmpmagt.cfg"	Master agent configuration file-name
"/snmp/masterAgentConfigDir"	"lib"	Master agent configuration directory
"/snmp/masterAgentParamName"	"NOV"	Master agent parameter name
"/snmp/masterAgentParamDir"	"log"	Master agent parameter directory
"/snmp/accessControlList"	Not set	ACL for SNMP accesses

For a deployment of PATROL, the required agent configuration variables can be preset in the "config.default" file that will be distributed with all agents. Alternatively, agent configuration can be changed via `pconfig`, `xpconfig`, or `wpconfig` agent configuration tools, or via PSL menu commands making use of the PSL `pconfig` function.

19.4.2 PATROL 3.2 Master Agent Configuration

For a PATROL 3.2 agent, the correct configuration of SNMP also involves the SNMP master agent. The SNMP master agent executable is launched separately from the PATROL agent. This separate launching is caused by a command built-in with the computer class KM (e.g., SOLARIS.km or NT.km), which launches this daemon via PSL from a "setup action." Refer to the PATROL product documentation regarding the SNMP master agent configuration issues in PATROL 3.2.

19.4.3 Disabling SNMP Trap Support

Agent built-in SNMP traps can be disabled without affecting other SNMP functionality. For example, you can still use PSL SNMP get, set, get-next, and walk functionality on other MIBs even if traps are disabled. Similarly, other SNMP managers can still walk the PATROL agent MIB even if trap sending is disabled.

The solution is simply to disable all trap sending without disabling the SNMP subagent. The two types of PATROL agent SNMP functionality can be separated from each other. Various methods are available for configuring the agent not to send any traps while leaving other SNMP support enabled:

- Ensure that no SNMP managers are listed to receive traps in `piVlm_list`.
- Ensure that all event classes are set to `NO_TRAP` for the 3.1 style of trap sending.
- Disable the event manager trap sending via the `"/AgentSetup/pemSnmpSupport"` variable.
- Change the PEM SNMP trap filters so that no traps pass. For example, use an impossible severity.

More radical is to disable all SNMP support including trap sending, MIB access, and external access. This basically involves ensuring that the SNMP agent is not running. This can be achieved via:

- Disabling the agent configuration variable `/snmp/support` to "no" or "0".
- Ensuring the SNMP master agent executable fails to start. Prevent the KM launching that is found in the computer class KM's setup commands.

19.4.4 Event Catalog Configuration for Agent Built-in SNMP Traps

The second part of configuration of agent built-in traps involves the Standard Event Catalog. In version 3.1 and later agents, the Standard Event Catalog settings control when the agent sends its built-in traps according to the generation of the standard events. These give more fine-grain control over trap sending, although not the full control that using PSL offers, as discussed starting on page 250.

The event catalog specifies which events will send SNMP traps on creation of the event. The main event classes of interest are:

- `UpdParState`—update status of parameter
- `UpdInstState`—update status of instance
- `UpdAppState`—update status of application class
- 9—parameter alarm canceled, as exception no longer exists
- 11—parameter value exceeded alarm range threshold value

By knowing which events are created when, we can choose when SNMP traps are sent. Table 19-4 maps common situations to the events that the agent creates.

Table 19-4 State change event creation

Agent State Change Situation	Event Class Created
Parameter changes state from OK to WARN/ALARM	UpdParState and 11
Parameter changes state from WARN/ALARM to OK	UpdParState (but not 9)
Instance changes state from OK to WARN/ALARM	UpdInstState
Instance changes state from WARN/ALARM to OK	UpdInstState
Application class state changes (any)	UpdAppState

Note that some exceptions to these rules exist. For example, if a PSL `set` directly changes the “status” variable of a parameter to “ALARM,” this will cause an `UpdParState` event for the state change, but not an alarm range threshold exceeded event of type 11.

19.4.5 Configuring the List of Recipients for Agent Built-in SNMP Traps

The agent sends the built-in state change traps out to the list of recipients according to the agent configuration. The agent configuration variable with the list of trap destinations is called “/snmp/piv1m_list” (this name must mean something to somebody!). According to the default “`config.default`” file, this is the pre-defined list of SNMP V1 interested managers. The lower/upper case of the characters is important.

Note that this list of trap destinations does not affect the recipients of traps sent by PSL `snmp_trap_send`, which explicitly supplies the destination of the trap. Therefore, this need for configuration does not apply to the use of PSL recovery actions or PSL event notification scripts to send traps (i.e., instead of agent built-in traps).

When using agent built-in traps, be sure to configure this agent variable appropriately. This agent configuration variable can be controlled via the normal agent configuration issues:

- `pconfig` command-line tool for editing agent configuration settings
- `xpconfig` and `wpconfig` GUI tools for editing agent configuration settings
- PSL `pconfig` function for altering the configuration of the current agent via PSL
- Editing or changing of the “`config.default`” text file, which takes effect when the agent first starts

The list of trap recipients can also be controlled via PSL. The agent configuration represents the default setting at agent startup, but can then be altered via the PSL `snmp_trap_register_im` built-in function (means to register an interested manager). This PSL function can process and change the list of trap recipients that are registered with the PATROL agent. This list of recipients takes effect immediately without requiring any agent restart or other action. However, the changes made by this PSL function are not persistent and do not survive an agent restart.

Note also that a bug appears to be in at least one version of the 3.2 agent that causes this PSL `snmp_trap_register_im` function to have no effect. This is probably not a true bug, but due to the default agent configuration setting of `"/snmp/trapConfTable"` to `"no."` The workaround is to change the configuration itself via PSL `pconfig`, or to use the PSL `pconfig` function to explicitly and persistently set the recipients list directly, and this takes effect dynamically without agent restart.

The list of trap recipients can also be persistently changed in PSL via the `pconfig` function. By changing the agent configuration variable `"/snmp/pivlm_list"` via PSL `pconfig`, the changes will be persistent even if the agent is restarted. This change to the list of recipients also takes effect immediately without requiring any agent restart or other action.

Manual editing of the text file called `"config.default"` can also change the initial set of configured trap recipients. This idea can be used to streamline the agent configuration process by distributing a preset `"config.default"` text file with each agent. This avoids the extra manual step of configuring this variable via `pconfig`, `xpconfig`, or `wpconfig`. However, editing the `"config.default"` file does not get reprocessed dynamically via the agent, and cannot be used to change configuration while the agent is running. An agent restart is required for a new `"config.default"` file to be processed.

19.4.6 Advantages and Disadvantages of Agent Built-in SNMP Traps

A number of limitations and disadvantages of using the standard agent built-in SNMP traps occur.

- Initialization to OK state change or "startup" traps for parameters cannot be prevented. This sends a number of traps for the OK state when a parameter receives its first value (i.e., a state change from VOID to OK). No agent configuration variable is available to disable these traps.
- Three traps are sent for a parameter state change in the PATROL 3.0 trap format: parameter, instance, and application state change traps, with one trap per state change level. Fortunately, for 3.1 and later agents, this situation can be resolved by re-configuring the agent to send traps according to different settings in the event catalog. (Note that the SNMP trap format choice of 3.0 or 3.1 controls only the trap format, and not when the trap is emitted.)
- The enterprise OID cannot be changed. No agent configuration or PEM setting exists to change it.

- The subid (i.e., the event-specific id for enterprise trap type 6) cannot be changed via agent configuration or PEM. This fact makes per-KM trap filtering or correlation difficult on some third-party consoles.
- Re-arm traps from an alert state back to OK cannot be suppressed.

However, some advantages exist to using the built-in agent SNMP traps:

- Re-arm traps occur by default when the parameter goes from alarm or warning back to OK when a problem is fixed. This allows SNMP notification to a third-party console that a problem is resolved.
- No PSL coding is required. Traps are sent by the core agent without any coding needed.

19.5 PSL SNMP Trap Sending

The use of PSL for SNMP trap sending also allows lots more flexibility to control the traps that are sent. Most of the disadvantages of agent built-in traps can be overcome. This flexibility includes both when they are sent, and what format they use when sent. For example, it has control over the enterprise ID and OIDs used for the traps.

The main PSL function related to SNMP trap sending is called “snmp_trap_send.” Full documentation on this function is available in the PSL Reference Manual, but in brief, the arguments to the `snmp_trap_send` function are as follows:

```
snmp_trap_send("/host/port/community",
               "address enterprise trap_type subid uptime",
               "objectid1 type1 value1",
               "objectid2 type2 value2",
               . . . );
```

In other words, the first argument specifies host information, the second specifies origin information, and the third and subsequent arguments specify trap information.

An example of PSL code to send a simple self-testing SNMP trap is shown below.

```
good_trap = "110.12.12.12 .1.1.1.1 3 5 1000\n.1.2.3.4 string Hello"
trap_text = good_trap . " This is a test trap";
host = "localhost";    # Send it to self for testing
ret = snmp_trap_send(host, trap_text);
if (ret != "OK") {
    # Error was detected. Failure sending trap.
}
```

The destination host for the SNMP trap is explicitly specified. The `piV1m_list` configuration variable for agent built-in traps does not affect those traps sent by PSL `snmp_trap_send` calls.

19.5.1 PSL SNMP Built-in Functions

A variety of PSL functions support SNMP. Details about these functions can be found in the PSL Reference Manual. A full list of these functions with brief details appears in Table 19-5.

Table 19-5 PSL SNMP functions

PSL Built-in Function	Purpose
snmp_agent_config	View or set the agent's SNMP community strings
snmp_agent_start	Start the agent's SNMP subagent
snmp_agent_stop	Stop the agent's SNMP subagent
snmp_close	Close a session used for SNMP get/set access
snmp_config	Report on sessions currently open
_snmp_debug	Control internal agent SNMP debugging features
snmp_get	Perform an SNMP get action on a MIB object (using an open session)
snmp_get_next	Perform an SNMP "get next" action on a MIB object (using an open session)
snmp_h_get	Perform an SNMP get action on a MIB object using a brief session
snmp_h_get_next	Perform an SNMP "get next" action on a MIB object using a brief session
snmp_h_set	Perform an SNMP set action on a MIB object using a brief session
snmp_open	Open a new session to be used for get, set, and get next operations
snmp_set	Perform an SNMP set action on a MIB object (using an open session)
snmp_trap_ignore	Ignore traps incoming to the PATROL SNMP subagent
snmp_trap_listen	Start recording incoming traps to the PATROL SNMP subagent

Table 19-5 (cont'd) PSL SNMP functions

PSL Built-in Function	Purpose
snmp_raise_std_trap	Raise the standard PATROL trap
snmp_trap_receive	Read the traps that were accumulated from incoming traps to the agent
snmp_trap_register_im	Change or read the list of SNMP managers registered to receive PATROL traps
snmp_trap_send	Send an SNMP trap with full control over its format
snmp_walk	Perform an SNMP walk operation on a MIB using an open session

From Table 19-5, you can see that a number of categories of these SNMP functions exist. The SNMP functions that allow trap sending from the agent to a manager console are:

- snmp_raise_std_trap
- snmp_trap_send

In addition, the function to control the list of SNMP managers and their details is snmp_trap_register_im.

The functions that affect the configuration of the agent, particularly whether it is capable of sending traps, are the following:

- snmp_agent_config
- snmp_agent_start
- snmp_agent_stop

However, since the SNMP subagent is auto-started by the PATROL agent on its own startup, these functions, such as snmp_agent_start, are not so important for sending traps. You have no need to use these to start the SNMP agent, provided the agent is configured correctly as part of the agent configuration process.

19.5.2 SNMP in KM Development

The other PSL functions are not so important for SNMP trap sending. Some functions are important for Knowledge Module development in order to write a Knowledge Module that reads

and processes SNMP data from the MIB of the application. A Knowledge Module can, via PSL, gather values from the application MIB and populate this data into Knowledge Module parameters. The power of PSL means that the processing of this SNMP-source data can be quite complex, and advanced. PSL functions to which this applies include:

- snmp_close
- snmp_config
- snmp_get
- snmp_get_next
- snmp_h_get
- snmp_h_get_next
- snmp_h_set
- snmp_open
- snmp_set
- snmp_walk

Also, SNMP functions are important for Knowledge Module development for processing of SNMP traps. If the application that a Knowledge Module monitors sends SNMP traps, these traps can be intercepted. The traps can be converted into PATROL alarms or events via PSL and then become part of the notification methods used with PATROL. Alternatively, the traps could be immediately converted into SNMP traps from the PATROL agent using the PSL trap sending functions. The PSL functions that are involved in the interception of SNMP traps are:

- snmp_trap_ignore
- snmp_trap_listen
- snmp_trap_receive

Further details of Knowledge Module development issues involving SNMP functions are discussed in Chapter 31.

19.6 PSL Recovery Actions for SNMP Trap Sending

Recovery actions are commands that the agent launches when a parameter exceeds a threshold. These are commonly used for corrective actions, but can also be used for SNMP trap notification. A simple way to use self-styled SNMP traps only for those parameters that matter is to use recovery actions containing calls to the PSL SNMP functions.

An example PSL script that can be used as a recovery action to send SNMP traps is shown below. This example is actually from a regression testing KM, but the only difference is the use of "localhost" as the trap destination.

```
param_name = get("name");      # Get parameter name
inst_name = get("../name");    # Get application instance
```

```

app_name = get("../name"); # Get application class
hostname = get("/hostname"); # Get machine name
platform = get("/appType"); # Get platform (NT/UNIX/OS/2)
param_value = get("value"); # Current value of parameter

alarm_min = get("alarmMin"); # Minimum of activated alarm range
alarm_max = get("alarmMax"); # Maximum of activated alarm range
status = get("status"); # Parameter status

# Improve formatting for integral parameters
# Remove this line for non-integral or text parameters
param_value = int(param_value);

ipAddress = get("/ipAddress"); # Agent IP address

bmc_enterprise_id = ".1.3.6.1.4.1.1031";

#Hostname to send traps to; localhost to test
#recipient = "localhost";

# IP Addr to send traps to; localhost to test
recipient = "127.0.0.1";
current_time = date(); # Report the current time in the trap
message_format = "RECOVERY_TRAP %s %s %s HOST=%s IP=%s PLATFORM=%s
VALUE=%s STATE=%s MIN=%s MAX=%s TIME=%s";
mesg = sprintf(message_format,
    param_name, inst_name, app_name, hostname,
    ipAddress, platform,
    param_value, status, alarm_min, alarm_max, current_time
);
print(recipient, mesg);

# Use enterprise trap number 6, and subid 255, uptime 1000
trap_origin = ipAddress . " " . bmc_enterprise_id . " 6 255 1000";
trap_text = ".1.2.3.4.5 string " . mesg;

ret = snmp_trap_send(recipient, trap_origin, trap_text);
if (ret != "OK") {
    print("SNMPTEST: ERROR: Recovery action trap returned '"
        . ret . "' with errno " . errno);
    # Other error-handling method as appropriate
}

```

The above example is quite simple but effective. The above PSL script needs to be changed to have the correct trap destination, possibly more than one destination, which requires

multiple PSL calls to `snmp_trap_send` with the different destinations. The format of the trap text body can be an arbitrary text string containing any required information.

This example represents the basis of an SNMP trap sending recovery action. A number of possible advanced extensions to this recovery action PSL script include:

- Destination configuration: Not hard-coding the trap recipient where the traps are sent. This is hard-coded as “localhost” for testing in the example above. This could be read from an agent configuration variable or some other external file.
- Blackout: Adding timing logic to avoid sending traps at particular times, such as known maintenance downtime periods.
- Destination times: Sending traps to different destinations at different times such as day and night shifts.
- Destination classes: Sending traps to different destinations based on application, instance, or parameter name.
- Re-arm traps: Sending a re-arm trap when the parameter returns to the OK state.
- Per-KM OIDs: To ease trap filtering on some third-party products, you want to have a different subid for each application class (e.g., ORACLE and SYBASE), and this PSL script can be modified to achieve this. For example, it could read a map file that converts application class names into subid number values.

The advantages of using recovery actions include:

- Easy control over what parameters send traps—you add the recovery action script to the parameter for each one required.
- A single PSL file can be used for each parameter by getting context information from variables rather than from hard-coding items such as parameter names, alarm range thresholds, and so on.

The use of recovery actions is an effective and simple means of controlling SNMP traps. However, over time, the need to alarm on instance states and the impact of the KM upgrade problems become an issue, and for this reason, the use of event notification PSL scripts seems to be preferred.

19.7 PSL Event Notification Commands for SNMP Trap Sending

This is, in my opinion, the best way to send SNMP traps from PATROL if you want control over it. By using PSL, you have full control over the traps, and by intercepting events via the notification scripts, you are guaranteed to capture every action.

Notification scripts are PSL commands that run immediately upon creation of the event. These scripts are defined in the event catalog file, such as the Standard Event Catalog in “StdEvents.ctg.” The event catalog can be modified via the Developer console.

The sequence of actions for using notification PSL scripts to send SNMP traps usually involves:

1. PATROL agent generates an internal agent built-in event
2. The notification PSL script for this event is automatically scheduled by the agent.
3. Context details such as the parameter name are read by the PSL notification script via the set of builtin `%{ . . . }` event macros.
4. The PSL notification script does some user-specific processing to determine whether to send an SNMP trap for this parameter or object; this issue can be solved various ways.
5. SNMP trap is sent via PSL `snmp_trap_send` function in the notification script.

The main difference between recovery action trap sending and notification script trap sending involves the context. For a recovery action, you add only that recovery action to each parameter for which you want a trap sent. Therefore, the act of configuration of the recovery actions implicitly decides in what context to send traps, and you have no need to filter out unwanted trap situations. For example, if you want to send a trap for the ORACLE ServerDown parameter, but not for the UNIX FILESYSTEM FSCapacity parameter, you add only a recovery action to the ServerDown parameter.

Notification scripts are different in this respect. The agent will launch the notification script for all objects. You need some other way to specify that you send traps for ServerDown but not for the FSCapacity parameter. This can be solved in a number of ways within the PSL script. All solutions have the common element that the PSL notification script needs to read some configuration data somewhere and process it against the context information. For example, a common solution appears if you use an agent configuration variable to list the parameters for which you want traps sent. In this case, the notification script must use PSL `pconfig` or PSL `get` to read that list, and compare the parameter name from the `%{ . . . }` macro with that list. Based on whether the parameter appears in the list, a trap is sent, or another action may occur.

19.7.1 Event Notification Scripts and `%{ }` Event Macros

The PSL script that performs trap sending needs to determine its context. This is possible via the special built-in event catalog macros using the `%{macro_name}` syntax. These are placed in the PSL text for the event notification command, typically within the PSL command for an event class in the Standard Event Catalog. Table 19-6 lists all the event notification script macros.

Table 19-6 Event notification macros

Event Macro Name	Meaning
<code>%{EV_ID}</code>	Internal identification unique number.
<code>%{EV_STATUS}</code>	Status of event: e.g., open, closed, acknowledged,...

Table 19-6 (cont'd) Event notification macros

Event Macro Name	Meaning
<code>%{EV_TYPE}</code>	Type of event.
<code>%{EV_TIME}</code>	Time of event creation.
<code>%{EV_ORIGIN}</code>	Object triggering event, such as instance or parameter.
<code>%{EV_NODE}</code>	Hostname of node where event occurred.
<code>%{EV_DESC}</code>	Description of the event. Why it was triggered.
<code>%{EV_DIARY}</code>	Record of actions performed on event.
<code>%{EV_HANDLER}</code>	Person who last acted on the event.
<code>%{EV_OWNER}</code>	Person assigned as owner.
<code>%{EV_NAME}</code>	Name of the event catalog listing for event.
<code>%{EV_NSEVERITY}</code>	Severity of the event, a numerical value.
<code>%{EV_CLASS_NAME}</code>	Name of the event class.

19.7.2 Example: Event Notification PSL Script for OFFLINE Instances

The following is an example PSL notification script that intercepts instance state changes, particularly instances that go `OFFLINE`. This example is intended as a notification script for the `UpdInstState` event class in the Standard Event Catalog, as discussed on page 259.

The PSL script uses the `%{EV_...}` macros to gather context data. It also further processes the data from the event description, which in this case contains the original and current status of the object. Although the current status is available via `%{EV_STATUS}`, the original status of the object prior to the state change is not available via one of the `%{...}` built-in event macros.

This idea can clearly be generalized to intercept all types of events and send traps as desired. No specialized processing determines whether a trap will be sent based upon the name of the instance. Filtering or other logic should be added here. This can be coded via normal PSL methods to perform these tests.

```

# OFFLINE Instance Actions!
# Notify Command for UpdInstStatus event
inst = "%{EV_ORIGIN}";
desc = "%{EV_DESC}";
DEBUG = 0; # Internal debug for this script
# Process ORIGIN to get instance name
# ORIGIN is format APPL.INST
# Convert .'s to /'s
inst = nthargf(inst, "1-", ".", "/");
full_inst_path = "/" . inst;

# Process the description to get states
# Format of the description is..
# State Change: OLDSTATE NEWSTATE
# .. Reason for state change.

first_line = nthlinef(desc, 1);
newstate = ntharg(first_line, 4);
oldstate = ntharg(first_line, 3);
if (newstate == OFFLINE) { # We have an instance going OFFLINE
    if (oldstate != VOID) {# Change state (not creating new one)
        # Instance went from OK/WARN/ALARM to OFFLINE
    }
    if (oldstate == OK) {
        # Instance was OK and now just went OFFLINE
        # ... Send trap here...snmp_trap_send()
        printf("Instance %s has gone from OK to OFFLINE\n",
            full_inst_path);
    }
}

if (DEBUG) {
    inststate = get(full_inst_path . "/status");
    if (inststate != newstate) { # Just in case... validate the state
        printf("\n\n*** INTERNAL ERROR *** - Instance '%s', "
            ".State '%s' not consistent with expected state '%s'.\n",
            full_inst_path, inststate, newstate);
    }
    print("\n\nInstance Notification command\n");
    printf("OLD STATE = %s, NEW STATE = %s, INST = %s\n",
        oldstate, newstate, full_inst_path);
    print("\tTIME = %{EV_TIME}\n");
    print("\tNODE = %{EV_NODE}\n");
    print("\tORIGIN = %{EV_ORIGIN}\n");
    print("\tDESC = %{EV_DESC}\n");
    print("\tNAME = %{EV_NAME}\n");
    print("\tCLASSNAME = %{EV_CLASS_NAME}\n");
}

```

19.7.3 Standard Event Catalog Classes

The main place to determine which events cause traps is within the Standard Event Catalog. This can be edited via the PATROL Developer Console, from a menu command in the UNIX Developer Console, and as the last entry at the bottom of the outline view for the Windows NT console. The standard event catalog is actually stored in a text file named “StdEvents.ctg” in the “lib/knowledge” directory of the PATROL install on both UNIX and NT.

The event catalog is important when using PSL notification scripts to send traps. The standard event catalog is where you define which events have PSL notification scripts associated with them.

The event classes in Table 19-7 are most important when dealing with traps from PATROL state changes due to a detected problem in a monitored application. These are the classes that are most commonly altered to get the required traps.

Table 19-7 Standard Event Catalog event classes

Event Class	Purpose
UpdParState	New/updated parameter state.
UpdInstState	New/updated instance state.
UpdAppState	New/updated application state.
UpdMachineState	New/updated state for the entire agent.
9	Alarm is canceled; parameter back to OK.
11	Parameter exceeds threshold; triggered alert.

By knowing which events are created when, we can choose when the traps are sent. Table 19-8 maps common situations to the events that the agent creates.

Table 19-8 Agent state change results

Agent State Change Situation	Event Class Created
Parameter changes state from OK to WARN/ALARM	UpdParState and 11
Parameter changes state from WARN/ALARM to OK	UpdParState (but not 9)

Table 19-8 (cont'd) Agent state change results

Agent State Change Situation	Event Class Created
Instance changes state from OK to WARN/ALARM	UpdInstState
Instance changes state from WARN/ALARM to OK	UpdInstState
Application class state changes (any)	UpdAppState

Note that some exceptions to these rules exist. For example, if a PSL `set` function call directly changes the “status” variable of a parameter to “ALARM,” this will cause an `UpdParState` for the state change, but not an alarm range threshold exceeded event of type 11.

For completeness, Table 19-9 shows all the standard event classes. These other classes can be useful for sending traps on other situations, such as a console disconnecting.

Table 19-9 Event class explanations

Event Class	Meaning
RegApp	New KM class is now registered and running in the agent (e.g., when a new console connects requesting the KMs that it is interested in viewing).
UpdAppState	New/updated application state.
WorstApp	This application now has the worst state of all applications in the agent.
UpdParState	New/updated parameter state.
UpdInstState	New/updated instance state.
UnregAllApp	Unregister all applications.
UpdMachineState	New/updated state for the entire agent (due to some change in the state of an application).
Diag	Diagnosis event.
RemPsl	Used by remote PSL execution.

Table 19-9 (cont'd) Event class explanations

Event Class	Meaning
Result	Used by remote PSL execution.
PslSet	Used for remote PSL set execution.
RemProcess	Used in remote PSL file transfer and the API.
EventArchive	Events have been archived.
Disconnect	Console disconnected from agent.
Unload	Agent unloaded KM class.
R3FilterData	Used by the SAP R/3 KM only.
1	Agent's overall state has changed for this agent machine.
2	Worst application class name is provided in this event, when the agent's state has changed.
3	Worst application instance name is provided in this event, when the agent's state has changed.
4	Discovery has been started for a KM class.
5	Discovery has been disabled for a KM class.
6	Agent and console have different version of a KM.
7	Successful connection to the agent by a user (i.e., a normal console connection or one involving the API or PSL remote functions).
<8>	<No such event number 8 is in the event catalog>
9	Alarm is canceled because the condition regarding the parameter violating its thresholds has disappeared. In other words, the parameter's value is no longer a bad value that causes an alarm, and the parameter is going back to the OK state.

Table 19-9 (cont'd) Event class explanations

Event Class	Meaning
10	Recovery action has been executed for the parameter.
11	Parameter value has exceeded the alarm range thresholds. This will raise a warning or alarm state for this parameter.
12	All recovery actions have executed and failed to resolve the problem. The parameter will stay in its current state. Agent will not execute any more recovery actions for this parameter.
13	Suspended all parameters of this KM class.
14	Restarting all local and global parameters of this KM class.
15	Restarting all local and global parameters of this KM class.
16	Parameter description has been modified (i.e., by KM editing) and the parameter state is reset to OK.
17	Global parameter has started.
18	Global parameter has started.
19	Local parameter has started.
20	Parameter had bad output. For example, PSL set on "value" did not provide an integer to a graph or gauge parameter.
21	Local parameter is suspended and will no longer run.
22	Global parameter is suspended and will no longer run.
23	Global parameter is suspended and will no longer run.
24	Agent process cache cycle changed.
25	Agent process cache cycle changed.

Table 19-9 (cont'd) Event class explanations

Event Class	Meaning
26	Application discovery is disabled for this KM class.
27	Application discovery is enabled for this KM class.
28	Username/password were invalid to connect to the agent (e.g., via the API or PSL remote functions).
29	Internal agent or PEM failure of some type.
<30-37>	<No event classes numbered 30 through 37 inclusive exist.>
38	Parameters of a KM were restarted.
39	Parameter threshold was exceeded by parameter value. State change event.
40	PSL response-related event. Created when the agent launches a PSL response function.
41	Information event. Placeholder for user-defined events. Not generated internally by the agent.
42	Information event. Placeholder for user-defined events. Not generated internally by the agent.
43	Information event. Placeholder for user-defined events. Not generated internally by the agent.

19.7.4 Advantages and Disadvantages of PSL Event Notification Scripts

The advantages of using event notification PSL commands to send SNMP traps include:

- Easier KM migration—The configuration settings for this method are not in the KM files for other KMs, as recovery actions would be. Configuration can be in a separate KM file containing all the PSL logic. In this case, migration merely involves re-using the file. Alternatively, the settings such as the list of parameters to send a trap for can be in agent configuration variables or an external file. Both of these formats can be re-used as part of upgrading.

- Single file can be used—All of the SNMP trap logic can be centralized into a single KM, or even in a single PSL script.
- Instance state changes can be detected. By adding a notification PSL command to the event for an instance state change, you can send SNMP traps or other notification logic.
- Re-arm traps can be sent when a parameter or object returns to OK state from an alert state. Various event classes are triggered on all state changes including returning to OK.
- Initial startup traps can be suppressed. The PSL code can determine when the state change is an initial state change (from VOID), and the PSL code can avoid sending a trap in this case.
- Other events can cause events. Any of the built-in agent events can trigger SNMP traps if desired. One example might be console disconnection.

The disadvantages of using event catalog notification scripts arise mainly from the extra effort involved in building the infrastructure that you need:

- PSL coding is required. These features are not hard-coded into the agent without programming. However, using the expertise from this chapter, the coding requirements can be minimized.
- Event catalog file must be manually modified to add the notification script to the correct event classes. This action is necessary initially and probably on each upgrade release of PATROL.
- Configuration of which parameters or other objects require traps sent must be built into the PSL script. This can be both a strength and a weakness.

19.7.5 Repeated Trapping Support

A desired feature is often to re-send the trap repeatedly until a problem is fixed. You can achieve this via the PSL method by coding a recovery action or event notification command to repeatedly re-check the state of a parameter (or instance) to see whether it has gone back to OK, and re-trap if not. The PSL code for that might look similar to:

```
# Recovery action PSL script
status = get("status");    # Get the parameter's status
while (status != OK) {
    # send trap via PSL snmp_trap_send()
    sleep(10);    # Sleep for 10 seconds
    status = get("status");
}
# Alarm has gone away... Send a cancel trap
# send a new trap via PSL snmp_trap_send()
```

19.8 Alert Sending and Filtering Based on Status Pairs

The state changes for PATROL parameters and instances are sometimes too frequent. Via PSL, you can control which parameters send traps. The next level of control is clearly having control over exactly the circumstances under which traps are sent. For example, some of the common needs are:

- Alarm or warning—parameter goes to alarm or warning due to alarm range threshold exceeded.
- Instance goes to alarm or warning—due to PSL `change_state` or due to parameter state inheritance.
- Instances goes OFFLINE—due to PSL `change_state`; cannot occur due to parameter inheritance.
- Parameter returns to OK from an alarm or warning state — alert condition no longer exists (re-arm traps).
- Parameter is newly created—state change from VOID to OK (or an alert state) when the parameter is initially set to its first value.
- Instance is newly created—state change from VOID to OK (or an alert state) via PSL `create`.
- Parameter is deactivated—goes to OFFLINE state (OFFLINE parameter means absent).
- Instance is destroyed—goes to VOID state; usually via PSL `destroy` (also via `change_state`).

The main method of making conversions based on different statuses is via PSL and typically via event notification commands. PSL recovery actions do not have any way of detecting the previous parameter status.

PSL event notification commands can determine the previous state from the text in the event description, for at least one event class, which is the state change event. This event notification method of extracting the old status is demonstrated in the example showing how to send alerts on OFFLINE instances.

19.8.1 Distinguishing Instance State Changes from Propagated States

An application instance can achieve the WARN or ALARM states two basic ways: via PSL `change_state` or via an inherited parameter state.

The OFFLINE state can occur only due to PSL `change_state`, and cannot be inherited from parameter states. A parameter cannot go OFFLINE since this is used to represent an inactive parameter.

For WARN and ALARM, a direct state and a propagated parameter state can be detected by analyzing the “status” and “ruleState” variables in the agent symbol table. Typically, this is via PSL `get` in a PSL script.

The “status” and “ruleState” variables are built-in variables in the instance’s symbol table. The “ruleState” is the base state of the instance, and “status” is the combined state of the instance plus any propagated parameter states. Their values can be read via PSL `get` function calls. The situations in Table 19-10 apply.

Table 19-10 Interpreting instance state settings

“Status”	“ruleState”	Meaning
OK	OK	Nothing going on
WARN	OK	Propagated WARN parameter
ALARM	OK	Propagated ALARM parameter
OK	WARN	Not possible
WARN	WARN	PSL <code>change_state</code> to WARN
ALARM	WARN	PSL <code>change_state</code> to WARN, but also a parameter in ALARM
OK	ALARM	Not possible
WARN	ALARM	Not possible
ALARM	ALARM	PSL <code>change_state</code> to ALARM
any state	OFFLINE	PSL <code>change_state</code> to OFFLINE

19.9 Testing Agent SNMP Trap Sending

Testing is the next step after the SNMP agent is configured correctly to send SNMP traps. You have a few basic options for testing, and all involve watching for SNMP traps coming out:

1. SNMP manager console—check that it is receiving the traps as desired. This idea does not use any PATROL console technology.
2. Agent self-testing—run a PSL script in the agent to receive its own traps and print them. The logic involving SNMP trap receiving can be used in this way, such as PSL `snmp_trap_listen` and `snmp_trap_receive`. An example appears later in the chapter, on page 271.

3. Deployment KM's SNMP Trap Watch Window—this is a tool available with PATROL that loads menu commands into the computer icon. One of these menu commands pops up a dynamic response dialog (or “watch window”) that shows the outgoing traps of the agent. It achieves this via self-registering for its own traps via PSL `snmp_trap_listen` and `snmp_trap_receive`. This requires a 3.1 agent and console or above. On 3.0 consoles, you have an alternative Watch Task menu.

Unfortunately, currently no simpler mechanism is available for a quicker test of SNMP trap sending. No internal agent variables or counters indicate the number of traps sent.

19.10 Console SNMP Trap Sending via State Change Actions

One final alternative to the various PSL agent-based trap-sending methods is to use a PATROL Console feature. The built-in traps, PEM controls, and PSL trap sending methods all originate from the agent. One console feature is occasionally used for console-side SNMP trap sending.

The PATROL Console can launch system commands that can then send SNMP traps. A variety of attributes in the KM hierarchy are called “State Change Actions.” These are console-side commands that launch operating system commands on the console's machine. No support for PSL exists in console-side state change actions because the agent does not execute these commands. The console does not have a built-in PSL interpreter. However, some context information is available from a set of console-side `%{ }` macros for state change action commands (see Table 19-11).

Table 19-11 Console state change macros

Macro Name	Meaning
<code>%{HOSTNAME}</code>	hostname
<code>%{TIME}</code>	current time
<code>%{STATUS}</code>	status of object
<code>%{APP_TYPE}</code>	application name (e.g., ORACLE)
<code>%{INSTANCE_ID}</code>	instance id (e.g., Oracle SID)
<code>%{SID}</code>	instance id (e.g., Oracle SID)
<code>%{HOME}</code>	home of this object

Table 19-11 Console state change macros

Macro Name	Meaning
%{VOLUME}	sound volume
%{LANGUAGE}	language to use in sounds
%{COMP_TYPE}	computer type (e.g., SOLARIS)
%{COMP_STATUS}	computer state (e.g., VOID, OK)
%{CON_TYPE}	network connection type (UDP or TCP)
%{CON_STATUS}	network connection state
%{CON_INFO}	network connection info (i.e., hostname)
%{DISPLAY_NAME}	hostname where the console is displayed
%{WORST_CHILD}	worst child (parameter or instance) of this object
%{EVENT_INFO}	event that caused this user action

There are few advantages of using the console for SNMP trap sending. Firstly, the SNMP trap sending can originate from a single source, and only one place needs to be configured. On the downside, the use of the console becomes a point of failure. Also, if the network connection between console and agent goes down, the console can send only one trap noting the lost connection, and can no longer provide more detailed alarm conditions such as whether the database server is also down.

PATROL does not provide SNMP trap sending support out-of-the-box in the console. The use of SNMP traps by the console requires a system script that launches a trap sending executable. Some releases of PATROL have shipped with an external “snmpttrap” executable. This could be used for this purpose in console state change actions.

19.10.1 Disconnecting Agent Traps via Console State Change Actions

One of the primary uses of the console-based state change actions is to perform an action when an agent disconnects. The console executes the VOID state change action when the connection to an agent is lost unexpectedly. This can be used, as discussed above, to send an SNMP trap

when an agent dies or an agent machine crashes. The VOID state change action of the computer class ALL_COMPUTERS (or multiple specific classes like SOLARIS or NT) needs to be altered to have a new VOID state change action command. This must launch a shell script that uses some executable to send the SNMP trap, since PSL is not supported by console state change actions.

An alternative to using the VOID state change action is to use the ALARM state change action. However, this means that any alarms can cause traps, not just agent disconnections. Hence, you need to check the CON_STATUS state change macro. If it is “OK,” then the agent has not connected. Other values indicate that the console-agent connection is broken.

Another alternative method of detecting downed PATROL agents and sending traps is via the agent. Various “agent ping” KM methods are available. These effectively amount to using a KM to regularly poll another agent to ensure that it is still awake. Groups of agents can monitor each other in any configurations using these methods. This method is also agent-based and thus allows the full power of PSL in processing where and when to send an SNMP trap. One way to make this similar to the console-side method is to run an agent on the console machine and use this local agent to run an “agent ping” KM.

19.10.2 Configuring SNMP Management Consoles for PATROL Traps

SNMP trap notification requires configuration on two ends: the PATROL agent sending the traps, and the non-PATROL SNMP management console receiving the traps. The agent needs to know where to send the traps. The SNMP management console needs to know how to recognize PATROL traps and what to do about them. Unfortunately, the required configuration is highly specific to the management console. On some consoles, it involves configuration of internal rules and tables. In others, it may involve configuring the “trapd.conf” configuration file.

19.11 PATROVIEW Integrations versus SNMP Trap Sending

The PATROVIEW products always provide a much deeper integration into a console than does sending SNMP traps. SNMP traps may also require PSL coding for recovery actions or event notification scripts. However, if the only purpose of the enterprise console is to act as a help desk style of event management console, then SNMP traps are an alternative.

PATROVIEW components typically offer greater integration at the event level, launching the PATROL console, and iconic drill-down views of PATROL objects. This is certainly valuable functionality if the enterprise console is used by operators in ways that make these deeper features of PATROVIEW useful. For example, browsing PATROL objects and examining parameter values may be desirable for operators to gather more data about a problem. On the other hand, if the operator is simply viewing a textual list of events or trouble tickets, then sending problems to the enterprise console via SNMP is adequate.

The technology used by PATROVIEW differs from SNMP. It uses the PET component, which is based on the PATROL API and its use of the TCP or UDP transport layers. In theory, SNMP traps can get lost because they are unreliable, whereas PATROL’s extra layer of reliabil-

ity on top of the UDP or TCP connection avoids this possibility. However, the unreliability of SNMP trap sending is usually more of a theoretical concern than a practical problem.

19.11.1 Evolution of PATROL Agent SNMP Technology

SNMP support for the PATROL agent was first added in the 3.0 release. The PSL SNMP functions have been available since that release. Only a single SNMP trap format was available. The event manager did not have SNMP support in the event catalog. The SNMP subagent did not act as a true SNMP subagent, but required master agent access.

PATROL 3.1 saw a number of advances in SNMP technology. The new 3.1 trap format was added, but the old 3.0 trap format was still available via configuration. The event catalog was extended to offer SNMP trap sending support for all event classes. Still no support was available in 3.1 for true master and subagent capabilities in the SNMP subagent.

PATROL 3.2 saw the introduction of the new SNMP technology that is also available in the PATROL SNMP Toolkit. This introduced a true master agent and subagent capability for the SNMP support. This changed the configuration requirements for some aspects of the SNMP subagent. The SNMP master agent also had to be launched separately in the cases where the user did not already have another SNMP master agent on the machines using PATROL.

19.12 SNMP Trap Receiving and Forwarding

PATROL SNMP support can be used to receive SNMP traps from other sources. These traps can then be either forwarded as new SNMP traps, or converted to PATROL events or alarms for other forms of notification or historical recording. In this way, the PATROL agent can act as a form of SNMP middle manager. The PATROL Console has no SNMP support, and only the agent can receive traps. The console can manage traps indirectly if the agent converts them into PATROL events or alarms, which the console can process.

To convert the PATROL agent into an SNMP trap handler involves sophisticated PSL coding, no such functionality is built directly into the PATROL agent or console. Via a KM that uses PSL, the PATROL agent can be converted into some form of SNMP trap handling daemon. The PSL functions that support this method of SNMP trap receiving and analysis are shown in Table 19-12.

Table 19-12 PSL SNMP trap sending functions

PSL Built-in Function	Purpose
snmp_trap_ignore	Ignore traps incoming to the PATROL SNMP subagent.
snmp_trap_listen	Start recording incoming traps to the PATROL SNMP subagent.
snmp_trap_receive	Read the traps accumulated from incoming traps to the agent.

Another less common use of SNMP trap receiving occurs in KM development. If an application is sophisticated enough to trigger SNMP traps on internal problems, these can be used as alert conditions by the KM developer. The KM developer can add features such that the PATROL KM can raise PATROL events and alarms when it receives a trap from some other SNMP-enabled device/application.

19.12.1 Example: PATROL Agent Receiving Its Own Traps

A good example of how the agent can receive traps via PSL is to register for its own traps. The sequence of PSL code to achieve this is relatively simple:

```
# 1. Register interest to receive traps
# For this self-testing case,
# it is an internal registration with the agent itself via PSL

errno = 0;
# Add self as recipient/target host
ret = snmp_trap_register_im("localhost", "add");

if (ret != "OK") {
    exit; # Replace with appropriate error handling
}

# 2. Ensure that the SNMP agent is listening for incoming traps
E_PSL_SNMP_ALREADY_LISTENING = 98;
ret = snmp_trap_listen();
# Could return either OK or "" depending on if already going
if (ret != "OK" && errno != E_PSL_SNMP_ALREADY_LISTENING) {
    # Detected some failure, and not a harmless indication
    # that the listener was already listening.
    exit; # Replace with appropriate error handling
}

# 3. Loop waiting for incoming traps
while(1) {
    errno = 0;
    ret = snmp_trap_receive();    # Wait for incoming trap

    if (ret == "" || errno != 0) {
        # Some failure to receive a valid trap
        exit; # Replace with whatever error handling is appropriate
    }
    # process the incoming received trap here
    print("TRAP DETECTED: ", ret, "\n");
} # while loop
```

This sample PSL code illustrates the basic ideas behind receiving SNMP traps via PSL. The first step is not typical, because registration for traps is internal to the agent. The KM is registering with the agent to receive traps from the agent. A more typical use of registration for SNMP traps from some external non-PATROL device or application will require some other steps, such as either an SNMP set operation via PSL, or a PSL `system` call to launch a child process.

19.13 Gathering Reporting Data from the PATROL MIB

SNMP is an important integration point for tools that require KM data, such as data archival or reporting tools. The PATROL KM parameter values can be gathered via SNMP get operations for storage in some other repository. The PATROL agent MIB consists of a number of dynamic tables through which can navigate to gather PATROL KM data. This path is difficult to poll for some applications because of the dynamic nature of the object ids. However, another feature of the MIB has a separate “values” table that allows easier access to parameter value data via the name of the parameter. This allows a more static access to the same parameter data. This feature is long-standing and robust, having been available since version 3.0 of the agent.

You can, via an SNMP get operation, get a great many attributes of a parameter, instance, or application class object. Via this MIB table, you can effectively walk all the leaves of the agent internal symbol table (i.e., the area accessed via PSL `get`). Some examples include:

- Parameter value—SNMP get on the “value” variable of a parameter name
- Parameter status—SNMP get on the “status” variable of a parameter name
- Parameter last set time—SNMP get on the “time” variable of a parameter name
- Instance status—SNMP get on the “status” variable at instance level
- Application class status—SNMP get on the “status” variable at class level

Unfortunately, currently you can only receive the current value and other current statistics (e.g., time) for a parameter via an SNMP get operation. You have no way to receive the history of a parameter via SNMP get. This is the situation with the 3.2 release of the PATROL agent.

Another issue with this SNMP get access is that 3.0/3.1 differ from the 3.2 functionality. Although the feature is available from both 3.1 and 3.2 MIBs, the method of access involves one extra term in 3.2. This is an unfortunate problem with backward compatibility.

19.13.1 Example: Parameter Values from PATROL KMs via SNMP get in v3.1

The following shell script “`getVar`” is a UNIX shell script that uses the “`snmpget`” executable to gather data from a remote PATROL agent. This script uses the following executables:

- `snmpget`—standalone executable that does an SNMP get operation
- `conv`—homegrown executable that maps a string into its numeric ASCII form (e.g., “aaa” becomes “64.64.64”)

The getVar shell script can be used according to the following various syntaxes:

- getVar nanou /CPU/CPU/CPULoad—Gets the parameter value
- getVar nanou /CPU/CPU/CPULoad value—Gets the parameter value
- getVar nanou /CPU/CPU/CPULoad status—Gets the parameter status
- getVar nanou /CPU/CPU/CPULoad time—Gets the parameter's last timestamp
- getVar nanou /CPU/CPU/CPULoad interval—Gets the parameter's poll time
- getVar nanou /CPU/CPU status—Gets the CPU instance's overall status
- getVar nanou /CPU status—Gets the CPU class's overall status
- getVar nanou / status—Gets the agent's overall status

In order to understand how this function operates, let us see what SNMP MIB variable it accesses. The basic MIB oid for the PATROL MIB with the sequence:

```
patrolObjects.variablesTable.variableEntry.variableValue
```

is seen in the script as:

```
.1.3.6.1.4.1.1031.1.1.1.5.1.4
```

This is appended with a numeric representation of the ASCII name of the sequence. For example, the string /CPU/CPU/CPULoad/value becomes represented as if it were:

```
/.C.P.U./.C.P.U./.C.P.U.L.o.a.d./.v.a.l.u.e
```

This means that the full path of the SNMP oid is:

```
.1.3.6.1.4.1.1031.1.1.1.5.1.4./.C.P.U./.C.P.U./.C.P.U.L.o.a.d./
.v.a.l.u.e
```

However, this is actually represented in numeric form with the ASCII codes for these letters rather than the actual alphabetical characters. The 'conv' utility is a homegrown 10-line C program that simply takes its first string argument and converts it into period-separated numeric values.

```
#!/bin/csh -f
# Simple script to get PSL var value from Patrol agent via snmp v1
# Format: getVar host param_name [var_name]
#       var_name is 'value' by default
# Example: getVar hudson /FILESYSTEM/root/FSCapacity status
#
# Uses executables for Solaris:
set GET=/usr/local/bin/snmpget
set CONV=/usr/local/bin/conv
```

```

if ($#argv != 2 && $#argv != 3 ) then
    echo Usage: getVar host param_name '[var]'
    exit
endif

if ( $#argv < 3 ) then
    set var=value
else
    set var=$3
endif

# patrolObjects.variablesTable.variableEntry.variableValue
set varValOid=.1.3.6.1.4.1.1031.1.1.1.5.1.4
set oid=$varValOid.'$CONV $2/$var'
set result='$GET -v 1 $1 public $oid'
echo $result | fgrep -s Error
if ( $status == 0 ) then
    echo $2/$var : $result
else
    echo $2/$var $result | cut -d ' ' -f 1,3,4
endif

```

This example shows that the MIB for the PATROL Agent can be used to gather various kinds of KM data. This can be used for a reporting mechanism to poll parameter values via the PATROL MIB. It can also be used to examine statuses of objects to determine whether alerts exist. A great many other uses exist for this SNMP MIB access, because a great deal of data in the PATROL agent symbol table is accessible via the PATROL agent MIB.

19.13.2 PATROL 3.2 Parameter Data Access via SNMP MIB

The method of accessing parameter data via the MIB in PATROL 3.2 agents involves one extra term. The sequence to gather the parameter value on 3.1 agents is, as shown above, to perform an SNMP get on:

```
.1.3.6.1.4.1.1031.1.1.1.5.1.4./.C.P.U./.C.P.U./.C.P.U.L.o.a.d./v.a.l.u.e
```

However, on a 3.2 agent, an additional length term is before the first “/” character. So the feature becomes like:

```
.1.3.6.1.4.1.1031.1.1.1.5.1.4.<LEN>./.C.P.U./.C.P.U./.C.P.U.L.o.a.d./v.a.l.u.e
```

In this case, <LEN> is the length of the string “/CPU/CPU/CPULoad/value,” which is 22. So the correct access becomes the numerical equivalent of below:

.1.3.6.1.4.1.1031.1.1.1.5.1.4.22./.C.P.U./.C.P.U./.C.P.U.L.o.a.d./.v.a.l.u.e

Of course, the actual access method requires numeric values that are the ASCII equivalents of these letters.

Knowledge Module Development

KM Development Overview

This chapter aims to give a concise overview of what is needed to build a Knowledge Module for monitoring an application. Although the GUI navigation steps of the PATROL Consoles for Unix and Windows NT are not detailed, I include discussion of the PSL code and dialog settings that are best used to build a KM.

20.1 Required Development Tools

The PATROL environment includes a number of tools for developers. The basic development tools for building a KM include:

- PATROL Developer Console—GUI-based dialogs for basic features.
- PATROL Agent—the Knowledge Module needs an agent to run it. The console cannot run a PSL script or exhibit any KM features. An agent is a prerequisite for development.
- Editor—text editor to write PSL scripts, and also syntax checking features. Part of the Developer Console.
- PSL Debugger—GUI-based run-time debugger for PSL scripts. Part of the Developer Console.
- PSL Compiler—standalone PSL compiler called “`psl`,” which is an executable that is used to create PSL libraries or binaries.
- PSL Optimizer—optimization built into the PSL compilation phase.
- PSL Profiler—performance measurement and tuning tools, available either as menu commands or as the `AGENT_PROFILER` full KM package. This feature relies on the PSL profiler PSL APIs in the 3.2 agent release.

In addition, a variety of development tools are available in the PATROL Knowledge Module Toolkit, included in PDN membership:

- KM Lint Tool—find common errors in KM and PSL files by static analysis of code.
- Agent Error Log KM—find common runtime errors in KMs or PSL by looking in the agent's error log for the messages that signify these KM programming failures.
- KM Development Tools menu commands—menu commands on the computer icon that contain a number of development tools, including debugging, tracing, and profiling.
- VisualPSL—graphical tool to create PSL response dialogs without PSL coding.
- IDL to KM Builder—convert SNMP MIBs to Knowledge Modules.

All these tools facilitate the development of a Knowledge Module based around the standard form, which is fully PSL-enabled. The typical prototype Knowledge Module is written entirely in PSL. Larger projects can examine other features if the need arises.

20.1.1 Advanced Tools

The main kit that you would use to build a Knowledge Module is the PATROL KM Toolkit, outlined above. In most cases, although they are development tools, you do not need to use any of the following tools to build a basic KM:

- PATROL SNMP Toolkit
- PATROL Toolkit for Console Integration
- PATROL API

These tools might be needed for an advanced KM, but are generally optional and usually unnecessary. The PATROL SNMP Toolkit is used for KM development only when you want an advanced instrumented SNMP-based solution. The PATROL Toolkit for Console Integration or the PATROL API are used only for an advanced instrumented solution (non-SNMP) or a KM with an advanced multi-agent architecture. For the majority of KM development cases where you are writing a simple non-intrusive non-instrumented single-agent KM, you do not need any of these other pieces. You simply need the tools that allow you to write PSL, which are the PATROL Developer Console and the PATROL KM Toolkit tools.

For completeness, I mention a few other PATROL components that you do not need to use when building a Knowledge Module. The following components work automatically with any KM, and you do not need to use them in development:

- PATROLVIEW integrations—views of agents and KMs inside third-party consoles such as Tivoli TEC, HP OpenView, and CA-Unicenter.
- PATROLWATCH for Web Browsers—Web browser views of agents and KMs.
- PATROL Pathfinder—Windows Explorer view of agents and KMs.

- PATROL CLI—ASCII agent interface.
- PATROL History Loader KM—centralizing and archiving history data for reporting.
- Email/pager notification.
- Other Knowledge Modules (unless you are relying on one as a data source).

These core services come for free. The use of a Knowledge Module leverages the underlying architecture. Since all these services deal with objects, alarms, and events from all Knowledge Modules, the only thing the Knowledge Module developer has to do is create the objects and define alarm and event criteria for the application.

20.2 KM Development Sequence

The number of different elements of a Knowledge Module are somewhat daunting at first. However, not all of them are needed for building a basic Knowledge Module. The following Knowledge Module components are usually necessary for a basic prototype Knowledge Module:

- Discovery script—A PSL script that serves to find the application and create an icon to represent it.
- Parameters—PSL scripts that measure specific values about the application, such as its performance, capacity, load, and so on.
- Alarm Range Thresholds—Simple numeric ranges that specify when a parameter will go into alarm because a parameter value is in a dangerous condition.
- Menu commands or tasks—Administrative actions that the user can perform from the console in an ad hoc manner. They can be written in PSL or in some other language.

The following KM components are desirable to achieve a more polished KM:

- InfoBox commands—quick popup reports available via MB2 on UNIX or a choice from the right mouse button in the Windows NT console.
- Pre-discovery scripts—a brief script to check for simple requirements, such as whether the KM is running on the correct platform agent (i.e., a KM for an NT application should not attempt to run on a UNIX agent).
- Recovery commands or tasks—automated actions to correct problems that are alerts. Sometimes recovery actions are also used for notification via pager, email, SNMP traps, or events. The KM developer should provide only true diagnostic reports and corrective actions and leave these “notification” actions to the administrator to define.
- PSL response function—the only way to create interactive GUI dialogs for PSL menu commands. This is portable to both UNIX and Windows NT consoles for PATROL.

The following Knowledge Module components are rather advanced and usually needed only for more complex Knowledge Module development tasks. However, some of these are required for formal certification of the Knowledge Module:

- Environment variables—system environment variables to be passed to system child processes that are launched by the Knowledge Module’s PSL scripts.
- Command Types—basic templates that extend the command types beyond “OS” and “PSL.”
- Global Channels—an efficient method of accessing data that is available through some interactive command such as an application-specific CLI.
- PSL libraries—a method of grouping common user PSL functions into one file to be shared. This is good programming style for larger KMs.
- PSL binaries—The creation of a binary format to offer performance enhancement and protect intellectual property. This is an optional choice and usually the last phase of development and packaging.
- Nested instances—A feature of the PSL `create` function to allow nested drill-down hierarchies for Knowledge Modules that create many instance and sub-instance objects.
- Online help—text help that the console can display about the Knowledge Module and its parameters.

Not everything within PATROL is valuable to the Knowledge Module developer. Here are some things that are best to avoid:

- Computer Classes—The KM developer is better to stay with Application Classes.
- Setup commands—Once-only commands that are available only for “Computer Classes.” These are best avoided for an ordinary KM.
- State change actions—Console-side OS commands that are used sometimes for notification, and rarely offer any value to the KM developer.
- Simple Discovery—Better to use PSL discovery.
- Odd PSL functions—`in_transition`, `full_discovery`

20.3 Creating an Application Class

The first step to building a new Knowledge Module is to create a new “Application Class.” This is really the name of the Knowledge Module as seen by the existing names such as “DISK,” “NETWORK,” “ORACLE7,” and so on. For details, refer to the PATROL documentation. In brief, the UNIX console allows you to add an application class from the list of application classes via the Edit menu, and the Windows NT console has a pulldown menu “New...” on the “Application Classes” line in the KM outline view.

The application class is the basis of the Knowledge Module. The simplest case requires only one class. For example, to monitor a machine’s CPU usage, only one class is needed in the

CPU or NT_CPU application classes. Once you get into more advanced Knowledge Modules, however, you need multiple application classes. For example, the PATROL KM for Oracle includes different application classes for the database server, tablespaces, and users.

20.4 Discovery Script

The purpose of the discovery script is to find the application and create an icon to represent it. The first piece of advice regarding discovery is to avoid the “Simple Discovery” option. It is usually too functionally weak to do what you want. Use the PSL Discovery choice instead. The simplest PSL discovery script is one line to create an icon. This looks something like:

```
create("fred", "", OK);
```

The arguments to PSL `create` are instance name, instance label, and initial status. Hence, this code creates an instance called “fred,” defaults the label shown in the console to the same as the instance name, and starts in an OK state indicating no current problems. A common error in learning PSL is to try simply one argument as follows:

```
create("fred"); # WRONG!
```

This will create an instance, but it is hidden because the default state in the absence of a 3rd argument is “NEW,” which is an invisible state.

This one-liner PSL script simply creates a new instance within the computer icon. You have to drill down into a computer icon for each agent to see this new icon.

One small point of note is that this one-liner PSL script is actually executed repeatedly, according to the discovery cycle. By default, it is 40 seconds. However, only one icon is ever created, because PSL `create` will not create a second icon if one already exists with the same name. Nevertheless, a slightly more pedantic form of the above script is to check for existence before creating, such as:

```
if (!exists("fred")) { # not already exists
    create("fred", "", OK); # create a new instance
}
```

20.4.1 Conditional Discovery

Both of the above discovery scripts are weak in that they assume that the software application is present and they simply create an icon. Clearly this is a good starting point, and you might want to move onto building some parameters just to see how that works. You could then return to the following discussion of the more advanced methods of building a more useful discovery script that tests for the presence of the application. Some of the common tests for an application’s presence include:

- Files
- Processes
- Registry (Windows NT)
- SNMP MIB

The simplest method to check for an application's presence is to test that a file exists. The PSL code using the "file" function is simply:

```
if (file("/etc/myapp")) { # UNIX style
    # file exists
    create("myapp", "", OK); # Create the icon
}
```

Processes are also simple to test for, using the PSL "process" built-in function. The PSL code can be similar to:

```
if (process("myserver")) { # UNIX or NT
    # Process existing matching regular expression 'myserver'
    create("myserver", "", OK);
}
```

The Windows NT registry can be tested via the PSL internal function. For example, the presence of a registry key may indicate that the application is installed. The PSL text can be:

```
if (internal("GetRegistryValue SOFTWARE\\MYAPP\\version") != "") {
    # Registry entry exists...
    create("myapp", "", OK);
}
```

The existence of an SNMP MIB variable can be the basis of discovery. This can be tested via the PSL SNMP functions:

```
mibvar = ".1.2.3"; # application-specific OID
ret = snmp_h_get("localhost", mibvar);
if (errno == 0) {
    # No error - success!
    create("myapp", "", OK);
}
```

These are all examples of very simple discovery logic. More complex issues can be easily coded by longer PSL scripts. Be sure to remember that the PSL discovery script is repeatedly executed at a default interval of 40 seconds. The discovery interval can be customized via the application properties editing dialog in the Developer Console.

20.5 Parameters

Parameters are the measured metrics for an application's performance, load, capacity, size, or other values. A typical Knowledge Module defines a number of parameter scripts that measure various metrics. As soon as an instance is created, all the parameter scripts begin to run to measure the values. These parameter scripts are regularly scheduled according to their polltime, which is a specifiable attribute in the Scheduling properties of a parameter script. The basic method of setting a parameter is to use the PSL `set` function to set a special "value" symbol. A dummy test parameter is as simple a PSL script as:

```
set("value", 10);    # parameter numerical value is 10
```

In reality, the value must be computed using data from the application that is gathered and processed via PSL. Various sources of data exist for KM parameters. Examples of common ways to compute parameter values include:

- File size—use the PSL `file` function as in "`ntharg(file("f.txt", 1), 5)`" to compute it.
- SNMP MIB value—use the PSL `snmp_get` function.
- Command output—use the PSL `system` or `execute` functions.
- Performance database value (Windows NT)—use the PSL `internal` function.

The most common form of parameter is a numerical parameter. This can be viewed as a graph or a gauge, and history is automatically collected. Numerical parameters are easy to assign thresholds for alarms, as discussed on page 286. However, text parameters are also useful to record simple textual results. Text parameters are defined via the parameter editing dialog as that type, and are set using the usual PSL `set` method.

```
set("value", "This is a text parameter!");
```

Here are some other more interesting examples:

```
set("value", process("*")); # List system processes; any platform
set("value", system("ls")); # List of UNIX files
```

More complicated parameter expressions are possible and common. For example, a parameter can easily be some dependent relationship on one or many other parameters. When computing parameters, you might record the absolute value, the relative value, and an averaged value. Be sure to get the right balance of meaningful data while at the same time ensuring that the Knowledge Module does not merely show a lot of similar values from a small amount of collected data.

One common advanced idiom is for a “collector” parameter to gather a lot of data, and then set a multitude of “consumer” parameters. This is an efficient solution in many cases. Explicit support exists for this model in the Development Console by defining the parameters as Consumer, Collector, or Standard.

20.6 Alarm Range Thresholds

Alarm range thresholds define for what values a parameter will issue an alarm. Assigning thresholds to parameters in Knowledge Modules is an important part of producing one. The aim is to set the thresholds to reasonable values for the majority of situations. This will reduce the need to customize the thresholds at user sites.

A related aspect of alarm generation is deciding on the mode to use. For example, reporting the alarm may make sense only after the threshold has recurred for a few times in a sequence. This helps to avoid one-off spikes that are not worth reporting. This type of consideration applies to the more dynamic measures such as CPU utilization, memory, and so on.

Text parameters are a special case. No numeric value exists with which to assign a threshold setting. Instead, the PSL script must set the status directly if you want to show an alarm. This is possible via PSL `set` on the built-in “status” flag. An example is simply:

```
set("status", ALARM); # Set text parameter to ALARM
```

This feature allows you to have a text parameter that represents a set of data and still show a meaningful status. For example, the text parameter could contain the log file text and alarm only if an error occurs. This code can be as simple as:

```
text = cat("/etc/logs/myapp.log"); # Read the entire file
set("value", text); # Store the value
if (grep("Error", text) != "") {
    set("status", ALARM);
}
else {
    set("status", OK);
}
```

This example is a little naive. It does not scale if the log file is too large, because PSL `cat` brings the entire file contents into memory and may grow the agent process size. A more mature version would tail the file using the external “tail” command on UNIX, or PSL `fopen`, `fseek`, and `read` functions to read only portions of the file into memory at a time.

20.7 Menu Commands and Tasks

A Knowledge Module that is purely for monitoring and alarming does not actually need menu commands. These are administrative actions that the console user runs via a mouse pulldown

menu off the instance icons. Note that the menu command shows up on the “instance” icon, which is not always the top-level icon. The menu entry appears on the right mouse button, directly off the icon on the UNIX console, or underneath the “KM commands” sub-menu on the Windows NT console.

The simplest type of menu command is one that just prints out data to the console window. This can be as simple as:

```
print("Hello World");
```

This output will appear in the console’s system output window for the host. Alternatively, you can check the “Task” type rather than “Command,” and check the “Pop Up Window Automatically” button. In this case, when you run the menu command, it will automatically pop up a new task window on the console. You can, of course, print out just about anything. Here’s an example:

```
print(cat("/etc/services")); # Print a file.
```

One common form of administrative command is that for debugging the KM itself. A common requirement is to update the values of a KM. This is not built into the console, and the KM developer can add a menu entry to refresh parameters via an OS command with text:

```
%REFRESH_PARAMS
```

Similarly, you can add a menu command to refresh application discovery. This runs the discovery scripts immediately, but unfortunately cannot be limited to just your KM, and updates all of them. The OS command is simply:

```
%REFRESH_APPL_DISC
```

Another to update the agent’s process cache that is used by the PSL `process` function is simply an OS command:

```
%REFRESH_PROC_CACHE
```

These are the simpler PATROL internal commands. More useful are application administration commands. For example, you might already have a UNIX shell script that you want to run via a menu command. This could be as simple as a PSL command using:

```
system("myscript");
```

This works, assuming that the `PATH` is set correctly to find the script. Error checking code is desirable for this menu command.

Another possible use of menu commands is to print textual reports. This uses the PSL `print` or `printf` function to print text. For example, you could print the results of the above command:

```
ret = system("myscript");  
print(ret);
```

A more advanced form involves some post-processing of the output to provide additional status information:

```
ret = system("myscript");  
print("MYKM: Command executed at time: ", date(), "\n");  
print("MYKM: Results of myscript command are:\n", ret, "\n");
```

Text output makes the distinction between a menu command and a menu task important. The menu command editing dialog allows you to specify that a menu is either a task or a non-task command. A menu task is different from a command in that a task pops up a special window just for the menu output.

The output for a menu command goes to the computer icon's system output window. Where you prefer the output to appear is a matter of style. A task is often preferred if you want the user to see the feedback of a window popping up with the results.

More advanced menu commands often require interface with the user. The PSL `response` function is the main method to ask the console user questions. Unfortunately, the syntax of the `response` function is difficult to learn, although the function is quite powerful.

20.8 Info Box Commands

Polished Knowledge Modules have useful information in the Info Box popups rather than relying on the defaults. The Info Box is available as a quick popup in the PATROL Console. On the UNIX console, it is opened via MB2 on any icon. On the NT console, it is an entry in the pull-down menu for each icon. In both consoles, it brings up a two-column text box containing the name of the items and their values.

The Info Box has two types of entries: built-in entries at the top that are hard-coded into the console, and KM-specific entries at the bottom that are those added to the KM by the developer. These allow the Info Box to be extended. Some typical values that might be printed in the Info Box for your KM include:

- Application version
- Application's installed directory name
- Knowledge Module version—what version of the KM is running?
- Quick status check—What is the application's status, and what is a one-line description of the cause of an alarm, if any?

The value shown in the right column is set via a PSL print command. Anything that a PSL Info Box script prints appears in the value column. For example, the Knowledge Module version might simply be a PSL command type with a `print` command that you change each time you revise the Knowledge Module:

```
print("Myapp KM version 5.7, revised 7/29/97");
```

More complicated ideas are to perform basic computations and then print them. For example, the application version might pull a computation out of the registry via PSL `internal` on NT, and then print it. Generally, the Info Box commands should show pre-computed values to avoid delays showing the results. A better way to offer the application version Info Box would be to set it to an agent symbol variable during pre-discovery or discovery, and then use PSL `get` in the Info Box command such as:

```
print(get("/myapp/the_version"));
```

You might also see this type of simple access as an OS command type using the agent's builtin `%ECHO` macro. The efficiency of this is basically the same as the PSL version. The OS version looks like:

```
%ECHO %{/myapp/the_version}
```

This simple form of getting and printing a value is most common, whether it is via PSL or OS types. However, any script is possible if you wish. You could even pop up a PSL response GUI dialog from an Info Box command for a neat effect.

20.9 Pre-Discovery Script

The pre-discovery script is an optional component of a KM. Typically, it is used as a stylistic aid to perform some basic checks. The most common check is to ensure that the KM is not running on an incorrect agent platform by examining the special `"/appType"` variable. For example, a script to disable the KM on any non-NT agent is simply:

```
if (get("/appType") == "NT") {
    set("active", 2); # Success - move to discovery
}
else {
    print("MYAPP: Disabling KM for non-NT agent\n");
    set("active", 0); # Failure; not NT - disable KM
}
```

The `"/appType"` variable is covered in more detail in "Detecting Machine Platforms" on page 365. This can be used to test for NT, OS/2, VMS, UNIX, or other variants. Other methods

of testing the version of the operating system are also available, such as to distinguish NT 3.51 and 4.0, covered in “Detecting Operating System Versions” on page 367.

Another less common simple check is to examine the builtin variable “/patrolVersion” to determine whether the agent is advanced enough to run the Knowledge Module. This is useful if the Knowledge Module requires 3.1 or 3.2 PSL features in other parts of the Knowledge Module.

20.10 Advanced Knowledge Module Features

You have much more to consider for a full Knowledge Module. However, for in-house applications the above discussion can often be adequate. The main purpose of the Knowledge Module is monitoring via parameters, alert generation via alarm ranges, and administrative commands via menu commands. Some of the other areas to consider include:

- Recovery actions—automated corrective actions that get launched when a parameter goes into alarm. These might fix problems automatically, or just be reports with more diagnosis information.
- Annotated data points for parameters via the PSL `annotate` function, perhaps launched as a recovery action, to provide more postmortem diagnosis information via drill-down.
- More advanced menu GUI interfaces via the PSL `response` function.

For a very advanced Knowledge Module development effort, some other features that might become desirable include:

- Event classes—extend the types of events that the KM can generate by providing a custom event catalog for the application class.
- Multiple application classes—create different sub-objects within the application view for items such as tablespaces, tables, users, processes, locks, and any other logically separate entity that is worth monitoring.
- Nested instances—create neater drill-down views for the sub-objects.
- Online help—user help text that helps the operator know what the Knowledge Module is monitoring, and what to do when a parameter goes red.

Many other aspects are involved in advanced Knowledge Module design and coding. Subsequent chapters address many of these areas.

KM Efficiency

This chapter addresses the issues of how to write a Knowledge Module (KM) so as to minimize the performance impact on the monitored system and application. This is a general discussion of the development tips toward more efficient Knowledge Module implementation. I assume that you have a Knowledge Module that you are currently designing or coding, and want its resource requirements to be minimal.

Performance tuning is something that developers seem to enjoy. No matter what environment or what application is involved, something magical lies in making the thing go faster. And no matter how good the automatic optimizer technology becomes, it still won't beat the use of good old human ingenuity. People are very good at tuning code.

Unfortunately, people are also very bad at tuning code. In particular, people are particularly poor at identifying where the performance bottlenecks are. Therefore, the first law of performance tuning should probably be:

Measure before you tune!

Only by measurement of the true performance characteristics of an application can you see where to focus tuning efforts. The same idea of measurement can avoid the Knowledge Module developer spending hours tweaking a PSL script that is run only once, and leaving alone another script that runs repeatedly. Various PSL profiling tools are available and should be used for measurement.

Another area of tuning that people often get wrong is retaining semantics. By tweaking the code to use particular tricks, you can often yield performance gains. By the same localized tweaking, you can also accidentally change the semantics, and introduce bugs. The second law of tuning should probably therefore read:

Don't micro-optimize!

Making localized small changes to a Knowledge Module or PSL is not always that useful, and carries with it a risk like any other software change. Usually, larger gains are to be made by changes to a Knowledge Module design rather than to a single PSL script. However, exceptions occur, such as if measurement shows that this one script, and one loop in this one script, is where most of the time is spent.

21.1 KM Design

Performance tuning for a Knowledge Module is often treated as a PSL tuning exercise. However, PSL is a low-level implementation of the high-level Knowledge Module design. A number of design choices can be made regarding the overall structure of the Knowledge Module that have a far greater effect on performance than PSL tweaking. Some of the design considerations involve:

1. Collector and consumer parameters—By limiting the number of executable collector and standard parameters by using consumer parameters instead, the overall number of PSL scripts is reduced.
2. Parameter Polltimes—Parameters run continually at a scheduled frequency. By changing the frequency at which a parameter executes, its impact on the system can be reduced on average. The overall cost is execution time by frequency, so either of these factors can be optimized.
3. Discovery cycle—The discovery script also runs repeatedly, and reducing its execution frequency will reduce the cost of the discovery process over time. Execution cost is also a factor, of course.
4. Pre-Discovery cycle—The pre-discovery cycle can run repeatedly under some conditions, and in these cases, similar comments apply to pre-discovery as for discovery.
5. Global channels—The use of global channels can reduce the number of child processes used by the KM and limit the number of redundant processes accessing the database or application.
6. Command types—Careful use of the command type facility can reduce child processes.

On the other hand, some areas of the KM do not bear much fruit in terms of performance improvement. A number of areas are commonly mistaken to be performance bottlenecks. However, because of the PATROL architecture or implementation of the particular subsystem, these do not impact performance:

1. Blocking PSL functions—A number of PSL functions have a blocking implementation as far as their PSL semantics. For example, PSL `read`, `sleep`, and `lock` block their PSL script until their needs are met. However, although one PSL process is blocked, this fact

does not put the PATROL agent process into a blocked state or a busy loop because of the internal agent PSL process scheduling algorithms. In fact, these PSL functions have no performance impact at all while they are waiting.

2. PSL process-like operations that are actually internal—A number of PSL built-in functions have names that are similar to common processes, such as PSL `grep` and `tail` functions. However, these PSL functions are actually efficient internal string operations and do not launch a child process. The only PSL functions that launch a child process are PSL `system`, `execute`, and `popen`.
3. Menu and InfoBox commands—Since menu commands and info box items are not executed unless the console user requests them, they are not usually worth tuning. Although these commands can be tuned like any other, tuning them is not as beneficial as tuning the scripts that run continually, which are the discovery scripts and parameters.
4. Recovery actions—Since recovery actions run only when a problem occurs with the application, which is hopefully rarely, generally not a large overall KM performance gain results by tuning the recovery action scripts. Although the speed with which a recovery action completes can be improved, which is by no means unimportant, the overall KM performance is not improved greatly on average.
5. Number of consumer parameters—Since consumer parameters have no runtime presence, they cannot consume CPU time, and whether a collector sets 10 consumers or 20 does not matter much. You have the issue regarding increased time for history storage for each consumer parameter, but this is usually marginal. Of course, consumers do also consume some agent memory and are not completely for free.
6. Once-only parameters—Not all parameters are re-scheduled forever. Once-only or other rarely executed parameters are not an area usually worth tuning.
7. PSL binaries—The PSL compiler can be used to create binary versions of the PSL scripts. These are files with a “.bin” suffix that are separate from the PSL scripts.
8. PSL libraries—PSL library files are very similar to binary files, but have a suffix of “.lib”. These are used to compile common functions into a binary format.

21.1.1 Collector and Consumer Parameters

A KM can choose from three different types of parameters. Each parameter type has different execution semantics and runtime performance impact:

1. Standard—This parameter runs a script and generates a value for itself (and also possibly others). It has an icon on the console to represent the parameter.
2. Collector—This parameter runs a script but has no value for itself. It is used to set values of other collector parameters. No icon is on the console display for a collector parameter.
3. Consumer—This parameter does not execute and has no associated PSL process. It passively waits for values from a collector. An icon on the console represents the parameter.

Performance improvement is clearly possible by allowing one collector to set multiple consumer parameters. In this way, only a PSL process occurs for the collector parameter. In addition to reducing the number of PSL processes, usually you have the advantage of collecting the data only once from the application. Without collectors, each parameter would probably be running similar queries against the application or database to generate related statistics. With collectors, the data can be collected once and used to populate many parameters.

With most applications, to return 10 values at a time is not much more expensive than to return a single value at a time, and to return those 10 individual values sequentially from the application is certainly more expensive. For example, consider the problem of collecting statistics from a database. You can choose between one SQL query to get 10 values, or 10 queries to get the 10 values. Usually, to batch the requests into the single SQL query is significantly cheaper.

An interesting point to note is that PATROL actually allows a few deviations from this basic collector-consumer model. For example, a standard parameter can actually set consumer parameter values, in addition to its own value. This capability can be useful, for example, if you want to give a collector an error code or status value by making it a standard parameter.

21.1.2 Collectors for Multi-Instance Applications

The use of the collector-consumer model can significantly reduce the number of parameters executing for a KM object. However, if you have more than one object, such as more than one database server, one collector parameter is still running for each object. In some cases, you can optimize this situation by providing one collector parameter for the whole application, rather than one per instance object. This “global” collector is typically in another application class, such as the parent application class for a nested instances situation, or sometimes in a dummy application class.

An example of this dummy application class optimization appears in the PATROL KM for Unix. This has a special COLLECTORS application class that contains the collector parameters for the various UNIX objects. For example, a collector exists for the FILESYSTEM applications, rather than one for each object of this file. This example works well because the data for all filesystems can be collected from a single command or kernel query. The collector for the FILESYSTEM’s FSCapacity parameter can be filled for each object from a single result.

An example in which the multi-instance collector idea does not work appears in many database or application monitoring situations. You usually cannot have a single collector for all the parameters of multiple database servers, because the data path to get information from each server is so different.

However, even the database monitoring situation provides the possibility of using this global collector optimization because of nested instances. The collector for all the tablespace or user objects within a database can be in the database application class, and fill the values for each of the tablespace or user parameters. This approach is more efficient than each tablespace

or user object having its own collector parameter. You have one tablespace or user collector per database server instead of per object.

Let us examine this example in more detail. Assuming that we have a “SERVER” class that detects the database server, we also want to have “TABLESPACE” and “USER” objects monitored. Clearly, to measure the amount of space left in a tablespace, we need a collector parameter. We could put a collector parameter in the TABLESPACE class, but that would mean one collector parameter for each of the many tablespace objects. Here we see that the tablespace information is coming from one database server, so we could add a “Tablespace_Collector” parameter to the SERVER class, and it would run to set the parameters for every TABLESPACE object. In this way, we have reduced the number of collector parameters from one per TABLESPACE object to one per SERVER object. Exactly the same idea applies to the USER objects. Since the analysis is of how much data or resources each USER object is using, we can do all that in one single collector parameter in the SERVER class, rather than putting a collector in the USER class itself.

21.1.3 Global Channels

The use of a PSL global channel is a design advantage for a KM where an external process is used to access the application. For example, a KM may use an application-specific command-line tool to access the database or application to read data from it. For such a tool, the least performance-effective method is to repeatedly launch it via PSL `system` or `execute` to collect each query. The more effective method is to use PSL `popen` to create a local channel to the executable, use PSL `write` to send commands down to the executable, and use PSL `read` to gather the output. This local channel allows repeated commands to be sent down to the executable, and permits the channel to stay open, thus avoiding the cost of process spawning. If only one collector parameter is using this channel, it can remain a local channel.

However, if more than one PSL process, such as more than one collector, needs to get information from the application, or if menu commands also need to send commands to the application, this is not efficient. A local channel can be accessed only by the PSL process that spawned the child via the PSL `popen` function. In this case, the more performance effective approach is to use the PSL “share” function to convert the local channel into a global channel. By using only one channel to the application, the impact on the application is minimized, and the system cost of process spawning is also minimized.

An example of when a global channel becomes valuable appears in a simple database monitoring case. If we have a SERVER, TABLESPACE, and USER class for database monitoring, they all need to send queries to the database. The inefficient way is for each class to have its own channel to the database, which would probably launch one child process per class. The efficient way is for only a single global channel to exist, probably in the SERVER class, and then all the classes use it to send queries to the monitored database server. The global channel idea involves only a single resulting child process.

The global channel is typically a pipe into an interactive child process. This child process could be as simple as a system command shell, a vendor-supplied command-line tool for a data-base or application, or a self-written tool for an in-house application.

The use of PSL global channels is effective for performance improvement only when the required conditions arise involving multiple PSL scripts and multiple child processes. The channel is an internal agent data structure and has no actual physical runtime presence on the machine. Certainly, the external child process launched by PSL `popen` has a presence, but the use of a global channel versus a local channel makes no difference to performance for a single PSL process. The use of a global channel has a performance advantage in that it allows one child process to be used by multiple PSL processes, and no negative performance overhead of the global channel structure balances this at all.

21.1.4 Parameter Polltimes

For each parameter, be it of type standard or collector, the polltime is the scheduling of how frequently the parameter data is collected or polled. Changing how frequently parameters are polled has an effect on the work load for the agent over time. The good news is that the cost of retrieving data for a parameter is much the same from poll time to poll time. However, the polltime also translates directly into how up-to-date the parameter is, and a trade-off occurs between performance and freshness.

The polltime for a parameter can be changed via the scheduling settings in the parameter editing dialog in the Developer console. For a less permanent setting, the polltime can be changed via PSL `set` on the “interval” or “pollingInterval” variables for a parameter object.

Note that although the cost of retrieving the data from poll time to poll time is pretty much the same on any system, this is not the case for the same parameters on different sized systems. Collection performance differs between applications on departmental, mid-range, and enterprise servers. Therefore, sometimes you would want to have different polltimes on different systems. However, this issue is not a KM development issue, but one of KM configuration.

21.1.5 Discovery Cycle Time versus Full Discovery

The discovery script runs repeatedly for each application. One common cause of inefficient KM design is the confusion surrounding the distinction between “partial discovery” and “full discovery.” According to the PATROL manuals, partial discovery runs every 40 seconds, and full discovery every 300 seconds. The natural assumption is that when you write a Discovery Script for your KM, it will do something “light” every 40 seconds, and execute fully every 300 seconds. In fact, the script will execute fully every 40 seconds unless you explicitly ensure that it does not. If the KM is doing non-trivial work in the discovery script, this execution is an unexpected performance hit.

The partial and full discovery distinction was introduced for “Simple Discovery” KM applications, and does not apply by default to “PSL Discovery” scripts. Unless you explicitly

code some early exit or other “light” version for the partial discovery cycle, the PSL script will be executed in its entirety every 40 seconds. The method for checking for a full discovery cycle is via the PSL function called “full_discovery.” For example, you could ensure that your KM discovery script exits early in partial discovery via a simple test:

```
if (!full_discovery()) {  
    exit; # exit early for partial discovery cycle  
}
```

However, this is often not the best solution. This solution is necessary only if you are explicitly using the “process cache,” on which the full discovery cycle is based. You can use the process cache only by using the PSL `process` function, and if you are not using this function as part of your discovery script, to test for full discovery to control the scheduling is pointless.

If you are not using the process cache, increase the 40-second discovery cycle via the GUI setting in the application editing dialog. The basic KM editing dialog has a set of timer buttons to specify the time period for the “Discovery Check Cycle,” available since PATROL 3.0’s initial release. This controls how frequently the discovery PSL script executes, and has a direct effect on the performance of the KM. This is a clearer control over the performance of the KM and can be tuned independently of the full discovery cycle. Users of the KM can also later tune the discovery cycle more easily if necessary. By using this method, you can tune your KM without tying yourself to the obscurities of the partial and full discovery cycles.

21.1.6 Pre-Discovery Cycle Time

The execution scheduling of a PSL “pre-discovery script” is almost identical to the discovery script. Generally speaking, the pre-discovery script should be brief and either move on to full discovery script via setting “active” to 2, or deactivate the application by setting “active” to 0.

However, if the pre-discovery script does not explicitly set the active flag to either 2 or 0, it remains at 1, and the execution cycle of the pre-discovery script continues. This is also 40 seconds by default, and is not affected by the partial or full discovery cycles unless explicitly coded. As mentioned for the discovery script, the preferred solution to controlling the execution frequency of pre-discovery is to modify the “Discovery Check Cycle” in the application dialog.

However, in the case of pre-discovery, the best solution is usually that the pre-discovery script should not run repeatedly, but should either start discovery or deactivate the application. For example, if pre-discovery detects the KM running on the wrong agent platform, it should deactivate itself.

Unfortunately, the situation is somewhat blurred if the script does not find the software installed or running on the system. Should the application continue to check for the application software? This continuous check achieves “automatic discovery” whenever the application is installed or started. But doing so does waste some agent resources. This issue is a matter of judgment on a per-KM basis.

21.1.7 History Storage Tuning

The cost of storing parameter values in the history database has some effect on agent performance and disk space usage. However, the effect on runtime performance is usually minimal, and the disk space used is approximately eight bytes per parameter value for the value and timestamp, with slight extra overhead for indexing. Tuning the history storage issue is a marginal improvement in some cases.

Parameter polltime changing is one way to reduce history storage. By reducing the frequency of parameter sampling, the number of values stored in the history database over time declines. This optimization has already been discussed on page 296.

Disabling history storage for a parameter is another possible tuning mechanism. Each parameter has a history level configuration setting. This can be edited via the Developer Console GUI in the parameter editing dialog. By changing the setting from “Inherited” to “Local,” and the number of days to “0,” you can ensure that no parameter values are stored in history for a particular parameter. This is a possible choice for less important parameters or parameters for which history has no particular meaning. On the other hand, in good KM style, parameters offer history for analysis, and turning off history storage should perhaps be left to a later KM configuration stage if performance becomes an issue.

Annotations are also stored in the history database and take disk space approximately equal to the number of text characters. Therefore, the Knowledge Module developer should also keep in mind the frequency and size of text annotations stored for parameters via the PSL `annotate` function.

21.1.8 Event Storage and Notification Tuning

The event subsystem and the event database are usually not very costly. The only exception to this case would be a KM that makes over-use of a user-defined KM event catalog for KM-specific events. For example, if the KM is pumping hundreds of events into the event subsystem, a performance bottleneck can result. However, reasonable use of KM-defined events is not usually a cost concern.

One of the hidden costs of storing events is that each event can potentially execute a notification script. The event catalog allows definition of scripts for “Notification,” “Acknowledgment,” and “Escalation.” Therefore, if a lot of events are stored and these events start running scripts, these can add to the agent load. When adding notification scripts or other scripts to the KM-specific event catalog, keep performance concerns in mind. However, a KM rarely needs to add such notification commands to the custom event catalog and certainly never needs to add them to the Standard Event Catalog.

21.1.9 Child Process Launching

The number of child processes launched by a KM has an impact on the overall system performance of the monitored system. Reducing the number or frequency of child process executions can improve performance significantly. Using global channels is the best method.

To reduce child process impacts, we need to know where the child processes come from. The KM can launch child processes from a number of avenues:

- PSL `system`
- PSL `execute`
- PSL `popen`
- OS commands
- Commands of other command types—Non-OS and non-PSL commands

The PSL profiler menu commands have one report that can be used to measure child process launches. This is available in the Development Tools menu commands on the computer icon. This report determines how many calls were made to PSL `system`, `execute`, and `popen` by each PSL script. This report is fully accurate in most cases, but some exceptions occur if the KM uses PSL `system` or `execute` to launch internal agent built-in commands, such as `%DUMP`, which do not actually launch a child process. Also, the `AGENT_PROFILER` tool in the PATROL Knowledge Module Toolkit does similar reports in a full KM rather than menu command format. Both the menu command and KM version are available in the KM Toolkit from the PDN.

One common pitfall with child processes is their use in pre-discovery or discovery. If the pre-discovery or discovery scheme requires the execution of a child process, this will always be spawned. However, if the software is not installed on that machine, the child executable may not even exist, leading to pointless executions that are costly. The solution is to check the PSL `errno` variable for returned status and check the text results for cases where the child executable failed to execute, and discontinue this use elegantly if so.

The use of special command types is one method of reducing the overall number of child processes on UNIX. Under some conditions, the UNIX agent will launch a child process to be executed by a shell for reliability reasons. So both a shell process and a child process are launched. The use of a special “Command Type” can avoid the shell process if it is clearly not needed. This optimization is relevant only to the UNIX agent.

21.1.10 Memory Usage

The memory consumption of the PATROL agent process is affected by KM operations. Each instance and parameter object uses space, hence creating thousands of instances and possibly imposing a high memory burden on the agent. However, the number of such objects is usually reasonable, and the general change in memory consumption is marginal.

The other form of memory usage is in the use of PSL variables. Certain other pitfalls must be avoided. For example, consider the script:

```
text = cat(filename);  
results = grep(pattern, text);
```


The PSL `cat` function copies an entire file into a PSL string variable in memory. If the file is too large, such as many megabytes in size, it will grow the agent's memory requirements significantly. One solution to this performance problem is possible—to use PSL `file` to test the file size and use PSL `system` to launch an external “`grep`” command, rather than an internal `grep` function if the file is too large. Unfortunately, this solution does not work in environments where the `grep` command is not standard.

```
MAXSIZE = 100000;      # 100K size
if (ntharg(file(filename, 1), 5) < MAXSIZE ) {
    # Use the internal grep function
    text = cat(filename);
    results = grep(pattern, text);
}
else {
    # Use the external grep process
    results = system("grep '" . pattern . "' " . filename);
}
```

This example shows that PSL variables can consume space. Fortunately, these PSL variables are cleaned up when the PSL script terminates, avoiding the need to explicitly free memory. However, for long-running scripts, to explicitly set variables to the empty string can sometimes be useful if they are large. This makes the memory available again for other agent actions, or potentially available to the operating system.

Persistent string values in the agent symbol table from PSL “`set`” are not cleaned up automatically. Therefore, the storage of large strings of data in the symbol table, such as command output results, can increase agent memory usage on a one-to-one basis. Large variables can be freed by setting them to the empty string or using the PSL `unset` function on temporary symbols. The agent can then re-use this memory for other data and thereby reduce its overall requirements.

PSL libraries also have a marginal effect on agent memory usage. For a widely used PSL function, storing it in a shared PSL library will keep only one copy of all the read-only segments of that function, including its code quad-codes, constants, and symbols. However, multiple copies are kept of its runtime data segments for each PSL process using the library. Although PSL libraries can have a marginal improvement on memory usage, the effect is not great, and PSL libraries should be considered for programming style issues rather than memory usage.

21.1.11 Network Traffic

Usually a Knowledge Module needs to do nothing to achieve efficient network traffic if it runs on a single host. This is more of an architectural issue in terms of the configuration of the agents and consoles. The autonomous agent architecture generally deals with network traffic optimization for a single-host KM.

A single-agent KM will probably not generate any untoward network traffic. Certainly, alarms will cause network traffic to connected consoles, but the KM should not be generating too many of those, or else it is not going to be of much use.

However, network traffic may be an issue for a couple of advanced KM architectures. These are KM architectures that involve more than one PATROL agent, such as proxy KMs or KMs for distributed multi-machine applications.

Proxy KM architectures may use network traffic in the methods to go to the remote application and extract data. In this case, the polltimes of collector parameters may actually have an effect on the frequency of data collection requests that provoke network traffic. Similarly, the discovery script may provoke network traffic, and tuning the discovery cycle may impact network traffic.

Multi-agent KM architectures may also cause some network traffic in order to aggregate data across multiple agents. The network traffic may occur via the PSL remote access functions, the PATROL API, or the PATROL CLI executable.

21.1.12 Knowledge Module Configuration Issues

Although the general issue of configuring KMs for performance is beyond our scope, the KM developer can offer a few points to simplify the process. By adding tuning features to the KM itself, you can greatly simplify on-site configuration. By opening up configuration interfaces or even interactive tuning menu commands, you can facilitate the tuning process. A number of possible avenues can be aided:

1. Pre-discovery or discovery looping—Make a configuration setting for whether discovery will fail when it fails to find the installed software, either immediately or after “n” times.
2. Annotations—Make a configuration variable to allow turning off annotations for parameters (simply by not calling the PSL `annotate` function if set), or to allow annotations only if the parameter has gone into alarm (which is testable via PSL `get` on “status”).
3. Polltimes—Usually you do not want to handle these explicitly, because the basic KM infrastructure allows changing them. However, you can also add your own configuration settings. For example, if you can determine automatically that this application is “big,” you might reduce the intervals automatically via PSL `set` on the “interval” variable.
4. History retention—This is also usually left to the infrastructure, and is difficult to control in any case.

21.2 Event-Driven versus Polling Architectures

A polling architecture refers to when the KM repeatedly asks the operating system or application for information to measure. An event-driven architecture is where the operating system or application will notify the KM about a change to a monitored value. Clearly, an event-driven architecture has the potential for better performance. However, an event-driven architecture is not always possible.

Consider a simple example of a parameter measuring the size of a file. The simple polling mechanism is for the KM parameter to check the size of the file via the PSL `file` function. It does this repeatedly and stores the value in the parameter history database. A better approach would be if the operating system would automatically tell us when the file has changed size, or better still, when the file reaches a certain threshold level. This would allow us to store values only when the file size changes. However, no general operating system mechanism where this can easily occur is portable, easy to access, and widely supported.

In some cases, you can achieve an event-driven architecture. The PSL `read` function when waiting for output from a global channel launched via PSL `popen` can yield an event-driven architecture. If the child process detects events, it can produce output, which then wakes up the PSL `read` function call as if it had just received notification about the event. By then processing the results from PSL `read`, you can set parameters using an event-driven non-polling architecture.

Event-driven architectures are more likely to be achieved by a cooperating application. By adding intrusive monitoring and management instrumentation to the application, you can achieve this goal. Intrusive instrumentation is discussed in “Application Instrumentation” on page 203. It involves issues of how to modify the monitored application to make itself more management-friendly to a KM that manages it.

PSL Performance Measurement and Tuning

The same general comments apply to PSL tuning as to code tuning in general. Measurement should be the first step to determine where the bottlenecks are. And these bottlenecks should be addressed by higher-level design or structural changes rather than micro-level code changes. In the discussion following, I examine measurement tools and the various macro and micro optimizations that are possible in PSL.

Two logical steps are in the measurement procedure. The first is to determine which PSL script in a Knowledge Module is consuming most resources to focus efforts on it. The second step is to localize the area within that PSL script to lines of code or a particular function or loop. Generally speaking, a small area of the PSL script is consuming most of the computation time.

The PSL profiler is the best method of measuring the top PSL processes. The PSL profiler is available in PATROL 3.2 releases. The profiler's PSL APIs are supported in the 3.2 agent and are layered upon by various PSL-based tools for reporting, including the `AGENT_PROFILER` tool in the PATROL KM Toolkit from the PATROL Developer Network. For PATROL 3.1 and earlier, a number of less accurate methods of measuring the top processes are based on sampling the agent's PSL process list via the `%PSLPS` command.

Once the PSL script is chosen for optimization, to determine which areas are heavy is not so easy. Although the PSL profiler does show built-in function call counts, it does not offer any low-level line-by-line or loop count analysis of the code. So the only method for measuring time usage within a script is to explicitly annotate blocks of the code with timing snippets of PSL code, which is effective, but time-consuming. In PATROL 3.2, this timing code can use the PSL functions that generate profiling reports, but in PATROL 3.1 and earlier, the main primitive is the PSL `time` function.

Another possibility for locating bottlenecks within the PSL script is analysis of the quad-codes for the script. The “-q” option to the ‘`psl`’ standalone compiler can generate a quad-code

listing. The PSL optimizer options “-OP2” and “-OP3” to the ‘psl’ compiler also produce useful reports related to quad efficiency.

22.1 PSL Processes List via %PSLPS

In the absence of the PSL profiler (i.e., for PATROL 3.1 and earlier), we can use simpler sampling methods. The list of PSL processes currently running in the agent can be sampled and used to generate a rough picture of the cost of each PSL process.

We can view the PatrolAgent’s PSL process list via the “%PSLPS” button in the PSL Debugger GUI, or by executing the command “%PSLPS” as a PSL command or task. Table 22-1 shows an example of the condensed results returned from the %PSLPS command.

Table 22-1 Sample results from %PSLPS command

PSL Process List						
PID	NAME	STATUS	TYPE	APPLICATION	INSTANCE	PARAMETER
18	SWAP	HALTED	PREDISC	SWAP	-	-
19	SWAP	HALTED	DISCOV	SWAP	-	-
20	SMP	HALTED	DISCOV	SMP	-	-
21	SECURITY	HALTED	PREDISC	SECURITY	-	-
22	SECURITY	HALTED	DISCOV	SECURITY	-	-
23	PROCESS	HALTED	PREDISC	PROCESS	-	-
24	PROCESS	HALTED	DISCOV	PROCESS	-	-
25	PRINTER	HALTED	PREDISC	PRINTER	-	-
26	PRINTER	HALTED	DISCOV	PRINTER	-	-
52	DFColl	HALTED	PARAM	COLLECTORS	COLLECTORS	DFColl
53	PrinterColl	HALTED	PARAM	COLLECTORS	COLLECTORS	PrinterColl
54	SARColl	HALTED	PARAM	COLLECTORS	COLLECTORS	SARColl
55	NETColl	HALTED	PARAM	COLLECTORS	COLLECTORS	NETColl
56	NFSColl	HALTED	PARAM	COLLECTORS	COLLECTORS	NFSColl
57	PSColl	HALTED	PARAM	COLLECTORS	COLLECTORS	PSColl
58	VMColl	HALTED	PARAM	COLLECTORS	COLLECTORS	VMColl
59	SMPColl	HALTED	PARAM	COLLECTORS	COLLECTORS	SMPColl
60	UPTColl	HALTED	PARAM	COLLECTORS	COLLECTORS	UPTColl
61	LOGFileSize	HALTED	PARAM	LOG	usr-adm-sulog	LOGFileSize
63	LOGFileSize	HALTED	PARAM	LOG	usr-adm-snmpd.log	LOGFileSize
64	LOGFileSize	HALTED	PARAM	LOG	usr-lib-cron-log	LOGFileSize
65	LOGFileSize	HALTED	PARAM	LOG	usr-spool-lp-log	LOGFileSize
107	ORACLE7	HALTED	PREDISC	ORACLE7	-	-
108	ORACLE7	HALTED	DISCOV	ORACLE7	-	-
TOTAL: 185						

The list in Table 22-1 is shortened quite considerably. As you can see, 185 processes in total should occur (note that this list is 185 internal PSL processes, not system processes). The PSL Process List shows some key attributes of each PSL process. Each PSL process has a

unique numerical process id, as shown in the `PID` column. Associated with the `PID` is the PSL process name, which is shown in the `NAME` column. Each process has a state shown in the `STATUS` column. Table 22-1 shows that all processes were in a state of `HALTED` at the time when the process listing was requested. This is quite normal and refers to them all in a condition of currently halted and waiting for their next scheduled execution. The `STATUS` column would have identified that a PSL process was being executed with a `RUNNING` status.

22.1.1 PSL Execution Counter Menu

Repeatedly running the `%PSLPS` command can give you useful information about the top PSL processes. This can be automated, and one example in PSL is the “PSL Execution Counter” menu command that is available in the menu commands under the “Development Tools” menu on the computer icon (if you have installed these tools from the PATROL KM Toolkit).

The PSL Execution Counter is used to get an understanding of the number of timeslice executions % time each PSL process is consuming within the agent. This counter needs to be run for a reasonable number of minutes before the results can be accurate, due to the cumulative nature of the report and the fact that different parameters have different poll times. The task should be run for at least the amount of time of the longest poll period for a parameter on that agent. The longer it is run, the more accurate it becomes.

Unfortunately, the PSL Execution Counter is dependent on two performance variables rather than only one. The first is the amount of time each PSL script is consuming. The second is the polltime that controls how frequently the script is rescheduled. Therefore, the highest PSL scripts can be “tuned” either by real code tuning or simply by changing the parameter polltime or discovery cycle time. Nevertheless, the measurements are a real estimate of the amount of CPU time that a script is consuming on average in real usage.

22.1.2 PSL Processes Watch Window Menu Commands

A number of useful menu commands are available in the Development Tools menus in the PATROL KM Toolkit. The PSL process monitor displays the top 10 currently `RUNNING` processes as well as the blocked I/O processes identified by the `IOWAIT` status type. This tool is great for showing any PSL processes that may continuously be in the `RUNNING` state. Further investigating these PSL processes would reveal why they are always in the `RUNNING` state. Perhaps their polltime needs looking into or the collection routine itself.

The “Deployment KM” also has a PSL process monitor that shows those PSL processes that are waiting on the external system processes, as well as the external system processes themselves. These PSL processes will typically be in a state of `IOWAIT`, which is to be expected.

These tools are very useful when needing a better understanding of the PSL processes on an agent. However, these tools can be expensive in regard to CPU usage because of their rapid sampling techniques. Care should be taken as to when the tools should be used for PSL process analysis.

22.2 PSL Profiler Tool

The PSL profiler is available in the PATROL 3.2 release of the agent. This is a measurement component built into the agent to measure how many resources each PSL script consumes. This allows the determination of which scripts have the largest cost, with some information as to where the cost is distributed within the PSL script. The PSL profiler has a number of interfaces:

1. PSL functions—The profiler can be dynamically enabled via PSL, and reports can be printed in PSL.
2. Menu commands—A number of profiler menu commands are available in the KM Toolkit that make use of the PSL primitives to produce useful profiling reports.
3. AGENT_PROFILER tool—A full KM using profiling functions in the KM Toolkit.
4. ppv utility—The ppv executable is an off-line reporting tool that analyzes a profiling data file that the agent can generate during profiling.

In all cases, the output report is of a similar format. The PSL profiler reports tell you which PSL scripts have the highest CPU usage, and also measure how much each built-in PSL function has used.

Scheduling frequency of the PSL processes clearly affects the result, as they are cumulative timers. However, since the number of executions is also measured, the average costs per execution can also be easily generated in reports.

The main agent resource measured by the PSL profiler is CPU time. This is broken into user time and system time, to indicate that the system calls may take some time in some of the PSL built-in functions. The profiler does not explicitly measure other cost measures such as the number of child processes, the amount of memory, PSL locks, global channels, and so on. However, some of this information can be gathered from the execution counts and the built-in function call counts. For example, child processes can be measured reasonable accurately as the number of calls to PSL `system`, `execute`, and `popen`.

The main limitation of the PSL profiler is that it does not go very deep within each script to examine performance. For example, no line-by-line report shows which lines are executed most. The analysis of built-in functions is useful, but no similar analysis of PSL user-defined functions occurs. The profiler can be dynamically enabled and disabled during agent execution. Enabling the profiler will turn on the measurement for the specified PSL scripts. Disabling the profiler will discard all measured data and return the agent to the normal non-profiling state.

The impact of turning on the PSL profiler is a marginal degradation in PSL execution performance, since each PSL timeslice must also add to measurement counters. However, this is mainly an in-memory cost that is low. The largest costs are when the reports are generated, and these reports are not occurring often because they are useful only when the performance data has been collected.

22.2.1 Profiling Menu Commands

Menu commands are not usually the first processes to tune in a KM because they are executed only on a one-off basis, whereas parameters and discovery run repeatedly. Nevertheless, to tune a menu command that appears to be taking too much time can be valuable.

Unfortunately, transient PSL processes such as menu commands are more difficult to measure than permanent processes. Parameters and discovery scripts are permanent because of their regular scheduling, and the PSL profiler can easily measure them.

The problem with menu commands is that the profiling reports will report only on the current list of PSL processes. And if the menu command has completed execution, its PSL process no longer exists.

One solution to profiling a menu command is the explicit addition of PSL profiling code to the menu command. The PSL functions to control the profiler can be used to turn on profiling, and generate a profiling report at the end of execution.

```
function start_timing()
{
    # turn on profiling via ProfOptions and getpid
    local mypid;
    mypid = getpid();
    ProfOptions(mypid, 15);    # Enable all profiling
}

function timing_report()
{
    # Print a profiling report...
    local mypid, text;
    mypid = getpid();
    # print reports using ProfGet
    text = ProfGet(mypid, 20);
    return text;
}

function stop_timing()
{
    # turn off profiling with ProfOptions
    local mypid;
    mypid = getpid();
    ProfOptions(mypid, 0);    # Disable all profiling
}

start_timing();
..... # Menu command main code
report_text = timing_report();
print(report_text);    # Print the report, if desired.
stop_timing();
```


This idea can be applied to any PSL script. In particular, this makes the technique useful for menu commands, info box commands, and event notification commands. These processes are all difficult to measure via normal profiling.

22.2.2 Profiler PSL Functions

The PSL profiler has a number of PSL functions to control the profiler. The profiler can be turned on and off via PSL, and reports can be generated. The PSL functions relating to the PSL profiler are:

- `ProfOptions`—turn profiling on and off for one, many, or all existing PSL processes.
- `ProfDefaultOptions`—handle profiling defaults for PSL processes not yet created.
- `ProfGet`—generate reports for one or many PSL processes.
- `ProfTop`—generate reports for one or many of the top “n” PSL processes.

These PSL built-in functions are documented fully in the PSL Reference Manual.

22.2.3 Profiler Console Menu Commands

A number of menu commands have been developed to control the PSL profiler functions and to generate more complex reports from the basic PSL reports. These are available in the “Development Tools” menu commands on computer icons from the KM Toolkit, and also in the “Deployment KM” menu commands, if they are installed. The main sequence of using these menu commands is:

1. Choose “Enable Profiler” to start profiling on all current PSL processes.
2. Wait! The results from profiling reports are not immediately useful. At least 5 minutes is recommended, but longer time periods make results more accurate.
3. Choose one or many of the reporting menu options. For example, reports are available about the top CPU consumers, child process launch reports, and PSL built-in function counts.
4. Choose “Disable Profiler” to return the agent to non-profiling state.

Nothing is magic about these menu commands. They are using the PSL interface to the profiler to control it, and using string operations to generate summarized or specialized reports. In fact, you can do a quick-and-dirty system output window method even if you do not have the tools:

1. `%PSL ProfOptions( "", 15 );`—Turn on profiling.
2. Wait!—The agent needs several minutes to record information.
3. `%PSL print( ProfTop( "", 10 ) );`—Print a top 10 report.
4. `%PSL ProfOptions( "", 0 );`—Turn off profiling.

Further use of the PSL profiler functions is a matter of coding. Examining the menu commands can give you ideas for extending the profiling ideas to your specific needs.

22.2.4 AGENT_PROFILER Tool

The AGENT_PROFILER tool is an alternative to using the menu profiling commands. It is available via the PDN in the KM Toolkit as a separate KM tool. AGENT_PROFILER is based upon the same PSL APIs in the 3.2 agent, but its reporting method is different from the menu reports. Rather than use a user-initiated menu command to start the profiling, the AGENT_PROFILER tool starts profiling once loaded. It sits in the background, measuring all CPU usage on a per-KM basis and a per-PSL-process basis. This means that the tuning data is available per-KM, per-PSL process, and further down to the PSL built-in functions that are used.

The profiling data gathered by AGENT_PROFILER is reported in some neat ways. The iconic view shows the top level and then KM icons with KM profiling reports, and drilling down again gets to the PSL processes individually. A lot of the reports are stored in text parameters. However, menu commands also show the same text in a popup task for easier printing.

The AGENT_PROFILER tool itself consumes a lot of CPU time initially, due to the large number of objects and sub-objects it creates. Over time, the CPU usage is much lower. However, because of the initial startup and the extra memory usage from the objects, the AGENT_PROFILER should be used only in a development environment. The menu commands for profiling are safer to use on an ad hoc basis for a brief period on a production site.

22.2.5 PSL Profiler and PPV Tool

Dynamic control of the PSL profiler via PSL is not the only method of profiler usage. The profiler can be enabled in a more controlled batch manner where a command-line option starts the agent in a profiling state, and the agent writes a data file on agent termination. This data file of profiling measurements can be used to generate reports from the standalone “ppv” profiler reporting tool.

Since the results of ppv are identical to those of the PSL profiling report functions such as ProfGet and ProfTop, and these can be done dynamically, usually no reason exists to use the more complicated process involving agent startup and shutdown. Nevertheless, some occasions are more suited to batch processing and off-line analysis.

The PSL profiler can be activated by the “-profiling” command-line option when the agent is started. The environment variable “PSL_PROF_LOG” must also be set with the name of a file in which to store the profiling measurements. This file is a binary data file generated by the agent that is used with the “ppv” utility.

The standalone interpreter executable “psl” can also run with profiling via the “-P” command-line option. However, don’t forget that not all PSL built-in functions work in the standalone interpreter. Therefore, off-line profiling using “psl” is not usually successful for many PSL scripts.

The profiling data is written to a binary file when the agent terminates (or when the PSL standalone interpreter completes). The `ppv` utility can be used to view the binary files. The `ppv` utility displays statistics from the agent for each PSL process executed. The statistics collected and displayed by `ppv` are as follows:

- Number of PSL processes the PatrolAgent profiled.
- Elapsed time of the PatrolAgent.
- Cumulative CPU time of the PatrolAgent.
- The average %CPU load of the PatrolAgent.

The PSL process statistics include:

- The number of executions of the process.
- The real time and the CPU time of each process.
- The number of calls, along with the real and CPU times for each PSL built-in function a process calls.

This information is displayed in descending order, displaying the process with the largest CPU first. This format of the `ppv` output is almost identical to that from the PSL `ProfTop` function.

22.2.6 Profiling Report Formats

The format of the PSL profiling reports is similar for the PSL `ProfGet` and `ProfTop` functions, and also the `ppv` standalone reporting utility. Below is an example report for a single PSL process.

```
Process Name: ProfGet.psl
Executions: 1 times
```

Process Summary:

Code Section	Real Time	User Time	System Time	%Proc	%Total
PSL	0:00:00.10	0:00:00.0	0:00:00.0	0.00	0.00
Native	0:00:00.0	0:00:00.0	0:00:00.0	0.00	0.00
TOTAL	0:00:00.10	0:00:00.0	0:00:00.0	100.0	0.00

Per-function Summary:

Name	# Calls	Real Time	User Time	System Time	%Proc	%Total
ProfOptions	1	0:00:00.0	0:00:00.0	0:00:00.0	0.00	0.00

getpid	1	0:00:00.0	0:00:00.0	0:00:00.0	0.00	0.00
getpname	1	0:00:00.0	0:00:00.0	0:00:00.0	0.00	0.00
nthargf	2	0:00:00.0	0:00:00.0	0:00:00.0	0.00	0.00
print	3	0:00:00.0	0:00:00.0	0:00:00.0	0.00	0.00
-----		-----	-----	-----	-----	-----

The “PSL” data row is the time spent interpreting all PSL instructions except for “native” (i.e., built-in) function calls. The “native” functions are the PSL built-in functions like `nthargf` and `grep` that are functions implemented in agent code, i.e., not user-defined PSL-coded functions. Actually, the “PSL” values include some of the time taken to set up the function call, but not the actual time spent inside the call. The “Native” row is the time spent in executing native (built-in) functions.

The first percent column, “%Proc,” is the percent of the total CPU time for the process. The second column, “%Total,” is the percent of the total CPU time for the Agent.

The total agent CPU time is available at the top of the report. It shows the elapsed (wall clock) time of the Agent and the total CPU time for the Agent. The percent CPU load caused by the Agent (CPU/elapsed) is shown in parentheses.

22.3 Timing PSL Code Blocks

Once the profiler identifies the offending script, the next step is to measure that script in detail. The PSL profiler does not record per-line or per-statement resource usage. Other methods must be used. The use of the PSL functions that measure time is a brute-force solution to the problem of measuring the cost of blocks of PSL code within a single PSL script. The main primitive is the PSL `time` function, which reports the current time in seconds. Unfortunately, this method works only for very cost-intensive PSL scripts that run for many seconds. Measurement of a block of code is possible via simple means:

```
start = time();
# ... Block of code being measured...
end = time();
printf("Time taken = %f seconds.\n", end - start);
```

A slightly more accurate method is to ensure that the code is not swapped out by the PSL interpreter’s timeslicing method. This can be achieved via playing with the PSL `timeslice` special variable, or via the trick that blocking functions will force a swap-out, such as PSL `sleep` with a 1 argument (for a one-second delay; unfortunately, a 0 delay does not give this effect). Because this forces a swap-out, a full timeslice will occur when it is swapped back in on return from that blocking built-in function. Therefore, putting a call to a blocking function before the measured code will allow a full timeslice. However, these tricks are not really relevant, because of the coarse granularity of this method involving PSL `time`, which means that many timeslices will occur, anyway.

Unfortunately, this measurement method has a few limitations. Firstly, it has a coarse granularity of whole seconds, which makes this method unusable for all but the longer running PSL scripts. Secondly, it includes time other than the PSL execution, including time used by other PSL scripts that are running at the same time. Thirdly, it includes any delays in script execution from blocking functions such as from PSL `sleep`, `system`, `lock`, and so on. All these limitations ensure that this method is rarely applicable. Unfortunately, no PSL function, even in PATROL 3.2, has lower granularity than the PSL `time` function.

22.3.1 Fine-Grain PSL Profiling

The PSL functions that control the PSL profiler can be used for fine-grain profiling information about a script's execution. This requires a PATROL 3.2 agent to get the profiling support. The method to achieve this is to enable the PSL profiler for the current script, and then sample the profiling reports at key points in the script to determine where the PSL script is using its resources. Using the user-defined PSL functions `start_timing`, `timing_report`, and `stop_timing`, as defined in a previous example, to control the PSL profiler, we can do something like:

```
# turn on PSL profiler using ProfOptions and getpid
start_timing();

# save an initial PSL profiler sample using ProfGet
report_before = timing_report();

# .... Block of code being measured...

# get second PSL profiler sample using ProfGet
report_after = timing_report();

# Perform analysis & arithmetic on both profiler samples

# turn off profiling with ProfOptions
stop_timing();
```

22.3.2 PSL Quad-Code Analysis

The PSL profiler does not provide much information to identify the lines of code in a single PSL script that are a performance bottleneck. The standalone PSL compiler available as the “`psl`” executable can be used to get some information about “static” tuning issues, such as the number of quad-codes that are being executed. This is simply the “`-q`” option as in “`psl -q myfile.psl`” and a report goes to standard output. This shows the total number of internal quad-codes, and some summary statistics. These summaries are not particularly useful for performance tuning, because they indicate static percentages without any weighting on how often they are executed.

The quad-code listing at the front of the report is more useful because it shows how many quads the agent must go through, a number that is roughly proportional to the CPU used by the PSL script (although built-in function calls like `grep` take more than other steps, so it's a rough measure). The most useful thing is that analysis of the quad list can show you where your “foreach” and “while” loops appear, and give you some ideas on how many quad-codes are being repeated in each loop. You can also see useful things like the fact that “`print (a,b) ;`” using the comma has fewer quad-codes than “`print (a.b) ;`” using the string concatenation operator.

The PSL optimizer also has options that produce a lot of useful information in reports about optimization. The “-OP2” option to ‘psl’ can show useful information, and will actually show you that the above-mentioned comma-versus-dot-operator optimization is actually automatically found by the optimizer. The “-OP3” option is also available, although this is more of an optimizer debugging setting.

22.3.3 PSL Operation Benchmarks

The use of “`psl -q`” to show the number of quad-codes gives a good idea of the performance impact of each individual operation. Generally speaking, each quad-code is about the same execution time within the interpreter, with the exception of calls to PSL built-in functions. However, not all PSL operations involve just one quad-code. Table 22-2 shows the number of quad-codes for each operation, excluding the cost of computing their operands, which of course may involve other operations.

Table 22-2 PSL operation benchmarks

PSL Operation	Number of PSL Quad-Codes (Non-Optimized)
Numeric constants (e.g., 1, 3.14)	0
String constants (e.g., “abc”, “1\n2\n3”)	0
Named constants (e.g., ALARM, EOF, TRUE, etc.)	0
List [. . .] operator (e.g., [1, 2, 3])	1 + 1 per every 3 list arguments
Mathematical operators (binary) +, -, %, *, /	1
Mathematical operators (unary) +, -	1
Logical operators (&&, , !)	1

Table 22-2 (cont'd) PSL operation benchmarks

PSL Operation	Number of PSL Quad-Codes (Non-Optimized)
Relational operators (<, >, <=, >=, ==, !=)	1
String pattern relational operators (=~, !~)	1
Shift operators (<<, >>)	1
Bitwise operators (&, , ^)	1
String concatenation “x.y” operator	1
Assignment operator (=)	1
Extended assignment operators (+=, -=, etc)	1
Increment (++); decrement (--)	1
Ternary operator (? :) (PATROL 3.2)	1
Compound statements (i.e., { ... } blocks)	0
if statement	1
if-else statement	1
switch statement	1
User-defined function call (entry to function)	2 + 1 per 3 parameters
function statement	0
return statement (in user-defined function)	1
exit statement	1

Table 22-2 (cont'd) PSL operation benchmarks

PSL Operation	Number of PSL Quad-Codes (Non-Optimized)
foreach loop statement	2
while loop statement	2
next statement	1
last statement	1
export statement	0 (compilation cost only; no runtime issues)
requires statement	0 (compilation cost only; no runtime issues)
for loop (PATROL 3.2)	2
PSL built-in function call	2 + 1 per 3 arguments + execution cost

The performance characteristics of the various PSL built-in functions is not as simple as most other PSL language features. They have the extra cost factor that is the actual execution cost of performing the function operation itself. This is not as constant for built-in functions as it is for other quads. A number of the PSL built-in functions can potentially be very expensive, but it depends on the data sent into the function. For example, PSL `grep` is fast on a short string and potentially costly when applied to a much longer string. Table 22-3 aims to offer some rough guidelines by listing some of the built-in functions that can become performance burdens if used too frequently or on large data sets.

Table 22-3 PSL built-in function costs

PSL Built-in Function	Critical Cost Factor
<code>grep</code>	Regular expression complexity; number of lines.
<code>cat</code>	Size of the file.

Table 22-3 (cont'd) PSL built-in function costs

PSL Built-in Function	Critical Cost Factor
system	Character length of the output; time taken to launch a child process; time taken by child process.
popen	Low cost to the agent because of immediate return but overall effect of time taken by child process.
execute	Character length of the output; time taken to launch a child process; time taken by child process.
set	Internal symbol table accesses take longer than simple string operations (e.g., <code>index</code> , <code>substr</code>).
get	Internal symbol table accesses take longer than simple string operations (e.g., <code>index</code> , <code>substr</code>).
ntharg, nthargf	Number of lines in the input string (columns are not so much an impact on performance).
nthline, nthlinef	Number of lines in the input string.

PSL functions should not be considered only in terms of the internal agent cost. The external access PSL functions also contribute to the overall system cost. The PSL `system`, `execute`, and `popen` functions have a process spawning cost and also increase the number of system processes. The PSL `file`, `read`, and `write` functions increase the amount of file and I/O operation requests that are sent to the operating system.

22.4 PSL Optimizer

The PSL Optimizer in PATROL 3.2 provides a transparent optimization step. It aims to take the generated quad-code from the PSL compiler and optimize the quad code where possible. The PSL Quad Optimizer is a subsystem that accepts PSL quads as input, optimizes the quads, and produces PSL quads as output. Below are some examples of what the PSL Optimizer can do to improve the efficiency of the executed quad-code:

- Collapses jump chains
- Removes redundant and unreachable code

- Removes redundant assignments
- Collapses “.” concatenation operator chains—e.g., `a.b.c.d` or `[a,b,c,d]` turns into `join("",a,b,c,d);`
- Rearranges all looping constructs so that they are more efficient
- Parameter packing so that less overhead is associated with function calls
- Folds and propagates expression values

The PSL quad optimizer performs two basic functions:

- Reduces the number of quads to execute
- Reduces the number of intermediate values used in evaluation

The changes performed by the PSL optimizer can be examined via the “-q” option to the “psl” executable. By using `psl` with and without a “-O” option, you can see what changes are occurring. Tests using the optimizer show that on average the PSL scripts were 15-20 percent more efficient in terms of number of quads executed. However, this fact may not be that noticeable unless you have a heavily loaded machine with many scripts. In this case, you will see an improved throughput of PSL scripts. The optimizer “-OP2” and “-OP3” options to `psl` can also be used to generate reports about optimization results.

When building PSL binaries or libraries, you can choose the PSL optimization level via the “-O” option to the “psl” command. The “-O0” option disables optimization and “-O3” chooses the maximum level. Clearly, in all cases of pre-compilation, the maximum optimization level is worth pursuing. In addition, the optimizer will also actually reduce the total number of quad-codes, and thereby reduce internal agent memory usage. The quad-codes will be different after optimization, but this really affects only the PSL runtime tracing output, which may become a little harder to decipher. This issue is certainly secondary to runtime performance improvement.

For the situation of KMs where PSL source files are used, the optimizer is more of a configuration issue than one for KM development. Since the agent compiles all PSL source before execution and will apply optimization if configured to do so, the benefit depends on what optimization level the agent is configured to have, rather than on anything done by the KM developer to the PSL source.

The “pragma” statement can be used to override agent or “psl” optimizer settings. Since optimization is a post-compilation step, the settings are not used until after the first phase of compilation, and the last “pragma” statement found in a PSL script will be the optimization setting that is applied. Therefore, where in the script a pragma statement is added does not matter. The pragma statement is purely a compilation statement and has no runtime impact. The syntax for the pragma statements would be:

```
# Turn on maximum optimization for this PSL script
pragma "O3"
```

```
# Turn off optimization for this PSL script
pragma "O0"
```

The `pragma` statement can have either “O3” or “-O3.” The dash is optional. The “P” option is also available and allows printing of optimization information. The option could even be “-OP3” to get level 3 optimization with printing.

This `pragma` statement allows the KM developer to have control over optimization even in a pure PSL source KM. Style decides whether to mandate maximum optimization in the scripts, or to leave this to the agent configuration settings for the PATROL administrator to set appropriately.

22.5 PSL Efficiency Tips

PSL is like any procedural language, and the majority of traditional performance-tuning techniques apply. Some of the general performance-tuning techniques include:

- Function call reducing—using straight line code instead of user-defined functions.
- Loop optimizations—reducing the work done each loop iteration.

As with any programming language, loops can be costly, and PSL `foreach` and `while` loops are likely causes of inefficiency. Try to move unchanging computations outside the loops. Try to avoid using nested loops because these are even more of a bottleneck. However, be careful manually optimizing these, because they are risky changes to make, and the PSL optimization level “-O3” may do a lot of the simpler loop optimizations automatically. Some of the general loop optimization steps you can do manually include:

- Code “hoisting”—moving code out of loops if it need not be recomputed each time.
- Loop unrolling—using straight line code for short loops instead of looping conditions.

The PSL built-in functions are efficient relative to single PSL operations. By knowing the large list of PSL built-in functions available standard with the language, you can often achieve quite startling performance results. Some of the functions you should become very familiar with include:

- `ntharg`, `nthargf`, `nthline`, `nthlinef`—string manipulations
- `grep`—including the “v” option to remove, and “c” option to count
- list and set operation functions—`union`, `difference`, `subset`, `sort`
- `sprintf`—the variety of C-like `printf` formats is very powerful for string building

22.5.1 PSL List Functions

The PSL set and list functions are a particularly good area of optimizations. For example, how can you remove a line of text from a long results string? What if that line of text is not known in

advance, only that it matches a pattern, such as removing all the header lines from a query? One quick way is using PSL `grep` with the “v” option:

```
text = grep(pattern, text, "v"); # remove lines matching pattern
```

Another potential candidate is the PSL “difference” function. Equivalent to the `grep` command would be:

```
lines = grep(pattern, text);  
text = difference(text, the_lines);
```

Although the use of “v” with `grep` is faster than `difference` in this case, in many cases, the PSL set functions such as `difference`, `union`, and `sort` can be very efficient for processing long multi-line strings.

22.5.2 PSL index instead of PSL grep

The PSL `index` function is significantly faster than PSL `grep` under some circumstances because `grep` uses regular expressions and `index` uses string matching. PSL `grep` has been internally optimized since 3.1 to detect the case when no special regular expression characters exist and to perform normal string matching if so. However, although this does avoid regular expressions for plain text patterns, PSL `grep` is still slower because it does the text searching for every line, whereas PSL `index` does it once only. Therefore, use PSL `index` to test for the presence of a pattern rather than PSL `grep`:

```
if (grep("ERROR", lines_of_output)) { .... }  
if (index(lines_of_output, "ERROR")) { .... } # FASTER!
```

Unfortunately, you cannot generally use PSL `index` instead of PSL `grep` in non-conditional cases. PSL `grep` returns a list of lines, whereas `index` returns an integer. So you cannot do the following because the return value differs:

```
ret = grep("fred", text);  
ret = index(text, "fred"); # WRONG! Faster but not the same!
```

Also, you cannot make the swap from PSL `grep` to PSL `index` unless you are sure that the pattern is a plain text string pattern and not a complex regular expression. Typically, these two restrictions mean that you can optimize calls to PSL `grep` when they are in an “if” statement test, and the pattern is a simple string literal in double quotes that does not contain a fancy regular expression. And, of course, when optimizing PSL `grep` to use `index`, be sure to remember to reverse the arguments.

22.5.3 PSL grep Regular Expression Optimizations

Using some knowledge of the underlying implementation of PSL `grep`, here are some ways to improve performance. Regular expressions are expensive and the PSL `grep` function is expensive, especially on multi-line strings, because it does a regular expression match against each line. The PSL `grep` call is expensive only if special characters are used, including “`*+[]^$`”, etc. Hence, `PSL grep("pattern", text)` is a fast plain-text match without regular expressions, but `PSL grep("^pattern*", text)` is several hundred times slower. The cost of the regular expression match is directly proportional to the number of lines in the text being searched. Using this knowledge, you have some surprising optimizations of PSL `grep` usage. An example is where “text” contains multiple lines, an optimization is to change

```
ret = grep("fred.*barney", text);
```

into the lengthier, but actually more efficient code:

```
# Faster if lines with "fred" are few
ret = grep("fred.*barney", grep("fred", text));
```

The multi-line text string will probably have only a few lines containing “fred,” and so we do a fast non-regular expression plain text `grep` first to reduce the number of lines that will need a slower regular expression match applied. Note that you can’t just search for “barney” because then you’d get the wrong matches for “barney...fred” as well. Numerous similar improvements use this idea.

```
PID = 1234; PPID = 4321; # PID & PPID got from somewhere
ret = grep( PID . "[ ]*" . PPID, text) ;
```

A more efficient version could be:

```
ret = grep(PID . "[ ]*" . PPID, grep(PID, text));
```

Note: These optimizations matter only if the searched text has many lines. For one line or even a few they’re not really worthwhile.

22.5.4 Expression Short-Circuiting via Nested If Statements

Although the conditional test expression “`if (e1 && e2) { ..code.. }`” will short-circuit efficiently in C, it does not short-circuit in PSL. So the `&&` and `||` operators are potentially more expensive than the KM developer expects. If the expression `e2` is expensive to compute, the only PSL short-circuiting method is to use nested `if` statements:

```
if (e1) { if (e2) { ..code.. } }
```

This wastes some space in the agent with a few extra quad-codes but will be faster if “e1” can be false often and is a quick test. However, this optimization is likely to be minor unless e2 really is an expensive test, so don’t bother unless e2 is complicated, such as involving a function call.

22.5.5 Character Conversions

Functions that aim to convert a single character type to another character are often inefficiently implemented. For example, to convert the string “/usr/local/bin” to “usr-local-bin” is one such need, and could be implemented by a complex `foreach` loop combined with extracting the arguments one at a time with `ntharg` and then rebuilding them via the string concatenation operator. For example, the slow code might be something like:

```
str = "/usr/local/bin";
i = 1;
ret = "";
part = ntharg(str, 1, "/"); # get first word before "/" separator
while (part != "") {
    if (ret != "") { ret = ret . "-"; } # add the separator
    ret = ret . part;
    i++;
    part = ntharg(str, i, "/");
}
```

This code is ugly and inefficient. However, if you know the built-in `ntharg` function better, this can be replaced by a call to `ntharg`, which utilizes the 4th argument, the separator for the result returned.

```
str = "/usr/local/bin";
ret = ntharg(str, "1-", "/", "-");
```

This one-liner replaces a complex PSL loop that does dozens of quad-codes with a handful of quad-codes from a single call.

22.5.6 String Concatenation Optimizations

A number of minor optimizations apply to string concatenation operations in PSL. However, the PSL optimizer now automatically performs these optimizations. For PATROL 3.2 and beyond, you want to avoid sacrificing readability for these minor optimizations, because the optimizer will handle them.

The PSL string concatenation operator is one area that can be optimized. However, the PSL optimizer now does some of these optimizations automatically. Long concatenations are not that efficient, and for example:

```
some_var = a.b.c.d.e.f.g;
```

should be replaced for efficiency by a call to `sprintf`:

```
some_var = sprintf("%s%s%s%s%s%s", a,b,c,d,e,f,g);
```

Note: one exception to this optimization is if `a,b,c,d,e,f,g` are all quoted string constants when the concatenation is done at compile time once. Another exception is that the PSL optimizer does this optimization automatically in 3.2, making both statements have the same effect.

A related optimization involves common use of the PSL `print` function with string concatenation. In this case, the “.” operator is not particularly slow. However, one common inefficiency is:

```
print(x . y);    # string concatenation (slow)
```

A more efficient version is to use the comma syntax:

```
print(x,y);    # commas (faster)
```

The use of commas is marginally faster if you examine the quad-codes that are generated. The comma in a function call is marginally better than using the string concatenation operator.

22.5.7 Group Pattern Checks with `grep`

A common need is to check the output of a command or the contents of a file for more than one pattern. Do not check multiple pattern values this way:

```
if (index(ret, "Permission denied") ||
    index(ret, "Command not found") ||
    index(ret, "Access denied") ||
    index(ret, "GO AWAY!!") {
    .....
}
```

Instead, if the output variable ‘`ret`’ is guaranteed to exactly match one of these strings, use this code with PSL `subset` instead:

```
bad_values = "Permission denied\nCommand not found\nAccess
denied\nGO AWAY!!";
if (subset(bad_values, ret) {
    .....
}
```

A more general alternative that allows more flexibility in the searched string is to use the PSL `grep` function with multiple patterns via its “or” query options specified by the “|” sign.

Note that the “|” character must be escaped to PSL `grep` to get this special “or” meaning. The method to `grep` a multi-line string for multiple string patterns is:

```
bad_values = "Permission denied\\|Command not found\\|Access
denied\\|GO AWAY!!";
if (grep(bad_values, ret) {
    .....
}
```

Several calls to `index` have been replaced by one call to `subset` or `grep`, and little extra cost is involved if more strings are targeted for search.

22.5.8 Loop Iteration Reduction

A typical PSL action is to loop through a list of lines of text and perform some action on them. Another typical aspect of this problem is that you want to skip some lines. The simple and inefficient implementation of this idea is:

```
foreach ln (text) {
    if (index(ln, bad_pattern) {
        next; # skip this line
    }
    do_something(ln);
}
```

This is inefficient because it still does the entire loop iteration for lines to be ignored. A much more efficient solution is to remove the unwanted lines before the loop:

```
# remove unwanted lines before loop
good_text = grep(bad_pattern, text, "v");
foreach ln (good_text) {
    do_something(ln);
}
```

Notice also that the argument orders for PSL `index` and `grep` are the reverse. This order is a common pitfall in PSL coding. Optimizing code can easily introduce new PSL coding errors.

22.5.9 Common Subexpression Elimination

Another common issue is code that does a PSL `set` operation on many variables for a single object. For example, it might set some hidden data values, or it might be a collector setting all the values for a number of consumer parameters for a particular instance. Inefficient code might well look like the following, with lots of string concatenation operations:


```

.....
set("/APPL/" .inst. "/something", 1);
set("/APPL/" .inst. "/something_else", 1);
set("/APPL/" .inst. "/something_else_again", 1);
set("/APPL/" .inst. "/something_different", 1);
set("/APPL/" .inst. "/something_strange", 1);
set("/APPL/" .inst. "/something_bizarre", 1);
.....

```

Far more efficient is to use this code instead:

```

.....
prefix = "/APPL/" .inst;  # Common subexpression!
set(prefix."/something", 1);
set(prefix."/something_else", 1);
set(prefix."/something_else_again", 1);
set(prefix."/something_different", 1);
set(prefix."/something_strange", 1);
set(prefix."/something_bizarre", 1);
.....

```

The cost of computing the “prefix” is done once for all the `set` calls instead of once per `set` operation. A common subexpression has been eliminated from each time and done only once.

22.5.10 Pre-Computing Loop Invariants

Consider the following dummy code:

```

fred = "fred";
derf = "derf";
foreach inst (instances) {
    ..... # this code does not modify fred or derf
    some_func(fred.derf,1);
    some_other_func(fred.derf);
    yet_another_func(fred.derf);
    ..... # this code does not modify fred or derf
}

```

The expression “`fred.derf`” is repeated and leads to many string concatenation operations in each loop iteration. This code should be changed to precompute the subexpression:

```

fred = "fred";
derf = "derf";
fredderf = fred.derf; # Pre-computing the value!

```

```

foreach inst (instances) {
    ..... # this code does not modify fred or derf
    some_func(fredderf,1);
    some_other_func(fredderf);
    yet_another_func(fredderf);
    ..... # this code does not modify fred or derf
}

```

The cost of calculating “fredderf” is done once at the start of the code instead of once per iteration of the loop.

22.5.11 String Formatting in PSL

The `sprintf` function is a more efficient way to format strings than trying to do so manually. Brute-force string formatting without `sprintf` often involves the use of `substr` and a `foreach` loop. The `sprintf` function can often do the entire result in a single call because of its powerful C-like formatting capabilities. Below are two examples of code optimizations involving proper use of `sprintf`’s full capabilities.

First, consider the value that the response function spinner widget returns. Here are functions that will format the output, with the inefficient version first:

```

function convert_seconds_to_time(seconds)
{
    local num_hours;
    local num_minutes;
    local num_seconds;

    num_hours = ntharg((seconds / 3600), 1, ".");
    if (num_hours == "0")
    {
        num_hours = "00";
    }

    seconds = seconds - (num_hours * 3600);
    num_minutes = ntharg((seconds / 60), 1, ".");

    if (num_minutes == "0")
    {
        num_minutes = "00";
    }

    seconds = seconds - (num_minutes * 60);
    num_seconds = ntharg(seconds, 1, ".");
    if (num_seconds == "0")
    {
        num_seconds = "00";
    }
}

```

```

    }

    return num_hours ":" . num_minutes ":" . num_seconds;
}

```

This is clearly an expensive PSL function involving many PSL operations. And now here is a much more efficient version of the same user-defined function using the PSL built-in `sprintf` function.

```

function convert_seconds_to_time(seconds)
{
    local num_hours;
    local num_minutes;
    local num_seconds;

    num_hours    = int(seconds / 3600);
    seconds = seconds - (num_hours * 3600);
    num_minutes = int(seconds / 60);
    seconds = seconds - (num_minutes * 60);
    num_seconds = int(seconds);

    return sprintf("%02d:%02d:%02d", num_hours, num_minutes,
                  num_seconds);
}

```

22.5.12 Remove Loop Function Call Overheads

Removing function call overhead from loops is often possible. For example, the following code does quite a lot of work inside the loop body. This code aims to take a list of string items and reformat them left-justified with a 50-character field width on each line.

```

fmtList = "";
foreach item (lst)
{
    fmtField = sprintf("%-50s\n", trim(item, " \t\n"));
    fmtList = fmtList . fmtField;
}

```

Too many calls to the “trim” built-in function are inside the loop. The same effect can be achieved more efficiently by doing the “trim” operation prior to the loop:

```

fmtList = "";
lst = trim(lst, " \t\n"); # trim all unwanted characters now
foreach item (lst)
{
    fmtField = sprintf("%-50s\n", item);
}

```

```
        fmtList = fmtList.fmtField;  
    }  
    fmtList = nthline(fmtList,"1-"); # trim multiple "\n\n..." lines
```

In the improved version, the cost of calling the “trim” function has been removed from the `foreach` loop. Actually, if the output of “`psl -q`” is examined for each code segment, it will show that this reduces the number of quads inside the `foreach` loop by 50%!

KM Debugging Techniques

Debugging and testing a Knowledge Module during the development process is conceptually not very different from any other debugging task. However, several PATROL-specific tools and techniques can be used. This chapter therefore examines general debugging approaches and PATROL debugging tools, such as the PSL Debugger.

23.1 Applying General Debugging Techniques

Debugging is something that applies to all languages, and most of the general techniques apply just as well to PSL as to any other programming language. Some of the well-known programming techniques for testing code include:

- Unit testing—writing code to test individual functions or elements of the code.
- Inspections or walkthroughs—manual, human analysis of source code.
- Self-testing code—adding code specifically for testing into the mainstream code.
- Assertions—a particular form of self-testing code with emphasis on condition testing.
- Defensive coding style—writing the program to gracefully handle unexpected conditions.

All these ideas can be expressed in PSL just as well as in any other programming language. The next sections discuss each of these areas.

23.1.1 Unit Testing

The simplest form of regression testing is testing the individual unit parts of a program. In KM development, the “units” that you could test could be individual PSL files, PSL libraries, or PSL user-defined functions.

Some PSL library functions can be unit tested by the PSL standalone interpreter. By applying the executable “psl” to a test file that exercises the library functions, you can unit test a lot of functionality. However, the fact that the PSL interpreter does not deal with all agent-specific built-in functions does limit the scope of this method. For example, it cannot launch child processes.

Another method is to write a separate testing KM to perform unit testing of functions in PSL libraries. This KM can use the PSL libraries and exercise them in the full agent environment. Clearly, it cannot mimic the results of functions that access the external environment (i.e., files, processes, etc.), but it can mimic the internal agent environment such as the symbol table, and a lot of other functionality can be tested.

23.1.2 Code Inspections and Walkthroughs

Manual human inspection of code is a surprisingly effective debugging tool. Taking your source code home to read can usually help bugs and solve insomnia at the same time. More formal procedures are possible involving group inspections of code. Other possibilities are formalized walkthroughs, where the developer has to present and explain the code by walking through its execution. None of these techniques are new to developers and apply just as well to PSL as other languages.

23.1.3 Self-Testing Code and Assertions

The general purpose of self-testing code is to add code that performs internal checks for odd conditions. For example, a PSL user-defined function could validate its own arguments. Many other forms of self-testing code are possible. For example, your PSL code could contain sophisticated internal unit tests, provided that they are hidden by a special test that will not occur in production:

```
if (get("/doInternalSelfTests")) {  
    ret = test_my_function(1);  
    if (ret != 0) { print("Internal unit test failed\n"); }  
}
```

The performance cost to production code is minimal. Unfortunately, no preprocessor exists in PSL with which to compile out these tests. Manual commenting out is, of course, an option prior to completion.

Assertions are a specialized form of self-testing code. No specialized support for assertions is in the PSL code or in any standard PSL libraries. However, you should not have much difficulty adding a user-defined PSL function called “assert” and having it act the way assertions are expected to. For example, the code could be:

```
function assert(condition, message)  
{
```

```
        if (! condition) {
            print("ASSERT FAILED: ", message, "\n");
        }
    }
    ....
    assert(x > 0, "x out of bounds");    # use assert functionality
```

Unfortunately, PSL cannot do some of the things commonly handled by C language assertions, such as “stringizing” the condition, showing the line number in the script, and filename. The origin of the assertion could be automatically detected by some clever use of PSL `get on “name”` and some knowledge of the different contexts of execution in the symbol table.

23.1.4 Defensive Programming Techniques

Knowledge Module quality depends a lot on the coding style of the developer. In particular, does the code handle the unusual cases involving external failures, or even internal bugs in the code?

Return code checking is the simplest of defensive coding techniques. The code should handle all the situations even if they are not expected. The clearest need for return code checking occurs in external functions that could fail due to something beyond the control of the program, such as a channel failure, a dying process, a missing or corrupt file, and so on. For this reason, some of the PSL built-in functions that clearly should have return codes checked include:

- `system`, `execute`, `popen`
- `read`, `write`, `fopen`, `fseek`, `ftell`
- `process`, `proc_exists`
- `file`, `cat`

Checking for failure conditions has a few aspects. The clearest is the actual return code. Another is that some functions set “`errno`” to indicate failure. Finally, PSL `system` and `execute` will set the “`exit_status`” PSL variable with the child process’s execution termination status.

An even more defensive style of coding is to check all the return codes in PSL. This includes not only other built-in functions, but also the user-defined PSL function results. This safety net can find not only exceptional conditions, but also internal hidden coding bugs. However, it is a lot of work, and a balance needs to be set according to the project influences.

23.2 Debugging Output

The simplest and most common debugging technique is to print out messages during PSL execution. By printing out values of variables and also messages at points in the script, you can debug problems with data and flow of control.

Adding debugging printouts can be as basic a technique as adding “print” or “printf” PSL statements to code during debugging, and then removing them after debugging. The next level involves leaving the statements in the code, but hidden behind a test on some debug variable setting. More advanced systems allow lower granular control within the test for debug output control. The granularity can be over classes of PSL (e.g., discovery, menus, parameters), and for even more advanced systems, based on the actual name of an individual menu command or parameter.

Debugging printouts are one part of a KM that makes many people happy. They are not only useful for the KM Developer during testing and debugging. Supporting the KM through phone support is also much easier if debug output tracing can be dynamically enabled via a menu command.

23.2.1 Simple Debugging Output Methods

The simplest method of controlling debugging is to hard-code tests into the code. For example:

```
DEBUG = 1;    # 0 to disable, 1 to enable
....
if (DEBUG) { print("Starting discovery script\n"); }
....
```

This also requires manual code editing to remove the prints, but only to one character. Unfortunately, this method leads to dead code in the final KM, and no PSL preprocessor exists with which to use `#ifdef` statements or the like, which are well known to C. Preprocessing is not supported by PSL. Admittedly, these statements are not totally “dead,” in that the user of the KM can edit the PSL to turn on debugging, which is somewhat useful for supporting the KM.

The better approach is not to have to edit PSL code to turn on debugging code. For this reason, the most sophisticated debugging code relies on tests based on PSL `get` from the agent symbol table. The code might look like:

```
DEBUG = get("/MYKM/Debug");
....
if (DEBUG) { print("Starting discovery script\n"); }
....
```

The variable “Debug” is not a special built-in agent macro. Instead, it is a macro that is purely KM-specific, and the KM developer has full control over it. The name does not have to be “Debug,” and in practice many KMs use their own name. Generally speaking, the KM will have a menu command, or set of menus, that uses PSL `set` to turn debugging on or off by modifying this variable. The menu command “Turn On Debugging” could be just one line of PSL:

```
set("/MYKM/Debug", 1);
```

Naturally, the equivalent “Turn Off Debugging” would be:

```
set ("/MYKM/Debug", 0);
```

23.2.2 Selective Debugging Output

Having only one global debug variable for controlling many different PSL scripts quickly becomes unusable. Since PSL scripts have interleaved execution within the agent, all the debug output messages from the different PSL scripts become mixed in the output.

One global debug setting for menu commands is usually adequate, because the user typically has only one menu command running at a time. Similarly, pre-discovery and discovery need only one global setting, because only one such script exists per KM. However, parameter debugging is often still annoying if only a global level setting exists. Many parameters are running at the same time, and they can interleave output. Multiple copies of the same parameter are running if more than one application instance exists.

23.2.3 PSL Library Debug Output

PSL libraries propose a unique difficulty in adding debug output to a KM. The problem is basically that you have no easy way to set a global variable within the library space. For example, the following code method will not work:

```
# PSL main file..
requires mylib;
....
DEBUG = get ("/MYKM/Debug");
....
lib_func(); # Call library function
.... # rest of main file

The library file as follows:

# PSL library file...
export function lib_func;

function lib_func() {
    # THIS DOES NOT WORK!!
    if (DEBUG) { print("Entering library function\n"); }
}
```

This code will never print the message from the PSL library. The problem with this attempt is that within the PSL library, the variable “DEBUG” is a global variable that is local to the PSL library space. The reference to “DEBUG” does not find the variable from the caller data space. In PSL programs, a PSL library cannot reference a global variable in the space of the caller function.

You have no easy way to fix this problem. The main file that uses a PSL library function can introduce values into the PSL library's data space only when the library explicitly exports a variable. This idea gives us the next more successful attempt:

```
# PSL main file..
requires mylib;
....
DEBUG = get("/MYKM/Debug");
LIB_DEBUG = DEBUG; # set library variable too
....
lib_func(); # Call library function
.... # rest of main file
```

The library file is changed to contain an additional `export` statement as follows:

```
# PSL library file...
export function lib_func;
export LIB_DEBUG;

function lib_func() {
    if (LIB_DEBUG) { print("Entering library function\n"); }
}
```

This method does actually work and the message can be printed. In addition, this method will print messages only when this particular PSL process calls the library functions, because of the way library exported variables work. Each library exported global variable has a value on a per-PSL process basis, rather than sharing one value for all users of the library.

The only downside to this method is that you need one debugging variable per library used. Hence you need a "LIB_DEBUG1" and "LIB_DEBUG2," and so on, according to whatever naming scheme you choose for these variables. Each PSL script needs to set each and every one of these library exported variables for all the libraries that are used.

23.3 PSL Debugger Usage

The PSL Debugger is the main debugging tool provided on the Developer Console. It is accessible via the "Developer" menu on computer icons as the "PSL Debugger" menu command. This launches an interactive GUI window with many features for debugging PSL scripts.

You have two basic methods for using the debugger. The first is to load a PSL source file, compile it within the debugger, and then run it. The second is to attach to an already compiled and running PSL process that is currently on the remote agent.

The PSL Debugger has a number of powerful features for debugging each script. Unfortunately, the possibility of getting to this point in the debugging cycle is hampered by some serious usability issues of the debugger. However, once familiar with the debugger, features such as single-stepping and tracing can be very useful for diagnosing PSL coding problems.

23.3.1 Debugging a Parameter

You can load the PSL source text for a parameter, compile it in the debugger, and run it. However, almost invariably, this approach causes a problem because the debugged script does not run in the parameter context. The results are not usually satisfactory.

Parameters are persistent commands that always have a PSL process in the agent. The best way to debug a parameter is to use the “attach” command in the PSL Debugger. This will attach to an already-running PSL process and allow a debugging session to begin. Potentially, you could even attach to a script in the middle, but unless your parameter happens to be running at the time you choose the command, you will just start at the beginning of the script.

An “attach” command is in the PSL Debugger command set. The basic procedure for attaching to a parameter’s PSL process is as follows:

1. Click the %PSLPS button to see the list of PSL processes.
2. Find the PID of your parameter’s PSL process in this list.
3. Type the command “attach <PID>” in the command entry small window, below the output window.
4. The debugger will then attach to the agent’s PSL process, and report success.
5. Load the PSL source code via the “Load” button, manually finding the correct “.psl” file in the file tree for the “psl” directory.
6. Now you can perform debugging with an interactive session with a process and PSL source text.
7. Detach from the PSL process via the “detach” command, or by exiting the PSL Debugger window.

This procedure shows how to find and attach to a PSL parameter. You complete a lot of manual steps without automatic control to get into the debugging session. However, to use advanced features like stepping is usually worth the effort.

The only additional point about debugging parameters is in the rare situation that you need to attach to the very first execution of the parameter, at the very start of the script. In this case, using the PSL Debugger and “attach” does not guarantee that the parameter has already run by the time you get to attaching to the parameter. In this case, you need to use the PSL “debugger” built-in function, as discussed on page 337.

23.3.2 Debugging a Discovery Script or Pre-Discovery Script

Discovery and pre-discovery scripts have identical scheduling profiles to parameters. They are permanent, repeating PSL processes that always have a process in the agent. Hence, the basic method of debugging a discovery script is usually via the “attach” command, with the same set of basic steps as covered for parameter debugging. You can also directly load, compile, and run a discovery script, but again the context of the script is not the same, leading to different results.

Catching the very first execution of discovery is similar to parameters in the need for using the PSL “debugger” function. Pre-discovery almost certainly requires the “debugger” built-in function, because pre-discovery will often only run once, and it runs immediately after the KM is loaded. However, unsuccessful pre-discovery can loop repeatedly if it does not set the “active” variable to 0 or 2, in which case, attaching can be a debugging method.

23.3.3 Debugging Menu Commands

The PSL Debugger is useful for debugging the KM menu commands. A menu command is a console launched command that has no initial presence on the agent. Therefore, you can debug a menu command from the PSL debugger by performing a load of the PSL source, compiling it, and running it. The debugger will send the PSL script to the agent just like the console would normally do when the user chooses the menu command.

Unfortunately, one problem occurs with debugging a menu command in this way, regarding the symbol table execution context. All debugger scripts run at the top-level computer “/” context, which affects PSL `get`, `set`, and other symbol table operations. However, a menu command when executed against an icon will run in that instance’s context. Therefore, differences in the menu command will occur versus running in the debugger relating to symbol table operations (PSL `get`, `set`, `exists`, if their path is relative) and inheritance of attributes (e.g., environment variables). Depending on the situation, these differences may or may not be important. For example, if all PSL `get` and `set` operations are coded to use absolutely specified paths for the variable starting at the top-level / character (i.e., “/appl/inst/param/whatever”) rather than using relative paths (i.e., “. ./param/whatever” or just “whatever”), then the symbol table context may not matter. Nevertheless, this peculiarity is an annoying coding style requirement to impose just to work around the debugger limitations.

An alternative method of debugging a menu command is to use the “attach” command in the debugger. However, it is more difficult to attach to a menu command because its process does not initially exist. Attaching to a parameter or discovery script is easier because the processes already exist. Attaching to a menu command is achievable mainly via the PSL “debugger” function, which is discussed on page 337.

23.3.4 Debugging InfoBox Commands

Behind the scenes, InfoBox commands are just like menu commands. The console treats them just like menu commands in how it sends them to the agent, but the console redirects the output from these commands into the special info box entries. Therefore the debugging methods for info box items are identical to menu commands.

23.3.5 Debugging Recovery Actions

The issues involved in debugging a recovery command are almost identical to menu commands. Recovery commands have processes only when they are spawned, just like a menu command. Hence, the main methods of debugging a recovery action are (a) via load, compile, and run

within the PSL Debugger, which has the problem of the wrong context; or (b) via the PSL “debugger” function and an “attach” command within the PSL Debugger.

23.3.6 Debugging an Event Notification Script

Event notification commands can be handled in the PSL Debugger only via the “attach” command. The source for a notification command generally cannot be loaded into the debugger, compiled, and run, because the debugger does not handle the special event macros such as `%{EV_TYPE}`. This limitation leads to compilation errors when compiling the PSL source via the debugger.

This leaves “attach” as the only method for debugging a notification script. However, since notification scripts are initiated without a permanent process id, the only way to attach to a notification script is to use the PSL “debugger” function within the event notification script itself.

23.3.7 Using the PSL “Debugger” Function

The PSL language has a special function named “debugger” that can aid in the “attach” procedure for the PSL Debugger. The name “debugger” for the function is misleading, because it does not pop up the PSL Debugger GUI, but instead merely suspends the PSL process, awaiting an “attach” command from the PSL Debugger. If this never occurs, the PSL process is suspended forever, so clearly this PSL function is only for development environment usage.

The “debugger” function is used mainly to catch the very first invocation of a script, such as a parameter, menu, discovery, pre-discovery, or notification script. The PSL script suspends immediately upon execution of this built-in function. The syntax for the debugger function call is very simple:

```
debugger();    # suspend process awaiting PSL debugger attach
```

The status of the PSL function becomes suspended for debugging immediately, and this can be seen in the output of the STATUS column in the `%PSLPS` process list. The “attach” command is then used in the normal manner to begin debugging the script.

The debugger function does not suspend the process if the process is already being debugged. That makes it easy to step over in the debugger during a debugging session when you re-run the PSL process.

23.3.8 PSL Debugger Limitations

The PSL Debugger has a number of limitations that reduce its usefulness. Some of the tasks in which you might find it wanting include:

1. PSL libraries—Debugger support is weak for PSL libraries because only a single PSL source file can be displayed, which leads to problems tracing variables, line numbers, and so on.
2. No editing—The debugger has no integration with the PSL text editor, and PSL source cannot be edited within the debugger GUI.
3. Local variables—The support in the debugger for printing variables or other variable actions has no way to distinguish between contexts of global and local variables, which leads to problems when function local variables and global variables have the same names.
4. Source not found automatically—To have to manually find the PSL source file is annoying during an “attach” debugging session.

23.4 PSL Runtime Errors with PslDebug

The PSL Debugger is not the only built-in debugging support available. You also have PSL runtime error checking. The special variable “PslDebug” can be used to control the level of runtime warnings. The PSL interpreter can detect a number of common coding errors with arithmetic, operators, and built-in functions. This can be a useful first debugging step. The simplest method of enabling PSL runtime errors is to add the following line to the start of the PSL script execution:

```
PslDebug = 1;    # Enable all PSL runtime errors
```

On the other hand, when your KM goes to final stages, you might want to turn off all runtime errors. The trade-off here is that users of the KM might be confused by warning messages if they appear (i.e., those that were missed during testing), versus the fact that PSL coding bugs may be harder to trace if they occur silently. The way to ensure that runtime errors are disabled is simple code at the top of the script:

```
PslDebug = 0;    # Disable all PSL runtime errors
```

An alternative compromise is to have better control over PslDebug usage in your KM by basing it on a PSL get value. Then you can add menu commands for support usage that enable and disable the runtime warnings (and also the runtime tracing discussed on page 340). This means that all PSL scripts need code like the following at the top of the script:

```
# Get it from some KM-specific variable name
PslDebug = get("myPslDebug");
```

You can achieve finer-grain control over the different types of errors by using the bit-level values for individual error classes. By setting PslDebug to a bitwise-or of the values, you can

individually control classes. This is discussed in the PSL Reference Manual, but, in brief, the bit values also are in Table 23-1.

Table 23-1 PslDebug runtime error settings

Bit-flag for PslDebug	PSL Runtime Error Class
1	All error classes combined
2	Arithmetic involving empty string values
4	Arithmetic on non-numeric strings
8	Illegal or undefined arithmetic operations
16	Uninitialized variable usage
32	Built-in function warnings (basic)
64	Built-in function warnings (advanced)
128	Not used (reserved for future use)

The default for PATROL is 32, which enables the most basic of built-in function warnings. For example, it warns about PSL `cat` when the file does not exist, and other cases that are clearly errors. In practice, you may find that enabling all PSL warnings gives too many messages, depending on your coding style. Notably, warnings about PSL `get` on a non-existing object can be annoying because using PSL `get` is a common way to test for a variable, since it is slower to use both PSL `exists` and then PSL `get`. To enable other warnings but not the advanced built-ins class, which includes the PSL `get` warning, you can use:

```
PslDebug = 62; # bitwise-or of classes: 2 | 4 | 8 | 16 | 32
```

A good general practice is to enable all runtime errors right from the start. You can often save yourself time by having PSL find the problem automatically. You have a number of ways to achieve this aim. The very first line of every PSL script you write could be “PslDebug = 1;”. Another method is to change the agent configuration variable so that the entire agent has PslDebug set to 1 by changing the configuration setting “/AgentSetup/PslDebug” in the “config.default” file.

23.5 PSL Runtime Tracing with PslDebug

PSL runtime tracing is a standard part of the PSL subsystem, available since PATROL 3.0's initial release. It is controlled via the “PslDebug” special variable that also controls runtime error checking. By setting PslDebug to -1, you enable all runtime tracing. Alternatively, more detailed bit flags for individual trace settings are discussed in the PSL Reference Manual, but, in brief, the bit values are also in Table 23-2.

Table 23-2 PslDebug runtime tracing settings

Bit-flag for PslDebug	PSL Runtime Tracing Class
-1	Enable all tracing classes, and all runtime errors
256	Enable all tracing classes
512	Function call tracing (entry to functions)
1024	Function call argument tracing (argument values)
2048	Function return value tracing (return values)
4096	Variable assignment tracing
8192	errno tracing
16384	PSL lock tracing

PSL tracing generates a lot of data output text to the system output window. It is most commonly used to try to figure out what is really going on in a PSL script. By seeing all the variables, arguments, and returned values, you can see where things are going wrong. The simplest way to add PSL runtime tracing to a small block of PSL code in a single PSL script is:

```
PslDebug = -1;    # Turn on all tracing
... # Block of normal PSL code
PslDebug = 0;     # Turn off tracing
```

This code will yield a stream of textual output from which you hope the error will become clear. The general comments related to PslDebug in the section on runtime warnings can also relate to runtime tracing. By adding a statement like the following, you can have menu commands control the level of runtime tracing for a particular script:

```
# Get it from some KM-specific variable name
PslDebug = get("myPslDebug");
```

23.6 PSL Compilation Warnings

Another debugging tool is the warnings that occur before execution during compilation. The PSL compiler emits warnings for common situations that look suspicious but are not actually compilation errors. These warnings can be checked in the Developer Console when the PSL editor session is completed (since compilation occurs here automatically), and explicitly via the “Check Syntax” option in the PSL editor’s Options menu. The standalone compiler executable “psl” can also be used to check external PSL files for warnings and errors. Some of the compilation warnings that can appear include:

- Assignment in conditional expression (e.g., assignment in `if` statement)
- Null effect statement
- Variable used but not set
- Variable set but not used

These warnings should not be ignored and often indicate serious problems with a PSL script. Good PSL coding style should always seek to have warning-free compilation.

23.7 Acquitting the PSL Optimizer

With the addition of the PSL Optimizer in PATROL 3.2, a convenient excuse for developers is available. Rather than admit to a PSL coding error or a lack of understanding of PSL features, the first thing to do is to blame the optimizer. This is poor judgment in general, because the problem is much more likely to be the former. The easiest way to prove that it is not an optimizer bug is to turn off the optimizer by adding a “pragma” statement to the start of your PSL script:

```
# Turn off the optimizer with a pragma statement
pragma "-O0"
.... # rest of the PSL script
```

By re-testing with this one extra line of PSL code, the optimizer can be ruled out, and the real cause of the problem can then be investigated.

23.8 KM Lint Common Error Checking Tool

The “KM Lint” tool is for automatic detection of the most basic common KM and PSL coding errors. Actually, two versions of it are available in the PATROL KM Toolkit:

- `KMLINT_TOOL` application class
- Menu command version

The full Knowledge Module version is called `KMLINT_TOOL` and can be loaded on the Developer Console. This version automatically starts running and generates hierarchical iconic views of the various application classes and shows red icons for their errors.

The menu command version can be found on the “Development Tools” menu on the computer icon, and also in the “Deployment” computer icon menu if a version of the “Deployment KM” is installed. This version is user-initiated when you choose the menu command, rather than automatically starting its error detection once loaded.

The “KM Lint” tool is named after a well-known UNIX C debugging tool called “lint.” It uses a number of heuristics to analyze the “.km” and “.psl” files, and also the agent’s internal symbol tree. Some of the PSL and KM errors it detects include:

- Compilation errors in PSL files
- Serious compilation warnings in PSL files
- Full file pathnames in icons
- Full file pathnames in PSL file references
- Localized knowledge stored in KM
- `UNITS` or `TITLE` not set in parameter definitions
- Misplaced brackets in PSL `get` conditions

The report generated by KM Lint can detect major problems. It is also a good stylistic check in that it can inform you of some less serious infractions with respect to usual KM style.

Common KM Development Pitfalls

The best way to avoid programming mistakes is to know the pitfalls. This chapter aims to raise the awareness of the most common programming errors in KMs and PSL coding. Some of these problems are automatically detected by PSL compilation or runtime warnings, provided that you pay attention to these. Unfortunately, others can lead to quiet failures and frustrating PSL debugging sessions where you learn about them the hard way.

24.1 KM Design Pitfalls

When creating a KM, many moving parts must be considered, including KM files, PSL files, libraries, and even other KMs. A number of common pitfalls await in the use of these different components. These pitfalls can be easily avoided through awareness and prevention.

24.1.1 Scheduling Pitfalls for PSL Scripts

Reliance on particular scheduling orders for PSL scripts in a KM is prone to error. Very little defined order of execution occurs for the various parts of a KM. For example, you have no guarantee that “startup” or “setup” actions will run first, no guaranteed order of parameter execution, and no guaranteed order of discovery for multiple applications. However, some scripts are guaranteed before others e.g., prediscovery is guaranteed before discovery and parameters.

The lack of discovery script ordering can be a problem for nested instance classes. For example, you cannot guarantee that the database server discovery script runs before the tablespace or user class discovery script. The only way to handle this uncertainty is through explicit use of the PSL locking facilities or other tests.

24.1.2 Dependencies on Other KMs

The majority of KM designs should be independent of other KM data. The main area of possible accidental dependence on other KMs is via PSL `get` or `set` using data generated by another KM. For example, the UNIX KM's computer classes will generate information such as the system version level and store it in the agent symbol table. Relying on this data can potentially cause problems on machines where the UNIX KM is not loaded (i.e., if the user does not want system monitoring), or on non-UNIX systems.

Legitimate dependency on other KMs can arise in a few cases. One case is where data about system attributes, such as filesystems, users, or processes, will be merged with the application data to generate more useful reports. For example, a database KM could use the operating system KM's data about filesystems to determine into what filesystems the database is storing tables. Another related form is event correlation, whereby the database KM could determine whether a performance problem is also related to a disk being full by testing the filesystem performance data from another KM. In all these cases, two possible solutions arise: either make the other KM a requirement for the application KM, or else check for the other KM and gracefully degrade performance in the absence of the KM. Checking for the other KM should be performed even in the case of a requirement, purely for supportability improvement. Checking for the presence of another KM is simple via PSL `exists`, `get`, or other symbol table access functions.

24.1.3 PSL Libraries Changed without Agent Restart

One common cause of frustration during KM development is odd behavior. For example, if something still fails and you thought you fixed it, you can waste a lot of time. One of the most common causes of this confusion is if the correction is to a PSL library, and you did not follow the steps required to use the new version of the library at the agent:

1. Re-compile the PSL source code with "`psl -l`" to rebuild the library "`.lib`" file.
2. Re-compile any other PSL libraries (or binaries) that use that PSL library.
3. Copy the library "`.lib`" file(s) to the remote agent (e.g., via FTP). The "Commit KM" does not send the new PSL library to the agent automatically.
4. Put the new "`.lib`" file in `$PATROL_HOME/lib/psl` on the agent.
5. Restart the agent (either by killing and restarting it, or via the "Re-initialize Agent" menu command).

Forgetting to restart the agent after installing new PSL libraries is a common mistake. In PATROL 3.2, you can explicitly check the last step to determine whether you have changed a PSL library without restarting the agent. The new agent internal command `%DUMP LIBRARIES` gives you a status warning about libraries that have changed since the agent started. Also, a warning message in PATROL 3.2 appears when the agent detects a library that has changed during compilation of a script.

The step of re-compiling any other dependent libraries is an easy one to miss. If the PSL library is a basis library for other PSL libraries, these PSL libraries also need re-compilation.

24.2 PSL Syntax Errors

PSL is a programming language with pitfalls like any other. These are the simplest types of errors to list. In fact, many of these errors are also documented in the PSL Reference Manual.

24.2.1 Operators `&&` and `||` Do Not Short-Circuit Evaluation

Experienced C programmers will find themselves expecting the PSL `&&` and `||` operators to short-circuit execution, but in fact they do not. The second operand of these operators is always evaluated and never short-circuited by the first operand. This discrepancy is usually a minor performance pitfall, rather than a major error. Refer to the PSL Reference Manual for more details.

24.2.2 Prefix and Postfix Operators `++` and `--` Both Act As Prefix

PSL has no concept of a postfix operation. Postfix `++` and `--` actually act just like the prefix operators. Refer to the PSL Reference Manual for more details.

24.2.3 Break and Continue Statements Not Supported in PSL

The words “break” and “continue” are not statements in PSL, and will be “null effect” statements receiving warnings. The correct keywords in PSL are the “last” and “next” commands. Refer to the PSL Reference Manual for more details.

24.2.4 Assignment in an ‘if’ Statement

A common coding error is to use the assignment operator instead of the equality relational test. The incorrect code is “if (x=y)” and the correct code is “if (x==y)”. A PSL compilation warning is emitted for any detected case of this problem.

24.2.5 Null Effect Statements

One common coding pitfall is to forget to add an operation to a command. For example, the statement “x+1;” does nothing to change “x” and gets a warning, and the corrected code is “x=x+1;”. Another example is mistakenly assuming that a PSL built-in function changes its arguments, such as “tolower(x);”, which also does not actually change “x”, and the correct code is “x=toupper(x);”. A PSL compilation warning is emitted for any detected case of this problem.

24.2.6 PSL [...] List Operator with Empty String in PSL Response

A strange feature of the [...] list operator exists, regarding empty string arguments. If one of the arguments in the list is the empty string, it is simply ignored, rather than a blank line being added. This idiosyncrasy makes the operator different from other PSL functions such as the set

operators, which allow the empty string as a member of a set. However, this feature of the [...] operator is usually harmless.

One case commonly arises with the PSL `response` function, where this feature causes a coding bug. Most of the PSL Reference Manual examples use the [...] operator, which is certainly convenient for expressing the complex `response` function syntax. An example of this error can be seen in the code that pops up a non-echoed text label element:

```
TEXT = 12;
ret = response("Title", "", "h=500,w=500",
               [TEXT, "Label", "Default", "e=1"] ); # Correct
```

The `response` element string from the [...] operator becomes “12\nLabel\nDefault\nne=1”, which is correctly handled. However, this PSL code fails if the default text is the empty string:

```
ret = response("Title", "", "h=500,w=500",
               [TEXT, "Label", "", "e=1"] ); # Fails
```

The `response` element string from the [...] operator becomes “12\nLabel\nne=1” rather than the expected “12\nLabel\n\nne=1”. The empty string for the default string has disappeared because of this feature in the [...] operator of PSL. The workaround is to use the format of the string literal instead of the [...] operator.

```
ret = response("Title", "", "h=500,w=500",
               TEXT . "Label\n\nne=1" ); # Correct!
```

24.3 PSL Built-in Function Errors

The PSL language defines not only a basic syntax, but also a pre-defined library of built-in functions like `set` and `grep`. The use of these function calls has a lot of pitfalls. Some of these functions interact with the agent’s dynamic data in non-intuitive ways.

24.3.1 PSL `grep` Reversed Function Arguments

The function arguments to PSL `grep` and PSL `index` are reversed. PSL `grep` has pattern and then text, whereas `index` has text and then pattern. A common mistake is to mix these arguments, particularly when you are changing code from one to the other. This occurs commonly when micro-optimizing code to replace `grep` calls with `index` calls.

24.3.2 `get` on Simple Discovery Agent Macros in PSL Discovery

A number of the instance built-in macros appear to contain useful information. However, some of these are available only if the discovery method is “Simple” and not “PSL.” Therefore, using PSL `get` on these variables will fail for an application with PSL discovery. The variables

involved in this PSL pitfall are “procParentPid,” “procCpuTime,” “procCommand,” “procSize,” “procResidentSize,” “procPercentCpu,” “procUser,” “procPid,” “procName,” and “procStatus.”

24.3.3 Symbol Table Execution Context for PSL get/set Differs

One common cause of failed calls to PSL `get` or `set` is that different types of commands have a different “context.” The context of the various types of PSL scripts is as follows:

1. Menu commands—Instance level. `get ( "name" )` returns application instance name.
2. InfoBox commands—Instance level. `get ( "name" )` returns application instance name.
3. Parameters—Parameter level. `get ( "name" )` returns parameter name.
4. Recovery commands—Parameter. `get ( "name" )` returns parameter name.
5. Discovery script—Application level. `get ( "name" )` returns application name.
6. Pre-Discovery script—Same as discovery.
7. PSL Debugger script execution—Computer level (“/”).
8. Standalone “psl” interpreter execution—Computer level (“/”).

This situation is clearly confusing and you must take care when writing the different PSL scripts. If a PSL `get` or `set` is not having the correct effects, this might be the cause.

On the plus side, this feature can be used to your advantage if you want to use PSL code for different reasons. This feature allows you to detect the type of script currently running. For example, this can be useful for an error message routine to auto-detect the origin of the message.

24.3.4 PSL Function Support Limited in Standalone ‘psl’ Interpreter

The ‘psl’ standalone compiler can also be used to interpret scripts in a limited manner. PSL scripts often have different results when run in the standalone interpreter as compared to execution in the agent. The standalone interpreter executable ‘psl’ does not support all PSL built-in functions, but only those that do not rely on any agent data structures. Agent object functions such as `create`, `destroy`, `exists`, and related functions do not work. Subprocess launching via PSL `system`, `execute`, and `popen` does not work. However, PSL file operation functions do work correctly, because they do not rely on internal agent structures. Functions that are fully supported because they are solely internal to the basic PSL language features include all operators, string functions, list functions, set functions, and mathematical functions.

Another difference relates to the agent symbol table. The symbol table is only partially mirrored, with some of the “/” top-level macros supported. The execution context of the scripts relative to the symbol table is always “/”.

24.3.5 popen Global Channels Are Not Cleaned Up by KM Unload

Global channels are not automatically destroyed or removed when a Knowledge Module is unloaded. So if your KM used a global channel, when someone stops running the KM on the

agent, the global channel and its child process can still exist. You can do nothing as a KM developer to avoid this situation. However, this does raise the issue that when your KM is creating a global channel, before doing so, it should check to see whether the global channel already exists. If it still exists, perhaps it can be used, or perhaps it should be shut down and a new one started.

24.3.6 Child Process Data Buffering in PSL read

One problem when using PSL `read` to gather output data from a child process spawned by a PSL process occurs when the operating system buffers the data. This can occur on some operating systems if the child process does not explicitly flush its buffers when generating data. So if you write a child process specifically to use with PSL `popen` and PSL `read`, you need to flush the buffers explicitly to ensure that PSL `read` will not block. The C function call to flush buffers is “`fflush,`” and this needs to be called whenever the child process generates new text data to standard output.

24.3.7 Read-Only Agent Macros in PSL set

Not all the macros in the agent symbol table are writable. When using PSL `set`, ensure that the variable is not one of the read-only variables. A PSL runtime error is created for this problem to help identify the problem. Some of the common agent macros that are not read-only include “`username,`” “`password,`” “`environment,`” “`AlarmMax,`” and “`AlarmMin.`” In some cases, these read-only variables can be set via the PSL `pconfig` function to change the agent configuration. The “`status`” variable is a special case because it was read-only in PATROL 3.0 and is now writable for parameters in PATROL 3.1 and above, but still read-only at the instance or application level.

24.3.8 PSL Response Dialogs Not Auto-Sized on PATROL UNIX Console

The PSL `response` function does not auto-size the GUI dialog on the UNIX X/Motif console, but needs to be explicitly set via the “`w=`” and “`h=`” preferences in the PSL code. On the PATROL 3.2 Windows NT console, the GUI is auto-sized. This difference can be a compatibility bug if you are doing development only on the PATROL Windows NT console. All of the menu commands might look quite poor on a version 3.1/3.2 PATROL UNIX console.

24.3.9 `getenv` and the `DISPLAY` Environment Variable

The environment variable “`DISPLAY`” has somewhat confusing semantics when used in PSL `getenv` or in child processes spawned by PSL `system`, `execute`, or `popen`. In menu commands, the `DISPLAY` environment variable should refer to the display of the console running the menu command. However, in many other cases, the interpretation of this environment variable is unclear.

24.3.10 PSL `change_state` Can Suppress Parameter Execution

When a parameter threshold is exceeded and the parameter and instance goes into alarm, all parameters including the one in alarm will continue to run normally. However, using the PSL `change_state` function to set an instance to `OFFLINE`, `WARN`, or `ALARM` causes the immediate suspension of every parameter script for that instance. The `change_state` that sets it back to `OK` will restart the parameters, but they restart from the beginning rather than where they were stopped. This discrepancy is very often a cause of confusion with PSL `change_state`, and because of the immediacy of the parameter stopping, it can potentially fall in the middle of important synchronization logic.

The preferred solution to this method is to use a special parameter for the application state and set its value, rather than using PSL `change_state`. This is achievable by using PSL `set` on “value,” or for PATROL 3.1 and beyond, the explicit setting of the “status” variable for the parameter to “WARN” or “ALARM.” By using the parameter to raise the alarm, all the parameters will continue to run.

Note that in some cases, you do not want parameters to continue to run. For example, if the discovery script detects a condition that it knows will cause all parameters to fail, stopping all of the parameters might seem convenient. However, a better approach usually is to degrade the parameters gracefully via some internal KM tests rather than using the brute-force tack of immediate suspension.

24.3.11 Blackout Suppresses Parameters

The PSL `blackout` function is not very commonly used, and does not perform a function that most KMs need to achieve. In using `blackout`, beware the fact that it will immediately suspend all parameters of the blacked out application instance, similarly to that from the built-in PSL `change_state`.

24.3.12 PSL FIFO and Named Pipe Operations Fail

The PSL `fopen`, `read`, and `write` functions do not work with FIFOs or named pipes. On some operating systems, the use of PSL operations on such a file can actually hang other PSL processes and all other agent operations (including responding to the console’s messages) until the file operation completes, making PSL `read` operations on FIFOs particularly unusable.

24.3.13 PSL `in_transition` Function

The PSL `in_transition` function has obscure semantics. The best recommendation is to avoid this function. The meaning is tied with full discovery and the process cache, and won’t be discussed here. You are better off to implement what you want to achieve via other means. Since the main reason to use this is to time some effects related to application instances, you can achieve this by storing the initial time in an instance variable, and then check it explicitly. This makes clear what you are trying to do, not to mention actually achieving what you want.

24.3.14 Trimming `\r` Carriage Returns for Portability

One important issue on Windows NT and VMS is the extra carriage return characters (i.e., `\r` characters) that can appear in strings. Some of the sources of strings containing `\r` characters are:

- Command output—PSL `execute`, `system`, and also PSL `read` on a `popen` channel
- File contents—PSL `cat`

Although all PSL string handling functions are identical across platforms, they do not treat the `\r` characters specially. For example, PSL `nthline`, `ntharg`, and other string functions treat the `\r` character as a special hidden character, rather than treating it as a separate line like the `\n` character.

The simplest solution to this problem is to use the PSL `trim` function to remove `\r` characters from any strings. This is simple to achieve in code:

```
text = trim(text, "\r");    # remove carriage return characters
```

24.3.15 `nthline` versus `nthlinef` Function

Some confusion surrounds the pair of functions called `nthline` and `nthlinef`. The `nthlinef` function was added later as a fixed version of `nthline`, but leaving `nthline` as is for backward compatibility reasons. The difference in functionality regarding `nthline` and `nthlinef` occurs regarding empty lines. The `nthline` function simply ignores empty lines. The following example should help clarify:

```
str = "1\n\n2";  
a = nthline(str, 2);    # Result is 2  
b = nthlinef(str, 2);   # Result is empty line
```

Generally speaking, your code should use `nthlinef` rather than the older `nthline`. For example, when analyzing the PSL response function return codes, be sure to get the correct line number. If a PSL response function has a text widget and the user does not input any data for that line, the use of `nthline` to get the returned codes will yield the wrong line. Therefore, always use `nthlinef` with response return value processing. The only time you want to use `nthline` is in cases where empty lines have to be ignored.

24.3.16 `ntharg` versus `nthargf` Function

Although the issues of `ntharg` and `nthargf` are similar to `nthline` and `nthlinef`, which to use is not such a clear distinction. The difference is related to multiple adjacent whitespace separators, just as `nthline`/`nthlinef` differed on multiple adjacent newline separators. Here is an example using tab characters:

```
str = "1\t\t2";  
a = ntharg(str, 2);    # Result is 2  
b = nthargf(str, 2);   # Result is empty string
```

Here the recommendation of one over the other is not strong. In many cases, multiple whitespace characters have to be ignored, making `ntharg` the correct choice. For example, in output from a command with multiple columns, you would often want to ignore it if one or more spaces or tabs are between the columns. In a smaller number of cases, exactly one separator character must appear between columns, in which case `nthargf` is the correct choice.

24.3.17 `fseek` Constants Are Not Predefined

An example in the PSL Reference Manual makes `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` appear to be predefined constants that can be used with PSL `fseek`. Unfortunately, these are not built into the language, and you must explicitly define them, and give them the correct values via assignment statements.

24.3.18 `grep \<` and `\>` Word Boundaries

The specialized patterns for word separators, `\<` and `\>`, are not actually fully supported by PSL regular expression code in some agent versions. I recommend not using these patterns with PSL `grep` or the other PSL regular expression operators. This problem should be resolved in a 3.2 maintenance agent release, but early 3.2 and pre-3.2 agents have this problem.

24.4 KM Attribute Errors

Any KM consists of two types of “knowledge”: static KM attributes and dynamic PSL scripts. Various possible errors can occur in the specification of static KM attributes. The console development GUI prevents some mistakes, but some others can sneak through.

24.4.1 Missing `TITLE` or `UNITS` for Parameter

A common KM attribute mistake is defining a consumer or standard parameter but not setting all the display attributes. Failure to set the units or title attributes means that the graphs in the console will not have a meaningful title or units heading. The “KM Lint” tool will find this problem, or it can be found by manual examination of the “.km” file.

24.4.2 No Border Range with Other Ranges

A stylistic recommendation is that a parameter should always have a border range if it has another alarm range. No parameter should have only Alarm1 or Alarm2 ranges but no border range. The reason is mainly to catch any odd cases. For example, if a disk full condition has Alarm1 at warning from 95-98, and Alarm2 has alarm from 98-100, what happens if the operating system for some reason reports 101% full? No alarm occurs. For this reason, parameters such as this should have a border range, and in this case of a percentage parameter, a border

range of 0 to 100 seems appropriate, to raise an alarm for a negative value or one greater than 100.

24.4.3 Alarm Ranges with Incorrect Values

Although not through the Developer Console GUI, manual editing of alarm ranges can lead to mistakes. The minimum should clearly not be greater than the maximum. Furthermore, a number of stylistic issues arise regarding alarm ranges. For example, when a warning range is 95-97 and the alarm range is 98-100, this parameter actually will not raise an alarm or warning at values like 97.5. The solution is to ensure that the ranges actually intersect at one margin in order to cover all the possible values. In this case, that would mean using 95-98 as the warning range, or 97-100 as the alarm range.

24.4.4 Full File Pathname for Icon Names

The icon name for the OK and not OK icons should not be a fully qualified pathname with `/` or `\` characters. It should just be a single icon filename, just the basename of the filename. Otherwise, the console might not be able to find the icon file that is specified when the KM is run on another console.

24.4.5 Full File Pathname for PSL File Loading

The PSL filenames for external files that are loaded from the KM file should not be a fully qualified path name with `/` or `\` characters. This precaution ensures that both the console and agent can find the correct PSL file when loading the KM. Normally, the Developer Console will ensure this, but manual editing of the KM could lead to this error.

24.4.6 Localized Knowledge in a KM

Although clearly frequently the user of a KM has localized customizations, this condition is not good for shipping a new KM. The local customizations will be ignored by other agents, unless the machine names accidentally match, since the customizations are hostname based. However, local customizations have another risk in that the KM developer might accidentally be developing, debugging, and testing in the localized settings. Whether you are editing a global or local parameter or menu command is not always obvious in the Developer Console. The “KM Lint” tool can find and warn about local customizations in KM files.

KM Portability

An important advantage of a KM as a form of application monitoring and management is the ease with which the KM can be ported to a new hardware or software platform. This advantage is achieved through the portable infrastructure layer created by the agent, KM, and PSL features of PATROL. However, PATROL cannot fully insulate the KM developer from the idiosyncrasies of the various platforms. Although all base PATROL functionality is portable, the boundary between PATROL and the outside world is the point where PATROL loses control, and the KM developer has to know with what to deal. Two main areas of the outside world can affect a KM:

1. Operating system—Differences between the external world of Windows NT and UNIX
2. Application—Differences in how the application runs in different contexts

Both these areas have their own special issues. This chapter aims at arming the KM developer with the knowledge and tools with which to insulate the KM against non-portable external behavior.

25.1 Operating System Portability

The PATROL agent is a virtual layer of abstraction above the operating system on which it runs. The KM executes within the agent's space rather than directly on the operating system. The agent insulates the KM from a number of portability issues. All the purely agent-related functionality within the agent is portable across all the various agent platforms. For example, the internal PSL syntax, operators, and control structures are portable.

Areas of non-portability arise at the interface with the operating system. The agent must gather information from the operating system, and this information is necessarily different on

different agent platforms, meaning that issues such as files, processes, and spawned executables have different behavior. Although PSL insulates against some of the porting issues, it cannot hide all the differences, nor would you want it to do so.

25.1.1 Portable Areas of PSL

The PSL language is portable for all the specific areas of functionality that occur fully within the PATROL space. Unless the PSL function has to go to the external operating system, all the functionality is identical across all operating system ports of the PATROL agent. Some of the specific areas where portability is ensured include:

- Control Flow—basic statements such as `if`, `else`, `while`, `foreach`, `switch`, etc.
- Operators—mathematical, string, logical, relational, conditional, ternary, etc.
- String built-in functions—e.g., string functions, list/set functions, sorting, and so on
- Agent symbol table—the variables controlled via PSL `get`, `set`, and other PSL functions
- KM objects—instances, parameters, recovery, menus, and so on are all fully portable
- Inter-PSL actions—locks, condition variables, and other synchronization primitives

25.1.2 Non-Portable Areas of PSL

The areas where PSL has to access the operating system for information are the causes of non-portable functionality. Although the functionality is generally the same across platforms, the results of these operations depends completely on the external environment. Some of these areas include:

- File access functions—e.g., `file`, `cat`, `fopen`, `fseek`, `ftell`, `read`, `readln`, `write`, `close`, etc.
- Process table analysis functions—e.g., `process`, `proc_exists`
- Process cache cycle dependency—e.g., `full_discovery`, `in_transition`.
- Child process launching—e.g., `system`, `execute`, `popen`, `read`, `readln`, `write`, `close`, and the “`exit_status`” special variable
- Environment variables—e.g., `getenv`.
- SNMP functions—e.g., `snmp_get`, `snmp_set`, `snmp_walk`, and so on
- Special areas—e.g., PSL internal function

Minor areas of non-portability in PSL are:

- `printf` and `sprintf` obscure formats—depends on the agent platform C library’s `printf` formats. However, the majority of formats are widely supported on all platforms.
- mathematical overflow and error-handling—e.g., some mathematical libraries handle overflow in functions like `atan` differently. However, remaining within the legal bounds ensures portability.

- `asctime` formats—depends on the agent platform C library's `strftime` formats.
- random numbers—different platforms have different characteristics.

These minor areas can usually be ignored by the KM developer. The usual development does not run into any problems in these obscure areas.

25.1.3 Carriage Return Characters in Text Data

Carriage return characters pose a portability problem on Windows NT, OS/2, and VMS for processing of text data. UNIX platforms separate text file lines with a newline character (i.e., `\n` in PSL vernacular). Windows NT, OS/2, and VMS all use a pair of newline (`\n`) and carriage return (`\r`) characters to separate lines in a text file. This issue poses problems in PSL in a number of areas related to text data:

- PSL `cat` on a text file
- PSL `system` or `execute` returning text data output from a child process
- PSL `read` or `readln` on a `popen` channel returning text data output by a child process

The effect is that the string variables containing text data from these sources will still contain the `\r` characters. This is problematic because the PSL string operation functions do not treat the `\r` characters as special, and will consider only the `\n` as the separator. For example, if you use PSL `ntharg` to return the first or second line, the returned line will still contain a `\r` character. This character is also difficult to see when debugging the script with debugging output statements.

No PSL functions handle `\r` characters as separators for lines. Some of the places that this fact poses a problem when processing these strings are:

- Line and word processing—`ntharg`, `nthline`, `nthargf`, `nthlinef`, PSL `foreach` statement
- Regular expression search—`grep`
- List or set operations—`union`, `subset`, `difference`

The solution to the `\r` problem is a pragmatic, if not particularly elegant, solution. The PSL `trim` function can be used to remove these characters before processing the string data. This is not elegant in that you have to remember to do it for every place where the KM gathers external text data. However, it is only one line in each such place, and an example involving PSL `cat` looks something like:

```
text = cat(pathname);      # Get the text data contents of a file
text = trim(text, "\r");    # Remove all \r characters from the text
```


One possible method of handling this problem is to wrap all calls to PSL `cat`. This requires calling another wrapper function instead of PSL `cat` each time:

```
function portable_cat(fname)

{
    local text, save_errno;
    text = cat(pathname);      # Get the text data contents of a file
    save_errno = errno;        # save errno just in case
    text = trim(text, "\r");    # Remove \r characters from the text
    errno = save_errno;        # reinstate errno
    return text;
}
```

Not all places have this problem with the `\r` character. One place where the carriage return character is automatically handled is the PSL file reading operations. The PSL `read` function on a channel opened via PSL `fopen` (with a non-binary mode) will not return the carriage return characters in the output. PSL `read` will automatically remove these characters from the returned result. So says the theory, but in practice you can trim them also, in case this isn't true on all agent platforms and versions.

25.1.4 Process Statistics Analysis

Operating system processes are another area where portability concerns arise. Processes are an integral part of the operating system layer, and although PSL attempts to hide some of the differences, some properties of operating system processes are not abstracted.

The main abstraction layer for gathering statistical performance data about processes is the agent's process cache. The only reason it is called a "cache" is that it records a snapshot of the process information within the agent's memory for rapid access. This is a regularly updated record of operating system processes and a number of statistics about them. The process cache is available via the PSL `process` function. For example, the process cache contents can be viewed via:

```
txt = process("");    # List all agent process cache processes
print(txt);
```

The process cache is built automatically by the agent. The PSL `process` function is very efficient because it is operating only within the agent's internal data structures, and is not performing external process analysis (this has already been done when the agent last updated the process cache). Behind the scenes, the agent is using a platform-specific method to gather the data. This method may be direct operating system statistic access or a system command. The process cache is updated according to the process cache cycle, which is by default 300 seconds (5 minutes), but can be configured on a per-agent basis. The process cache can also be updated

immediately via the special agent built-in command `%REFRESH_PROC_CACHE`, which can be launched as an ad-hoc command, or via PSL using the `system` function with this command.

Unfortunately, the process cache has a number of limitations. Clearly, it may be somewhat out-of-date, possibly by almost 300 seconds. The process cache cannot be updated for one particular process only. The whole process cache must be updated as a group. This constraint poses problems for a KM that needs to get regular up-to-date information about a small group of processes, if this data is needed more frequently than the process cache update cycle.

Portability of the PSL `process` function to all agent platforms is also somewhat limited to non-UNIX platforms. All UNIX agents should function identically, but some of the attribute fields (e.g., memory used by the process) are not always available in a particular agent port, such as Windows NT or VMS agents. The Windows NT agent is also different in that the number of columns and the meaning of each column are different from the UNIX agent. The Windows NT 3.2 agent has one less column, with the “status” column missing. Therefore, this imposes some ugly coding style to get process statistics in a portable manner. For example, on a 3.2 agent, the code to get the process CPU seconds used and the memory size is:

```
# Work around a PSL portability "feature" on NT versus UNIX
if (get("/appType") == "NT") {
    col = 5;
}
else {
    col = 6;    # UNIX and other non-NT platform
}

ret = process("*");
# get the CPU usage column
ret = ntharg(ret, col);
```

The PSL `process` function on Windows NT has also been revised in the 3.2 NT agent. Therefore, the number, position, and whether the column contains real values or zeros may have changed between 3.1 and 3.2 versions.

The PSL `proc_exists` function is another PSL primitive that examines the operating system process space. This function always immediately queries the process tables and has up-to-date information. However, its only value is in the presence or absence of a particular process, and it returns only a Boolean value. PSL `proc_exists` cannot gather any other more detailed statistics about a process. It also requires the PID of the process, and cannot be used to search for processes based on the name.

25.2 Launching Child Processes

Child process launching is an area with a number of portability concerns. The behavior of the child processes may differ, or they might not even exist on a particular platform. The KM can launch child processes into the operating system in a number of ways, including:

- PSL `system`, `execute`, and `popen` functions
- OS command types

The first problem with child process launching is whether the command exists on a particular platform. For example, the Windows NT “`dir`” command fails on UNIX, and the UNIX “`ls`” command is absent on Windows NT. The first level of portability is to ensure that the correct command is executed on the correct platform. No easy way to achieve this exists, other than to explicitly check the platform, which is possible via the “`appType`” variable (discussed on page 365). Ported KM code might often look something like:

```
platform = get("/appType"); # What type of agent am I?
if (platform == "NT" || platform == "OS2") {
    cmd = "dir";
}
else { # Presumably UNIX (assuming the KM disallows MVS/VMS/etc.)
    cmd = "ls -la";
}
txt = system(cmd);      # Launch the command
```

25.2.1 Child Process Error Handling

Launching the wrong command on the wrong platform causes the command to fail. Unfortunately, the PSL handling of this case is not as elegant as one would like. The agent will launch a child process attempting to run this command. On UNIX, it will typically use a shell, and on Windows NT, it will use the command interpreter to attempt to run this child process. The errors returned by the PSL `system`, `execute`, and `popen` functions are therefore dependent on the behavior of the shell or command interpreter, and this differs across platforms.

On UNIX, the text returned from a PSL `system` call that fails is typically a shell-specific error message. This message differs, depending on whether `sh`, `csh`, or `ksh` is the default shell for the PATROL user. Similarly, the messages on Windows NT and OS/2 are from the operating system command interpreter rather than PATROL-specific error message text. You can check for the error message patterns, which is a pragmatic although inelegant solution.

The PSL `exit_status` function can be used to determine the exit code of the child process. This code may actually come from the shell or command interpreter, but is generally accurate, since shells usually pass along the child process code, or have a valid failure code if the command is not found.

Unfortunately, the PSL `exit_status` special variable had early problems, and has been failing in pre-3.2 agents on some platforms and in some cases (due to a race condition). The 3.2 agent has resolved this issue, but compatibility with older agents may be questionable.

Prevention of command failures is not easy. In PSL, you cannot prevent an invalid command execution before launching the child process. No PSL function will check the `PATH` environment variable and determine whether the command is valid and available.

25.2.2 Child Process `popen` Pipes

The PSL `popen` function is a special function that launches a child process and creates pipes between the agent and the standard input and output of the process. This allows PSL `write` to send commands to the process's standard input, and PSL `read` to read the results from standard output and standard error. These are typically used in conjunction with global channels to use a command to interact with the database or application that the KM is monitoring.

The Windows NT implementation of PSL `popen` had early problems, particularly with early releases of PATROL version 3.1 and earlier. Although the problems should be resolved in the 3.2 agent, be aware of this possibility when writing or porting a Knowledge Module for Windows NT on PATROL pre-3.2 versions.

VMS execution of PSL `popen` has a particular quirk. This may be resolved in the 3.2 agent, but earlier agents have a limitation of PSL `popen` that other non-VMS platforms do not have. The PSL `read` function will not correctly report failure when the `popen`-launched child process terminates, and will block indefinitely if called to read output from a process that has terminated. This allows PSL `popen` to be used as an interactive channel to a child process, but does not allow you to launch PSL `popen` for a process that terminates. Since no form exists on non-blocking polling PSL `read` call (in 3.2), to avoid having a PSL `read` operation that will block is difficult. Although PSL `read` can be used non-blocking with codes like "-1", these will not prevent PSL `read` from blocking if no data exists at all, although they will return early if some data exists (i.e., they do not block until the entire process has terminated).

25.2.3 Backgrounding Child Processes

The creation of child processes in the background is a particular problem for a KM. The typical scenario is a KM that needs to re-launch a daemon process in a recovery action, once it determines that the daemon is failing or absent. Because of the manner of child process launching using the UNIX shell on some UNIX platforms, some of the common attempts have failed, and will block the call to PSL `system` until the child process completes. For example, here is a simple try on UNIX using the `&` syntax:

```
txt = system("my_daemon &"); # UNIX background with & syntax
```

This sometimes works and sometimes does not, depending on the UNIX platform. Some UNIX platforms are failing because the shell is waiting for the file descriptors of the child process. Hence, the following sometimes improves the portability by hiding the file descriptors from the shell (this is the use of UNIX `csh` syntax):

```
txt = system("my_daemon >/dev/null >&/dev/null &");
```

This is not shell-specific, and the solution to that shell portability issue leads to either a separate shell script file that is launched separately, or explicit coding of the shell in the command, such as:

```
txt=system("/bin/csh -fc \"my_daemon >/dev/null >&/dev/null &\" ");
```

When all else fails, the PSL `popen` command comes to the rescue. The PSL `popen` command has the property that it launches the child process and immediately returns control to the agent. Therefore, you can simply call:

```
chan = popen("OS", "my_daemon");
```

This will return control immediately to the PSL process, and leaves the child process running. However, behind the scenes, all output from the daemon is buffered in the operating system, because it is expected to be read by a PSL `read` call. Therefore, a slightly better sequence is to cut the child process away from the agent by closing the `popen` channel.

```
chan = popen("OS", "my_daemon");
close(chan); # close channel without killing child
```

Note that the PSL `close` function is called without a second argument. You should not use the flags in the second argument to kill the child process, because they are not what the aim is when backgrounding.

25.2.4 Command Types

Command types are used in a KM to wrap a global channel, particularly one involving usernames and passwords. Porting issues arise if the command type needs to change for a particular platform. For example, the command needed might differ between UNIX and Windows NT. We then need the command type to behave differently on different platforms.

A macro feature in command types allows dynamic changes to command types. The syntax is a `%{ . . . }` macro placed in any of the text fields of the command type. This macro is evaluated based on searching the agent symbol table tree, which is the same tree maintained by PSL `get` and `set`. Therefore, to change a command type on a per-platform basis, the command type need only use a `%{ . . . }` macro, and the PSL code can use PSL `set` to change the value of that macro.

A simple example would be when a command to be executed has a different pathname on UNIX and Windows NT. The solution is to check the “`appType`” variable, and then use PSL `set` to change a variable in the symbol tree.

```
if (get("/appType") == "NT") {
    set("/my_cmd", "\\bin\\mycmd.exe"); # NT executable
}
else {
    set("/my_cmd", "/bin/mycmd.exe"); # UNIX executable
}
```

The command type associated with this PSL code would have a macro `%{my_cmd}` in its text. The above PSL code could easily go in the pre-discovery script for a KM, since it needs be performed only once.

Other examples can be even more dynamic. For example, if you want to dynamically control what options are sent to a command, you would simply place a `%{my_options}` macro in the command type. These options could be changed at any time by `PSL set`, potentially prior to every call to `PSL execute` or `popen` that uses the command type.

In rare cases, the use of `%{ }` macros may not be adequate if you need a different “type” of Command Type. For example, the Command Type functionality differs, depending on whether a `%{command}` macro is present. If one platform needs this functionality and another needs the other type, you cannot use `%{ }` macros to solve the problem. Fortunately, this totally different functionality is needed rarely.

The brute-force solution if the `%{ }` macros are not adequate is duplication of command types with different settings. This involves having more than one command type, or one even for each platform. For example, you might have a command type for all UNIX and one for Windows NT. To use multiple command types, each command type will have a different name, and the KM must use the right name in the right place. The KM must explicitly change its PSL calls to `execute` and `popen` that make use of the command type. The correct name of the command type must be replaced on the correct agent port. This requirement is not difficult to achieve, but involves either touching every call to `execute` and `popen`, or using a global command type variable. The best solution is to centralize all command type and global channel operations into one place (e.g., a PSL library function call), which allows you to address easily the name changes per platform in this localized code block.

25.2.5 UNIX Shell Portability

UNIX shell dependency can be another portability trap, because you cannot be sure which shell is being used. Since all UNIX platforms support the standard set of shells (e.g., `sh`, `csh`, `ksh`), if your script is valid for all of these, no problem occurs. However, a problem arises if you use a shell syntax that is not portable to a particular shell. For example, the file redirection method “`1>&2`” is valid for “`sh`,” but not for “`csh`.” Non-portable shell syntax may occur when launching a UNIX shell script via `PSL system`, or when the `PSL system` call has a direct shell command. A common issue is building an environment setting via “`setenv`” or “`export`” in the actual command text to `PSL system`. This is a risky practice in terms of portability to all UNIX environments with PATROL installed. A better practice is to use KM defined environment variables.

The problem is that not all PATROL installations use the same default shell. The shell used by the agent when launching shell scripts or other shell OS commands is dependent on the agent’s default user account. Typically, this is the operating system “`patrol`” account, although some sites use other names like “`ptr131`” and so on. The default shell of this UNIX user is the default shell used by the agent to launch child processes for parameters and discovery

scripts. It gets even more complicated in that menu commands and info box commands, which are launched by the console, run in the context of the connected user for that console, and use that user's default shell.

Clearly the best solution is to either avoid shell-specific coding style or explicitly mandate which shell is being used. Choosing the shell can be done by coding the command to explicitly use the shell name such as `/bin/sh`, `/bin/csh`, or `/bin/ksh` with a `-c` option or similar. This is less efficient, but portable to all UNIX sites. Another alternative is to save the script command in an external file, ensure that it has a `#!/bin/csh` or similar header line, and launch it via PSL system as a named shell script, rather than internally generated command text.

25.3 File Operations

A number of portability issues surround the various file operations permitted in PSL. The issues and the PSL functions to which these portability issues apply include:

- File creation and permissions—`fopen`
- File attributes—`file`
- Random access to file data—`fseek`, `ftell`
- FIFO and named pipes—`fopen`, `read`, `write`

The PSL `fopen` function with a `"w"` or related file mode can create a new file. This file has ownership and permissions dependent on the operating system user associated with that PSL script. Typically, for a parameter that will be the agent's default account and for a menu command, the user will be running the menu command (i.e., the username and password used by the console to connect to the agent). You have no way to change who owns a file, as this could clearly raise security problems. The permissions of the newly created file are dependent on the default properties of that user. Unfortunately, no explicit control over these attributes is available in PSL, nor is a PSL `umask` or similar function.

The PSL `file` function can return a number of attributes about a file, such as its permissions and modification times. However, some of these fields are, of necessity, zeroed on platforms such as VMS and Windows NT, if the attribute is not relevant on that platform.

The size of a file can be portably generated via the PSL `file` function. This is one of the more portable file attributes that are returned in the value.

```
size = ntharg(file(filename, 1), 5); # Get file size
```

Random access to file data is possible in PSL via the PSL `fopen` and `fseek` functions. The `fopen` function must supply a mode that permits random access to the file. The PSL `fseek` function can then be used to move to a particular location in the file. The PSL `ftell` function also returns the current file pointer.

Although the `fopen` mode does not absolutely have to specify random access to allow the use of PSL `fseek`, it is desirable. For example, PSL `fseek` on a file opened in text mode (i.e., with no “b” specifier) may behave unexpectedly due to the `\n` and `\r` issues. The issues are the same as those for the C programming language. Note also that no “t” specifier is in the PSL `fopen` modes, although this specifier is common in Windows NT and other C libraries.

Another pitfall in PSL `fseek` is the use of the modes `SEEK_CURR`, `SEEK_SET`, and `SEEK_END`. Although an example in the documentation gives the appearance of being so, none of these three symbolic constants is defined in PSL. You must either explicitly initialize these constants to 0, 1, and 2 in the PSL code, or else use the numeric codes in the code directly.

One of the more obscure file portability problems common in C is made harmless in PSL. The rare problem of mixing reads and writes to a file without an intervening `fseek` has undefined behavior in C, but raises no errors in PSL, because PSL explicitly makes the C call to `fseek` behind the scenes. No such problem occurs surrounding PSL `read`, `write`, and `fseek`.

FIFOs and named pipes are not supported by the PSL functions. Although PSL `fopen` can open such a pipe, the PSL `read` function does not handle it correctly, and may block. In this case, the blocking that occurs is not like the normal harmless blocking that blocks only one PSL script, but this sometimes blocks the entire agent, including all PSL processes and all network transport handling. This connection failure leads to a timeout and shuts down the agent as far as the console is concerned, and the connection is lost. The workaround for this problem is to use scripts for the FIFO operations, and use PSL `system`, `execute`, or `popen` to launch these scripts.

25.3.1 Environment Variables

The environment settings are another issue that differs across platforms. These issues of portability are well-known and not specific to PATROL. For example, although the “PATH” environment variable is relatively portable, the UNIX X windows “DISPLAY” environment variable is not.

Environment variables are supported by PATROL in various ways. The PSL `getenv` function allows an efficient method of polling the agent’s environment that is portable to all agents. Special cases of the environment can be retrieved directly by PSL `get` from the agent symbol table; for example, `PATROL_HOME` is stored in “/patrolHome.” The inefficient method of polling the environment is via PSL “`system(“echo $PATROL_HOME”)`,” and this should be avoided because PSL `getenv` is superior in performance and portability.

25.3.2 File Pathnames

The filenames used to navigate through directories differ on various platforms. UNIX uses “/” whereas Windows NT and OS/2 uses “\”. VMS uses the “[x.y]” directory paths. These issues create problems for PSL code wherever it requires a filename, which includes the following PSL functions:

- File statistics—`file`
- File opening—`fopen`, `cat`
- Process launching (executable files)—`system`, `execute`, and `popen`

No help comes from PSL itself. None of these PSL functions have any built-in handling of different pathname formats on each platform. The filename passed as a PSL string to these functions is passed directly to the operating system level API calls. So PSL is non-portable to the extent that the OS API calls are different.

Fortunately, the OS API calls have come together somewhat. For example, Windows NT and OS/2 will accept either “/” or “\” as a pathname separator, simplifying porting UNIX PSL code to Windows NT or OS/2. However, the VMS square bracket format and the other “/” and “\” formats do not mix at all. When mixing VMS and non-VMS, a PSL function receiving the wrong file pathname on the wrong agent will simply fail, since neither PSL nor the operating system will convert the filename.

Another special case is when the separator character is the last character. OS/2 does not handle this separator the same way as do UNIX and Windows NT. For example, the directory filename “/usr/patrol/mydir/” when passed to the PSL file function will find “/usr/patrol/mydir” on UNIX and “\usr\patrol\mydir” on Windows NT, but on OS/2 it will fail, claiming that the directory is absent even if “\usr\patrol\mydir” exists.

Multiple separator characters are another issue. UNIX will treat a “//” in the pathname exactly the same as “/”. Windows NT will allow the duplication of “\” in the middle of a pathname, but not at the start. OS/2 has the same functionality here as Windows NT.

Launching child processes also requires the specification of a pathname to PSL `system`, `execute`, or `popen` functions. The pathname portability issues are unchanged here. The error handling of these functions is not strong because of some quirks in the process launching. For example, on UNIX, you are likely to get a textual error message from the UNIX shell, rather than a value set in the PSL `errno` variable.

Portably building file pathnames is a matter of correct coding to handle the details. The PSL script must explicitly build the filename in a format required by the operating system. For example, the following code is typical in the method and also needed to handle VMS as a special inelegant case:

```
platform = get("/appType");
if (platform == "NT" || platform == "OS2") {
    separator = "\";
} elsif (platform == "VMS") {
    separator = ".";
} else { # Assume UNIX
    separator = "/";
}

# Build the equivalent of UNIX's $PATROL_HOME/lib/psl/myfile.psl
```

```

patrolHome = get("/patrolHome");
myfile = "myfile.psl"; # base name of the file
if (platform == "VMS") {
    # VMS format is [dir.lib.psl]fname.psl
    fname = trim(patrolHome, "]") . separator . "lib" .
        separator . "psl" . "]" . myfile;
}
else { # UNIX, NT, OS/2
    fname = patrolHome . separator . "lib" .
        separator . "psl" . separator . myfile;
}

```

Filename conversion is clearly a candidate for a portability wrapper user-defined PSL function. No built-in PSL support exists for doing this, so it must be coded as a new user-defined PSL function.

25.3.3 Directory Handling

Little PSL support is available for handling directories in the file tree hierarchy. The PSL `file` function can be used to check permissions and other attributes of a directory file, just like it can be used on any kind of file. However, you have no easy way to get a list of files in the directory via PSL, nor does any standard PSL library or XPC library currently offer this functionality.

Although far from elegant, one successful solution is the launching of a child process via PSL system that does a UNIX `ls -l` or a Windows NT `dir` command. The PSL `ntharg` function can be used to cut out the correct column number that contains the filenames.

25.4 Detecting Machine Platforms

The operating system platform on which the KM is executing can be determined via PSL. The special variable `/appType` is predefined within the agent's symbol table. This variable gives a symbol name that identifies the platform type for the agent. Some of the valid name values for the `/appType` macro appear in Table 25-1:

Table 25-1 Agent platform naming

Agent Operating System Platform	Agent Class <code>/appType</code> value
Windows NT (Intel & DEC Alpha)	NT
OS/2	OS2
OS/390 or MVS	MVS
VMS VAX & AXP	VMS

Table 25-1 (cont'd) Agent platform naming

Agent Operating System Platform	Agent Class “/appType” value
AS/400	No native agent exists
NetWare	No native agent exists
Macintosh	No native agent exists
NeXT	No native agent exists
Windows 95	No native agent exists
UNIX Variants:	
Solaris (Sun)	SOLARIS
SunOS	SUN4
IBM AIX on RS/6000	RS6000
Digital UNIX (formerly OSF/1)	OSF1
Hewlett Packard HP-UX	HP
Bull BOS	BULL
Pyramid DC-OSX	DC-OSX
SCO UNIX	SCO
SGI Irix	SGI
System V for Intel	SVR4-i386
System V Motorola	SVR4-m88K

Table 25-1 (cont'd) Agent platform naming

Agent Operating System Platform	Agent Class “/appType” value
SEQUENT PTX	i386-SEQUENT
Linux	Linux (agent not available commercially)

An easy way to gather this form of information about the “/appType” values is to look at the list of computer classes in the console. This shows the same platform names that are produced by /appType. Case of the letters in the class name does matter, unless you explicitly code conditional tests to ignore it.

One common test is to check whether the machine is a UNIX machine. The easiest way is to assume that if it is not one of the other main non-UNIX platforms, then it must be a UNIX variant. The following simple one-liner PSL script can be used:

```
plat = get("/appType");
is_unix = !index("!MVS!VMS!NT!OS2!", plat); # Is it a UNIX agent?
```

This assumes that if it is not a known non-UNIX platform, then it must be UNIX. This form of exclusive negative test has the limitation that the later addition of another non-UNIX agent platform will cause it to be ineffective on that agent platform. A preferable approach would be to use a test for all the valid values of the variable for all UNIX, but a lot exist, and this would not protect when a new UNIX agent is added. The above test is a pragmatic and commonly used solution.

No easier efficient way is available to test whether the agent is a UNIX machine. Perhaps using a PSL file function call to test for the presence of “/bin” and “/etc” would also be valid. However, I prefer the use of “/appType” for performance and consistency reasons.

This simple idea gives rise to simple tests for other platforms such as Windows NT, VMS, OS/2, and MVS. No automatic method specifies that a block of PSL code applies only to a particular platform, so this test of “appType” is the most common solution:

```
platform = get("/appType");
is_nt = platform == "NT";
is_os2 = platform == "OS2";
is_vms = platform == "VMS";
```

25.5 Detecting Operating System Versions

The “appType” variable gives you only the highest-level agent platform information. It does not allow you to distinguish Windows NT 3.51 from 4.0, nor Solaris 2.25 from 2.26. This version information must be determined in other ways.

Windows NT versions are detected by the NT KM file. This is the command class for Windows NT, and the “Setup Commands” are where this information is generated from the registry via the PSL `internal` function. The build number and NT version are available via PSL `get`, after a short initial phase when the PSL setup command runs:

```
build = get("/reg_BuildNumber");    # Get NT build number
ver = get("/reg_OsVersion");        # Get NT version, e.g., 3.51 or 4.0
```

Another issue for Windows NT is distinguishing Intel and Alpha architectures. The agents for both forms of Windows NT simply return “NT” in the “appType” variable. A common way to detect NT Alpha platforms is via PSL `getenv` on the “PROCESSOR_ARCH” environment variable. This returns “x86” on an Intel platform and “Alpha” for an NT Alpha platform.

You also have other ways to distinguish NT Intel and Alpha. Certain registry entries have indicators of the underlying platform. These can be checked via PSL `internal`.

The UNIX computer classes also make OS version information available. This is set in the ALL_COMPUTERS class via a computer class parameter called “OSName,” which is usually based around the UNIX “uname” command and processing its output. The PSL `get` code to get the version number is:

```
version = get("/osVersion");        # Get UNIX version number
```

This too may not be available immediately upon agent startup. It is not set until the computer class parameter is executed. Therefore, take care when using these PSL `get` options in a pre-discovery or discovery script that might be scheduled before the computer class parameters or setup commands. Check whether this variable is the empty string, and add a delay of a few seconds with PSL `sleep` if so.

25.6 How to Port a KM

Porting a KM to a new operating system platform is often a very rapid cycle. Of course, this all depends on a number of factors, involving the source and destination platforms and the application behavior on the various platforms. Porting from one UNIX platform to another is often rapid, whereas porting a KM from UNIX to VMS is not as easy.

The functionality of the monitored application is also a clear factor. If the application behaves identically on the new platform, you will have little trouble. However, if the application stores its log files in different directories or has different formats in its command-line interfaces, then a major effort may be involved.

Some of the KM porting effort can be reduced by thinking ahead. Some guidelines on how to design a KM to minimize porting difficulty include:

- Not hard-coding filename paths
- Configuration settings for executable names

- Configuration settings for executable formats, such as column numbers
- Centralizing your data collection code, such as global channels
- Wrapping file operations
- Centralizing and wrapping all non-portable areas
- Shell script reliance minimized (to improve portability to non-UNIX)
- Wrapping operations for portability

25.7 PSL Portability Reference Table

The details of PSL portability are shown below in Table 25-2. The basic types of portability dependencies include functions acting on the operating system, mathematical function obscurities and C library string formatting idiosyncrasies.

Table 25-2 PSL feature portability

PSL Function	Agent Needed	OS Porting Risk Areas	Portability Comments
acos	3.0	Out-of-range	Mathematical overflow may give slight differences.
annotate	3.1	None	
annotate_get	3.1	None	
asctime	3.0	Odd formats, OS time	Most formats portable but it relies on C library <code>strftime</code> formats. Not all <code>strftime</code> formats supported on all platforms. Results also clearly depend on the OS keeping correct time.
asin	3.0	Out-of-range	Mathematical overflow may give slight differences.
atan	3.0	Out-of-range	Mathematical overflow may give slight differences.
blackout	3.1	None	
cat	3.0	OS files	Files may not exist, have different contents. Issues such as handling of the <code>\r</code> character cause portability differences.
ceil	3.0	Out-of-range	Mathematical overflow may give slight differences.

Table 25-2 (cont'd) PSL feature portability

PSL Function	Agent Needed	OS Porting Risk Areas	Portability Comments
chan_exists	3.0	None	
change_state	3.0	None	
chart	3.2	None	
close	3.0	OS files and processes	Can close a system file or kill a system process depending on the type of channel, and these OS actions might have different effects.
cond_signal	3.0	None	
__config	3.0	None	
console_type	3.0	None	
convert_date	3.1	None	
cos	3.0	Out-of-range	Mathematical overflow may give slight differences.
cosh	3.0	Out-of-range	Mathematical overflow may give slight differences.
create	3.0	None	
date	3.0	OS time	Results clearly depend on the current time. However, the format is identical across platforms.
debugger	3.0	None	
desktop	3.2	None	
destroy	3.0	None	
destroy_lock	3.1	None	

Table 25-2 (cont'd) PSL feature portability

PSL Function	Agent Needed	OS Porting Risk Areas	Portability Comments
difference	3.0	None	
event_archive	3.1	None	
event_catalog_get	3.1	None	
event_check	3.0	None	
event_query	3.0	None	
event_range _manage	3.0	None	
event_range _query	3.1	None	
event_report	3.1	None	
event_schedule	3.1	None	
event_trigger	3.0	None	
event_trigger2	3.1	None	
execute	3.0	OS processes	Launching a child process is clearly OS-dependent.
exists	3.0	None	
exp	3.0	Out-of-range	Mathematical overflow may give slight differences.
fabs	3.0	Out-of-range	Mathematical overflow may give slight differences.
file	3.0		

Table 25-2 (cont'd) PSL feature portability

PSL Function	Agent Needed	OS Porting Risk Areas	Portability Comments
floor	3.0	Out-of-range	Mathematical overflow may give slight differences.
fmod	3.0	Out-of-range	Mathematical overflow may give slight differences.
fseek	3.0	OS files	<code>fseek</code> on a file can have different behavior, such as seeking beyond the end of a file. <code>fseek</code> on a text file in non-UNIX may be affected by the <code>\r</code> carriage return issue depending on the <code>fopen</code> mode.
ftell	3.0	OS files	Depends on external file issues.
full_discovery	3.0	Process cache	Depends on the time it takes for the agent process cache to refresh.
get	3.0	None	
get_chan_info	3.0	None	
getenv	3.0	Environment	KM-set environment variables are portable. <code>getenv</code> will also check the external environment in which the agent was launched.
getpid	3.1	None	PSL pid (not system PID) is fully portable.
getpname	3.1	None	
get_ranges	3.0	None	
get_vars	3.0	None	
grep	3.0	None	
history	3.0	None	

Table 25-2 (cont'd) PSL feature portability

PSL Function	Agent Needed	OS Porting Risk Areas	Portability Comments
history_get_retention	3.1	None	
index	3.0	None	
int	3.0	Out-of-range	Mathematical overflow on large values may give slight differences.
internal	3.0	Agent platform	This PSL function is explicitly non-portable intentionally. The Windows NT, VMS, and UNIX agents have different behavior.
intersection	3.0	None	
in_transition	3.0	Process cache	Depends on the time it takes for the agent process cache to refresh.
isnumber	3.0	None	
is_var	3.0	None	
join	3.2	None	
kill	3.1	None	Kills PSL process, so fully internal and portable.
length	3.0	None	
lines	3.0	None	
lock	3.0	None	Locks are internal; not OS locks.
log	3.0	Out-of-range	Mathematical overflow may give slight differences.
loge	3.0	Out-of-range	Mathematical overflow may give slight differences.

Table 25-2 (cont'd) PSL feature portability

PSL Function	Agent Needed	OS Porting Risk Areas	Portability Comments
log10	3.0	Out-of-range	Mathematical overflow may give slight differences.
ntharg	3.0	None	However, confusion of \r and \n characters may arise, since only \n is used to separate lines.
nthargf	3.0	None	However, confusion of \r and \n characters may arise, since only \n is used to separate lines.
nthline	3.0	None	However, confusion of \r and \n characters may arise, since only \n is used to separate lines.
nthlinef	3.0	None	However, confusion of \r and \n characters may arise, since only \n is used to separate lines.
num_consoles	3.0	None	
pconfig	3.1	None	
popen	3.0	OS processes, NT on 3.2	Launching a child process is clearly OS-dependent. This function has also been failing on Windows NT prior to the 3.2 agent. VMS functionality may also be different slightly.
popup_report	3.2	None	
pow	3.0	Out-of-range	Mathematical overflow may give slight differences.
print	3.0	None	
printf	3.0	Odd formats	The majority of printf/sprintf formats are supported on all platforms. However, some obscure formats are not supported by the C library on which the PSL function is based.
proc_exists	3.0	OS processes	Checks for existence of external OS process.

Table 25-2 (cont'd) PSL feature portability

PSL Function	Agent Needed	OS Porting Risk Areas	Portability Comments
process	3.0	OS processes, NT different column fields returned	Reads the list of external OS processes from the agent's regularly generated internal process cache. Windows NT column meanings differ from UNIX columns in the PSL return string that results.
ProfDefault Options	3.2	None	
ProfGet	3.2	OS time	Results depend on internal OS timings.
ProfOptions	3.2	None	
ProfTop	3.2	OS time	Results depend on internal OS timings.
PslExecute	3.1	None	
random	3.0	OS algorithm	Random numbers are generated slightly differently on different systems, and although they will always be random, they will clearly not always be identical across platforms. Usually this fact is good!
read	3.0	OS file and OS processes	Reading file output depends on the contents of the file, and reading from a <code>popen</code> channel depends on the child process. Also, issues such as the <code>\r</code> and <code>\n</code> problem arise on Windows NT.
readln	3.0	OS file and OS processes	Reading file output depends on the contents of the file, and reading from a <code>popen</code> channel depends on the child process. Also, issues such as the <code>\r</code> and <code>\n</code> problem arise on Windows NT.
refresh_parameters	3.1	None	
remote_close	3.1	None	

Table 25-2 (cont'd) PSL feature portability

PSL Function	Agent Needed	OS Porting Risk Areas	Portability Comments
remote_event_query	3.1	None	
remote_event_trigger	3.1	None	
remote_file_send	3.1	None	
remote_open	3.1	None	
replace	3.2	None	
response	3.0/3.1	None	Dynamic response (D=1) and some newer GUI elements require a v3.1 PATROL console for correct display.
response_get_value	3.1	None	
rindex	3.0	None	
set	3.0	None	
set_alarm_ranges	3.2	None	
share	3.0	None	
sin	3.0	Out-of-range	Mathematical overflow may give slight differences.
sinh	3.0	Out-of-range	Mathematical overflow may give slight differences.
sleep	3.0	OS timing	The exact length of time slept may vary slightly.
snmp_agent_config	3.0	None	

Table 25-2 (cont'd) PSL feature portability

PSL Function	Agent Needed	OS Porting Risk Areas	Portability Comments
snmp_agent_start	3.0	None	
snmp_agent_stop	3.0	None	
snmp_close	3.0	None	
snmp_config	3.0	None	
_snmp_debug	3.0	None	
snmp_get	3.0	SNMP	Results depend on external SNMP data.
snmp_get_next	3.0	SNMP	Results depend on external SNMP data.
snmp_h_set	3.0	SNMP	Results depend on external SNMP data.
snmp_h_get	3.0	SNMP	Results depend on external SNMP data.
snmp_open	3.0	None	
snmp_set	3.0	SNMP	Results depend on external SNMP data.
snmp_trap_ignore	3.0	None	
snmp_trap_listen	3.0	None	
snmp_trap_raise_std_trap	3.0	SNMP	Results depend on external SNMP data.
snmp_trap_receive	3.0	SNMP	Results depend on external SNMP data.
snmp_trap_register_im	3.0	SNMP	Results depend on external SNMP data.

Table 25-2 (cont'd) PSL feature portability

PSL Function	Agent Needed	OS Porting Risk Areas	Portability Comments
snmp_trap_send	3.0	SNMP	Results depend on external SNMP data.
snmp_walk	3.0	SNMP	Results depend on external SNMP data.
sort	3.0	None	
sprintf	3.0	Odd formats	The majority of <code>printf/sprintf</code> formats are supported on all platforms. However, some obscure formats are not supported by the C library on which the PSL function is based.
sqrt	3.0	Out-of-range	Mathematical overflow may give slight differences.
srandom	3.0	OS algorithm	Random numbers are generated slightly differently on different systems, and although they will always be random, they will clearly not always be identical across platforms. Usually this fact is good!
subset	3.0	None	
substr	3.0	None	
system	3.0	OS processes	Launching a child process is clearly OS-dependent.
tail	3.0	None	
tan	3.0	Out-of-range	Mathematical overflow may give slight differences.
tanh	3.0	Out-of-range	Mathematical overflow may give slight differences.
time	3.0	OS time	Results clearly depend on the current time. However, the format is identical across platforms.

Table 25-2 (cont'd) PSL feature portability

PSL Function	Agent Needed	OS Porting Risk Areas	Portability Comments
tmpnam	3.0	OS algorithm	Results depend on the algorithm used by the OS to generate names of temporary files, which differs across platforms due to differences in where temporary file directories are placed, and other reasons.
tolower	3.0	None	
toupper	3.0	None	
trace_psl_process	3.2	None	
trim	3.0	None	
union	3.0	None	
unique	3.0	None	
unlock	3.0	None	Locks are internal; not OS locks.
unset	3.1	None	
va_arg	3.1	None	
va_start	3.1	None	
write	3.0	OS file and OS processes	Writing file affects OS files, and writing down a <code>popen</code> channel affects the child process. Also, issues such as the <code>\r</code> and <code>\n</code> problem arise on Windows NT.
write_to_report	3.2	None	

KM Compatibility

Compatibility differs from portability in that it does not relate to the operating system. Compatibility refers to correct functioning within any environment, and many compatibility issues are unrelated to the operating system platform. The other main compatibility area is with PATROL releases. Some of these issues include compatibility of the Knowledge Module with multiple versions of the agent, different styles of console (NT versus UNIX), and console release versions.

26.1 Agent Versions and Backward Compatibility

PATROL core functionality changes with new features in each major release. This raises the issue of having the Knowledge Module deal with multiple agent releases. The possible solutions to handling multiple agent platforms include:

- Using a minimal PSL/KM functional subset to support all agent platforms
- Using new PSL/KM functionality and not supporting the older agent versions
- Trying to support both agent versions and still use new PSL/KM functionality

A number of techniques can aid in this process. Fortunately, backward compatibility of PSL and Knowledge Module features has been achieved over time, which simplifies the process of moving an existing Knowledge Module to a new agent release.

26.1.1 Backward Compatibility of PSL and KM Features

All PATROL releases aim to support backward compatibility of all existing PSL and Knowledge Module features. For example, the intention with PATROL 3.1 was that all PATROL 3.0

PSL and Knowledge Module features would still work without change on PATROL 3.1 agents. This would allow a 3.0 Knowledge Module to run unchanged on a 3.1 agent.

The reverse is naturally not the case. A 3.1 Knowledge Module will not work on a 3.0 agent if it uses newer 3.1 functionality. The new functionality is available only on the new agent.

A desirable outcome would be for a 3.0 agent to run a 3.1 Knowledge Module, but run it using only the 3.0 features in that KM, but this is not currently available in an elegant form. The 3.1 Knowledge Module will simply not run on the 3.0 agent, either because the Knowledge Module format is not recognized because of new attribute types, or because the PSL script will no longer compile due to use of newer PSL functions.

26.1.2 Exceptions to PSL and KM Backward Compatibility

Although each release of PATROL aims to move forward with PSL and KM functionality, this ideal is not always achieved. A few exceptions exist to the transparent forward migration of PSL and KM features. The following are some of these:

- **New keywords**—Each new release of PSL adds new PSL built-in keywords. For example, the PATROL 3.2 release adds “for” as a new style of loop (to complement the existing “foreach” loop). This means that any existing 3.1 PSL script that has used the identifier “for” as the name of the variable will now fail compilation. No solution to this problem is available, other than editing the KM for the new agent version to use a different variable name. PATROL does not support any feature where a PSL script can run in an older subset of the PSL functionality.
- **XPC**—The XPC feature to use C/C++ functions via PSL is available with 3.2 agents. A very similar feature called “RPC” was available with 3.1 agents and was the XPC predecessor. The RPC older style is not supported by 3.2 agents. The solution is to rename the “.rpc” servers to “.xpc” and change “requires” statements to the XPC style.
- **Bug fixes**—Minor bugs are fixed in each release, which can change the semantics of the language. This change can cause a problem if your PSL script wrongly relies on “bug” functionality. Usually it does not.

26.1.3 Determining the Agent Version

You can determine the version of the agent via PSL. The special PSL built-in macro variable named “/patrolVersion” is a string that looks like “Version 3.1.9” or something like that. This can be examined via PSL `get` and processed to generate the numeric version. This can then be used to make coding decisions. For example, the following code detects whether we are on a 3.1 agent:

```
ver = get("/patrolVersion"); # Get agent version: "Version 3.1.9"
ver = trim(ver, "\n\r\t VERSIONversion"); # Trim letters: "3.1.9"
ver = ntharg(ver, "1-2", ".", "."); # Get initial 2 numbers: "3.1"
```

26.1.4 Testing Agent Version Requirements

A good style is for a KM to check the core architecture for support of its basic requirements and gracefully degrade when these requirements are absent. The best way to do this is in the pre-discovery PSL script for the KM. A KM can detect whether the agent version is adequate and terminate the entire KM if the agent is not up-to-date enough. An example PSL script could look like:

```
# Pre-discovery script
ver = get("/patrolVersion"); # Get agent version: "Version 3.1.9"
ver = trim(ver, "\n\r\t VERSIONversion"); # Trim letters: "3.1.9"
ver = ntharg(ver, "1-2", ".", "."); # Get initial numbers: "3.1"
if (ver < 3.1) {
    # We have a 3.0 or earlier agent, so we cannot operate...
    # Print a message; we could also log an event here if needed.
    print("MYKM: This KM requires a 3.1 or later agent\n");
    print("MYKM: Found a ", ver, " agent; terminating KM.\n");
    set("active",0); # disable this KM on this agent.
}
```

For this check to be valuable, the pre-discovery script must use a minimal PSL functionality subset. In the above script, the pre-discovery script should not use any new 3.1 PSL functions, because they would cause the script to ungracefully fail compilation on a 3.0 agent. Fortunately, pre-discovery scripts are usually short and to the point, and coding a minimal subset within this one script is possible, and does not prevent us from using newer features in other parts of the Knowledge Module.

Another issue might be the need for particular parameters or menu commands to disable themselves if their agent version requirements are not met. Unfortunately, this is difficult to do because the menu command or parameter will probably fail with a compilation error on an older agent version. Methods of avoiding this are available, but none of them are particularly elegant.

26.1.5 Coding for Backward Compatibility with Multiple Agent Versions

Since no built-in support allows graceful handling of multiple agent versions, the KM developer must add it via PSL. The simplest way to ensure that PSL will run correctly on multiple PATROL agent versions is to use the oldest version of PSL with a minimal subset. For example, coding in the subset of v3.1 PSL functionality should allow the PSL to run on PATROL 3.2 because of backward compatibility. However, doing so loses the possibility of using the more advanced features of PATROL to make the Knowledge Module more valuable.

The other, more difficult approach is to try to code around differences in the PSL language. Assuming that we have already computed the “ver” variable from the “/patrolVersion” symbol table entry, the code to use new features only on new agents would look something like this:

```
# NOTE: This method does not always work!
if (ver >= 3.1) {
    # We have a 3.1 or later agent... Use 3.1 new features..
}
```

The main problem with this approach of coding for multiple versions is the early failure of PSL compilation on unknown functions and features. For example, using the 3.1 PSL “annotate” function on a 3.0 agent means that the PSL code will fail with a compilation error because of the unknown function “annotate,” which is not known in v3.0 PSL. For example, the following code does not solve the problem because it will never run on a 3.0 older agent:

```
# Nice try but no cigar!
if (ver >= 3.1) {
    # We have a 3.1 or later agent... Use 3.1 new features..
    annotate(param, text);    # FAILS! Compile error on 3.0 agent
}
```

So even though you can check at runtime via “/patrolVersion” for the correct PSL version, your PSL program does not ever actually get to run, and the runtime check is useless. The compilation failure cuts short the whole idea on the older PSL version. This problem affects the use of new PSL built-in functions and also the extra arguments added to already existing PSL built-in functions.

Compilation errors would seem to put an end to the whole runtime decision approach. However, you have a couple of ways around these PSL compilation errors by tricking the PSL compiler:

- Compatibility PSL libraries containing stubs for the new PSL functions
- Using PSL script execution within a script via either “%PSL” or “PslExecute”

Both these methods defer the checking of the agent version number to runtime. By using the “/patrolVersion” variable, the PSL script can check whether to run a PSL script, and how to gracefully revert to the limited performance on the older agent.

26.1.6 PSL Library Stubs for Backward Compatibility

PSL libraries offer the first solution to preventing compilation errors. If a 3.0 agent does not know about the “annotate” function, build a PSL library with a function called “annotate.” The 3.0 PSL compiler is not going to know the difference. The PSL script looks like a normal 3.1 script, with no special code to even test whether the correct PSL version is used:

```
requires compat; # Load “compat” library containing “annotate”
....
annotate(".", "Some annotations");    # Use 3.1 function as usual
```

The compatibility library can simply be a stub for the function on a PATROL 3.0 compiler. It could even do something useful such as logging the annotation in a file. Multiple stubs for built-in functions can appear in the same library. However, here is the stub code just for `annotate`:

```
# Compatibility library for 3.1 built-in functions
export function annotate;

function annotate(param, annotation)
{
    # Do nothing
}
```

This library is loaded on the 3.0 version of the KM. On the 3.1 version of the KM, a different library must be used without definitions of these built-in functions, because of clashes with the 3.1 built-in names.

Clearly, you have a few limitations with this stub PSL library approach. It does not work for the case of extra parameters added to an existing PSL function in a new PSL release. It also does not allow you to use the built-in functions with different numbers of arguments, because the stub can have only a fixed number of arguments.

26.1.7 Using %PSL for Backward Compatibility

The second approach to avoiding compilation errors is the use of PSL execution from another PSL script. PSL scripts can compile and run other PSL scripts two ways. The obvious way is to use `PSL execute ("PSL", ...)`, but this does not work and fails with a runtime error setting the `errno` variable. The two successful methods are the use of the `PslExecute` function or a tricky use of the `%PSL` built-in agent command. Both of these methods work by hiding the PSL text that is to be executed from the compiler. Instead of having the PSL text in the main program, it is actually in string constants that bypass the compiler. The script must decide at runtime whether it can use the feature.

For example, the PSL profiler menu commands in the KM Toolkit make use of 3.2 PSL functions. In order to have graceful failures on 3.1 agents, you want the PSL script to first check `/patrolVersion` and then print an error message such as “The PSL Profiler is not available on 3.1 agents.” The problem is that the 3.2 PSL functions `ProfOptions`, `ProfDefaultOptions`, `ProfGet`, and `ProfTop` will cause compilation errors on 3.1 agents. The solution is to use PSL `system` to execute a `%PSL` command. The example text looks like:

```
if (use_32_functionality) {
    system("%PSL set(\"/prof_def_options\",
        ProfDefaultOptions());");
    sleep(1);    # Ensure agent has time to run the new PSL process
    ret = get("/prof_def_options");
}
```

```

    }
    else {
        print("The PSL Profiler is not available on PATROL 3.1\n");
        exit;
    }

```

This code assumes that the variable “use_32_functionality” has been computed earlier by analysis of the “/patrolVersion” agent macro at the top of the script. This basic idea works well in the 3.2 PSL profiler menu commands to produce graceful error messages on 3.1 and earlier agents.

This example code shows some of the pitfalls of having to use a separate PSL script to overcome PSL compilation warnings. The sequence aims to replace the simple 3.2 PSL code:

```

if (use_32_functionality) {
    ret = ProfDefaultOptions();
}

```

Clearly, the code that compiles on both 3.1 and 3.2 agents is harder to read and maintain. The call to PSL `sleep` is necessary because you have no guarantee of scheduling of the other script. Another better method of synchronization of the current and child PSL scripts would be to use PSL `lock` and `unlock` functions. Another problem is that you cannot easily have a PSL script return a value, and so the need for PSL `set` and `get` for handling the result of the call to `ProfDefaultOptions`. Another problem is that the script is running in a completely different PSL process that affects issues such as the symbol table context of PSL `get/set`, and per-process issues that will fail (e.g., PSL `getpid` and `getpname` functions). We could try to make this method a little less inelegant by wrapping some of the ugliness in a PSL library:

```

function exec_psl(expression)
{
    tempvar = "/some_temporary_variable";
    system("%PSL set(\"\" . tempvar . "\", \" . expression . \");");
    sleep(1); # Ensure agent has time to run the new PSL process
    ret = get(tempvar);
}

```

Even this code is not really fully satisfactory. More than one temporary variable name should be chosen dynamically to avoid problems if the library is called by multiple PSL processes. The script should also use PSL `lock` and `unlock` functions rather than the `sleep` function.

26.1.8 Using PsIExecute for Backward Compatibility

The alternative to using PSL `system` with `%PSL` is to use the built-in function called `PsIExecute` added in PATROL version 3.1 agents. This PSL function offers the equivalent of PSL

`execute("PSL", ...)` to run child PSL scripts. This function can also be used to avoid compilation errors and allow PSL scripts to run on multiple agent versions. The issues with its usage are basically identical to the `%PSL` method but the method, is not compatible with 3.0 agents, only with 3.1 and later agents.

26.1.9 Different KM for Each Agent Version

The final approach in handling KMs that run on multiple agent versions is to give up trying to support both versions with one KM package. This means to offer completely different KMs for different PATROL versions.

For example, if you are writing an Internet Explorer monitoring KM for Windows NT, you could have two KM files called `IE_30` and `IE_31` to work on 3.0 and 3.1 agents, respectively. Each KM could check its own agent version requirements via `/patrolVersion` and disable itself on the wrong versions.

This approach means a totally different KM file for the different versions, and usually different PSL scripts (unless some are using only the older functionality).

This solution is unfortunate because it adds to configuration complexity by introducing multiple KM packages. However, it may become necessary in order to offer a KM that works on all versions. On the other hand, it is also advanced in its use of new PSL features.

26.2 PATROL Console Compatibility Issues

Two issues arise with the form of PATROL console used to examine the KM. The first is the operating system platform on which the console executes, which affects what console-side operating system facilities are available to the KM. The second is the release of the console, which affects the functionality offered by the console as it pertains to the KM.

26.2.1 Console Dependencies: UNIX and Windows NT Consoles

The two main platforms for the PATROL console are UNIX and Windows NT. The X/Motif GUI console runs on UNIX machines, and the Windows NT GUI console runs on Windows NT machines. The functionality offered by these consoles is largely similar. However, a few differences occur in particular areas of console-side functionality that affect the KM.

- **PSL response auto-sizing**—The Windows NT console offers auto-sizing of PSL response GUI dialogs. This avoids the need for the `"w="` and `"h="` preferences that are commonly needed for UNIX dialogs via `PSL response`. However, relying on this auto-sizing means that the menu GUI dialogs will not operate adequately on the UNIX console.
- **Other PSL response "features"**—Some PSL scripts have come to rely on "undocumented" features of the `PSL response` function that occur only because of its interaction with the X window manager. For example, text fields can be elongated by adding a dummy all spaces text label above them, but this does not work on the Windows NT console response dialogs.

The other aspect of console operating system dependence is the external environment. Although much of the console's GUI functionality is internal, a KM can accidentally rely on a UNIX feature. Some particular pitfalls include:

- **xterm**—One common method of launching a server-side executable command-line loop (e.g., `sqlplus`) is to launch an “xterm” to display the window. This will work only for a UNIX console. It may work for a Windows NT console with an installed X emulator, but you do not want to rely on this situation.
- **DISPLAY environment variable**—The special “DISPLAY” variable is also X-specific. Any attempt to use the `DISPLAY` variable to show a window on the console is relying on a UNIX X server, and will likely fail on a Windows NT console.

26.2.2 Console Release Version Dependencies

Another aspect of console dependence is the version of the console. However, this is not usually a problem, because sites commonly have the most recent console, even if the most recent agents are not everywhere. In fact, PATROL's console-agent backward compatibility plans explicitly expect that all consoles are updated, and a new agent need not talk to an older console. Therefore, if a KM functions on a newer agent, then the newer console should exist. Nevertheless, some areas depend on the console version number:

- **PSL response dynamic updating**—The “D=1” preference for the PSL response function requires a PATROL 3.1 console or later.
- **Newer response elements**—Some of the newer PSL response elements numbered above 20, such as the ticker, require a 3.1 console or later.
- **Local menu commands** via “`##MODES% local`” require a 3.2 console.
- **Developer-only menu commands** via “`##MODES% dev`” require a 3.2 console.
- **Stoplight and Boolean parameters** require a 3.2 console (and 3.2 agent).
- **PSL desktop function** requires a 3.2 console (and 3.2 agent).

26.3 Compatibility Table of KM and PSL Features

Various different PATROL features have been added in each core release of agent and console technology. This means that the PSL and Knowledge Module features were introduced in a particular release and require at least that version of agent. Table 26-1 shows the releases that are required for every basic built-in non-PSL command; Table 26-2 shows PSL feature agent needs.

Table 26-1 PSL feature compatibility

Built-in Command	Agent Needed	Purpose
%PSLPS	3.0	List of PSL processes
%PSL	3.0	Run a PSL script
%DUMP CHANNELS	3.0	List of global channels
%DUMP LIBRARIES	3.2	List of PSL libraries
%DUMP ALL	3.0	Report all agent internals
%DUMP GLOBALS	3.0	Report global variables
%DUMP APPS	3.0	Report application classes
%DUMP APP_INSTS	3.0	Report application instances
%DUMP PARAMS	3.0	Report parameters
%DUMP RUNQ	3.0	Report agent run-queue
%DUMP TASKS	3.0	Report task commands
%DUMP RTLIST	3.0	Report agent run list
%DUMP ERRORS	3.0	Report agent errors
%DUMP KM_LIST	3.0	Report list of KMs loaded
%DUMP CONSOLES	3.0	Report connected consoles
%CTRL_ALT_DEL	3.1	Restart agent; developer only
%RESET_APPL_FILTER	3.1	Reset application filter list

Table 26-1 (cont'd) PSL feature compatibility

Built-in Command	Agent Needed	Purpose
%RESET	3.0	Reset something
%ECHO	3.0	Echo text or macro
%SNOOZE	Withdrawn	Snooze agent alarm (v2.0 only)
%SET	3.0	Set agent tuning variable
%REFRESH_APPL_DISC	3.0	Refresh application discovery
%REFRESH_PARAMS	3.0	Refresh parameters of instance
%REFRESH_ALL_PARAMS	3.0	Refresh all parameters of agent
%REFRESH_PROC_CACHE	3.0	Refresh process cache

PSL language features are another area of compatibility issues with agent releases. Table 26-2 shows the agent version needed for each PSL feature.

Table 26-2 PSL language feature compatibility

PSL Language Feature	Agent Needed
if, else statement	3.0
foreach loop	3.0
while loop	3.0
switch statement	3.0
arithmetic operators (+, -, etc)	3.0
inc/dec operators (++ , --)	3.0

Table 26-2 (cont'd) PSL language feature compatibility

PSL Language Feature	Agent Needed
string concat “.”	3.0
shift operators (<<, >>)	3.0
bitwise operators (, &, ^)	3.0
regexp operators (!~, =~)	3.0
logical operators (!, &&,)	3.0
relational operators (<, >, etc)	3.0
errno variable	3.0
exit_status variable	3.0
PslDebug variable	3.0
exit statement	3.0
export statement	3.0
requires statement	3.0
function statement	3.0
return statement	3.0
local declaration statement	3.0
last statement	3.0
next statement	3.0
for loop statement	3.2

Table 26-2 (cont'd) PSL language feature compatibility

PSL Language Feature	Agent Needed
ternary operator (?:)	3.2
ellipsis (. . .) varargs	3.1
pragma statement	3.2
do-until loop statement	3.2

PSL is only one part of a Knowledge Module. The other declarative features and attributes also have specific agent version needs. Table 26-3 shows these KM features and the agent version needed.

Table 26-3 KM feature compatibility

KM Feature	Core Version
Gauges parameter	3.0
Graph parameter	3.0
Text parameter	3.0
Stop light parameter	3.2
Boolean parameter	3.2
“status” flag of a parameter writable	3.1
set (“active”, 0) disables parameter	3.0
set (“active”, 1) re-enables parameter	3.1
set (“active”, 0) turns off application	3.0

Table 26-3 (cont'd) KM feature compatibility

KM Feature	Core Version
<code>set("active",1)</code> wakes disabled application	Not yet!
Menu commands	3.0
Info box commands	3.0
Pre-discovery	3.0
Discovery	3.0
Annotations	3.1
HTML online help	3.2
PSL libraries	3.0
PSL binaries	3.0
KML files	3.2
KM files	3.0

26.4 PATROL 3.1 KM/PSL Added Features

The following list shows all features of PSL and Knowledge Module attributes that were added in the PATROL 3.1 core release:

- PSL response dynamic D=1 preference added
- PSL `response_get_value` function added to support the D=1 response method
- `annotate` function
- `annotate_get` function
- `event_archive`, `event_catalog_get`, `event_range_manage`, `event_report`, and `event_schedule`
- `destroy_lock` function
- `encrypt` function
- `getpid` and `getpname` functions

- `history_get_retention` function
- `kill` function
- `PslExecute` function
- `refresh_parameters` function
- `remote_close`, `remote_event_query`, `remote_event_trigger`, `remote_file_send`, `remote_open` functions
- ellipsis “...” token for variable argument functions
- `va_start` and `va_arg` functions for variable argument user-defined functions
- `blackout` function
- `convert_date` function
- `pconfig` function
- `unset` function
- PSL `grep` “t” and “i” flags options added to ignore case and do text (non-regular expression) search

26.5 PATROL 3.2 KM/PSL Added Features

The PATROL 3.2 core release also added a number of PSL and Knowledge Module features. PSL Features added in PATROL version 3.2 include:

- PSL profiler functions added—`ProfOptions`, `ProfGet`, `ProfTop`, and `ProfDefaultOptions`
- PSL optimizer added to the agent and standalone compiler
- PSL `pragma` statement added for optimizer control
- PSL built-in constants added, such as `R_CHECK_BOX` for response and `R_CHART_LOAD` for chart, etc.
- PSL `for` loop added
- PSL `do-until` loop added
- PSL ternary operator added
- `set_alarm_ranges` function
- `getpname` function
- `desktop` function
- `write_to_report` function
- `popup_report` function
- `join` function
- `replace` function
- `chart` function
- `get_ranges` PSL function second argument added to report the state (i.e., OK/WARN/ALARM)
- `grep` options “b” (bailout on first match), “i” (ignore case), and “c” (count) added

Knowledge Module attribute features added in PATROL version 3.2 include:

- KM list “.kml” file support added to the PATROL Console
- Signal or traffic light parameter type added
- Boolean parameter type added
- Per-OS platform automatic supported added to KM loading
- `ExtraFilesList` added to KMs for “Commit KM” support of PSL libraries
- `%DUMP LIBRARIES` built-in command added
- Warnings about out-of-date PSL libraries generated when a PSL script is compiled
- XPC support of C/C++ library linking added
- Local menu commands via “`##MODES% local`”
- Developer-only menu commands via “`##MODES% dev`”

Advanced KM Architectures

The basic architecture of a Knowledge Module is to use a single agent monitoring the application on the local host. This single agent architecture is the basis of most of the PATROL Knowledge Modules. However, not all architectures follow this simple scheme of things. Here is a list of applications for which the Knowledge Modules are exceptions to this single agent architecture:

- Oracle Parallel Server
- SAP R/3
- Baan Applications
- NetWare
- AS/400

Not all these Knowledge Modules have the same architecture. Actually, two basic types of exceptions to the single agent architecture exist:

- Multi-agent distributed architecture
- Proxy architecture

The multi-agent architecture arises in response to distributed applications, and combines information collected from multiple hosts to centralize the logical representation. The proxy architecture arises for a different reason, which is typically the lack of a native PATROL agent on the hardware or operating system platform.

27.1 Multiple Agent Architectures

The majority of Knowledge Modules require only a single agent. The server-based architecture of most databases leads to this trend. The agent on the machine takes care of all the Knowledge Module issues for that database server. No inter-agent communications issues arise for these Knowledge Modules.

However, a number of applications are more distributed in nature. Some of the application types that lead to the need for multi-agent support in the Knowledge Module include:

- Distributed applications such as SAP R/3 and Baan, where the application itself is implemented across multiple machines.
- Clustering architectures where a set of host machines is logically grouped with shared resources for failover and high availability systems.

These applications lead to the need for the next level of multi-agent Knowledge Modules. These Knowledge Modules require data from multiple agents to gather all of the monitoring status for the entire application. These Knowledge Modules need to make use of one of the inter-agent communication methods to synchronize their data.

27.1.1 Inter-Agent Communication Technology

Several alternative technologies are available for communication between distributed PATROL agents. The evolution of the PATROL architecture has seen a number of new inter-agent technologies added, particularly in the 3.1 agent release.

- **PATROL API**—The PATROL API, specifically the PATROL Event Manager (PEM) API, can be used by one agent to access another agent's data or trigger events in the other agent. This is the technology used in the SAP R/3 Knowledge Module to consolidate data from multiple agents. The PATROL API has existed since PATROL 3.0, but was significantly enhanced in the 3.1 release.
- **PSL remote functions**—A number of PSL builtin functions are specifically for inter-agent access. The capabilities include remote event management, event triggering, and remote PSL script execution. This technology is used by the History Loader to implement its multi-agent Knowledge Module architecture. These functions were mostly added in the PATROL 3.1 agent release.
- **PatrolCli executable**—Based upon the PATROL API, this is an executable that can be launched to perform many of the actions possible within the API. In a sense, it is a client executable that speaks PATROL API. In some cases, launching the `PatrolCli` executable from a Knowledge Module can be easier than building your own executable using the C APIs from the PATROL API. However, in this case, to use the PSL remote functions is usually more efficient.

- **pconfig executable**—This is somewhat obscure technology, but it has been used in PATROL 3.0 for inter-agent access. With the advent of PATROL 3.1 with the API and PSL-based access, this usage is not recommended and is more of historical interest.
- **SNMP**—Since you have full SNMP support in PSL, including trap sending and SNMP set and get features, you can use SNMP launched from PSL in one agent to communicate with another agent. This feature can be used in PATROL 3.0, but has been superseded by the API and PSL features of PATROL 3.1 and later releases.
- **Operating system facilities**—The Knowledge Module can certainly launch operating system clients for remote access. For example, PSL can launch facilities such as `ftp`, `telnet`, `rcp`, `rsh`, and so on. These should not be needed except in PATROL 3.0, where the API and PSL remote access facilities were not available. This idea was used by the early versions of the History Loader for 3.0, but replaced by PSL remote access features in the 3.1 release.

Choosing an inter-agent communication method is an important part of designing a multi-agent Knowledge Module. This can be a major part of the development effort. Configuration issues are an important part of the overall inter-agent communication mechanism.

27.1.2 Configuration Methods

Configuration is an underlying requirement for multi-agent architectures. The inter-agent technologies generally require that the hostname, port number, and a username/password pair be supplied. Each agent in a multi-agent architecture may need some configuration information (see Table 27-1). Alternatively, a single central agent may require configuration information for each of the remote agents so as to access each agent's information.

Table 27-1 Inter-agent configuration needs

Inter-Agent Technology	Hostname Required?	Username/ Password Required?	Port Number Required?
PATROL/PEM API	Yes	Yes	Yes
PSL remote functions	Yes	Yes	Yes
PatrolCli executable	Yes	Yes	Yes
pconfig executable	Yes	No	Yes
SNMP	Yes	No	No

Table 27-1 (cont'd) Inter-agent configuration needs

Inter-Agent Technology	Hostname Required?	Username/ Password Required?	Port Number Required?
telnet	Yes	Yes	No
ftp	Yes	Yes	No

These information requirements impose configuration requirements on the multi-agent Knowledge Modules. Each agent has to be configured to have this information for each of the other agents with which it must communicate. Generally, this requirement means that the user must be asked for this information during setup for the multi-agent Knowledge Module. You cannot get a list of hostnames or other connection configuration data either from the agent or from a connected console. Similarly, username and password configuration items are not inherited or accessible from the console or from other agents for security reasons.

The Knowledge Module has to take care of these configuration issues just as a single-agent Knowledge Module takes care of username and password issues for connecting to a server. Minor tricks involve helping the user find the correct information. For example, on UNIX the hosts can be found by reading either the list of hosts from the file `/etc/hosts` or the output of executing `ypcat hosts` for NIS or `niscat hosts` for NIS+.

27.1.3 Centralizing Architectures: History Loader Example

A simple form of multi-agent Knowledge Module architecture is the “centralizing” architecture. In this architecture, all agents are feeding back their results to one central agent that is combining or using this data from multiple agents. The History Loader is one example where the remote agents propagate data back to a central agent.

The History Loader is the best example of the use of a centralizing architecture with Knowledge Module technology. In the History Loader, each agent gathers parameter history data for the current agent, and then forwards this history data back to a central agent. This central agent is then archiving the data from multiple agents into a central database (supporting a variety of database formats). This task involves a “push” model, since each independent agent is pushing the results back to the central agent, rather than a “pull” model, where the central agent would pull the data from each remote agent.

The remote access technology used by the History Loader is PSL remote access. Specifically, the main feature used by the History Loader is the `remote_file_send` function to send history data files to the central agent.

The configuration requirements of the History Loader are typical of centralizing architectures. Each agent needs to know a valid hostname, port number, username, and password for the

central machine. These allow the PSL remote functions to connect to the central agent. Unfortunately, you have no simple way to propagate this configuration information out to each agent, and each agent must be individually configured with the connection information. This configuration is necessary before it can begin to send history data back to the central agent.

27.1.4 Multi-Machine Applications: SAP R/3 KM Example

The monitoring of applications that are implemented across multiple machines imposes the need for inter-agent communication. The SAP R/3 Knowledge Module is the best real-life example of this type of Knowledge Module.

The technology used by the R/3 Knowledge Module for inter-agent communication is the PATROL API wrapped into a child executable called “`patcol`.” For PATROL 3.1 and earlier versions of the R/3 Knowledge Module, this process is a child process launched by a PSL `popen` command. For PATROL 3.2, the API is used in an XPC server executable created using the XPC C/C++ Knowledge Module linkage technology. The difference between XPC and `popen` is mainly in how the agent communicates to the local executable. The method used by the local server to gather data from remote agents via the PATROL API is the same.

The configuration requirements of the PATROL API are valid hostnames, port numbers, and username/password account information. The central collector must be configured to know all this information in order to use the PATROL API to get the information from remote agents.

The R/3 Knowledge Module also makes use of the special type of nested instances, called main map instances, to present a logical application view of the R/3 instances, rather than the usual host-based physical view. This improves the view presented in the PATROL Console. This view is achieved by special arguments to the PSL `create` function. For more discussion, see Chapter 29.

27.1.5 Clustering Architectures

A clustering architecture occurs when multiple machines share a resource, such as a hard disk. For the Knowledge Module Developer, this leads to some problems involving multi-agent synchronization. The problem is that each machine can have an agent running on it with the same Knowledge Module. Unless something is done, each agent will monitor the cluster disk drive, measure the same statistics, and report the same alarms. This approach is clearly wasteful of CPU resources to duplicate the monitoring of the shared device.

The solution to this problem is typical of all multi-agent Knowledge Module architectures. Any of the inter-agent technologies can be used to synchronize the agents, so that they can decide on the main agent that will perform the monitoring. One extra alternative available to the Knowledge Module developer is to use the shared device as an easy means of inter-agent communication. For example, the shared disk can be used by storing files on the disk, which can be read by all agents. If locking is supported within the cluster, each agent can try to lock a particular file, and when only one succeeds, this single agent is the one that does the monitoring.

27.1.6 Building New Inter-Agent Methods

An alternative to using the variety of PATROL inter-agent methods is to build your own. The basic idea is to build your own client-server layer over the top of the PATROL layer. Each agent can launch the new executable process that handles the inter-agent access. This capability means that the new executables have to take care of all of the network-related issues. For example, this requires the executable subagent to deal with UDP/TCP protocols, encryption, and any other security-related issues.

This use of your own inter-agent communication method can be a viable, if non-trivial, method of removing the need for configuration issues such as the usernames and passwords of a valid account on the remote machine. However, this turns over the responsibility for security validation to this new inter-agent subagent layer.

The connection of the PATROL agent to the local subagent is not so difficult. This can be implemented via one of the normal methods of communication with a subagent process, such as PSL popen or an XPC server executable.

27.2 Proxy Architectures

A proxy architecture occurs when one entity acts as a front for another. In the case of a Knowledge Module proxy architecture, one agent runs a Knowledge Module locally to represent the elements of another remote machine. The idea of using a proxy architecture arises mainly in connection with monitoring a machine that does not have a full PATROL agent on it. For example, this applies to monitoring platforms for which no PATROL agent is ported yet. Proxying involves using an agent on one machine to remotely monitor the system or application status without actually having an agent on the machine.

PATROL already has a number of successful examples of the use of the proxy Knowledge Module architecture. Two examples that rely on a proxy architecture that we will examine are:

- NetWare Knowledge Module—developed by BMC Software
- AS/400 Knowledge Module—developed by Proxima Technology

Both of these Knowledge Modules are examples of proxying as a solution for a platform without a native PATROL agent. However, these two Knowledge Modules are somewhat different in their use of the proxy architecture. The NetWare Knowledge Module is a proxy architecture involving a remote subagent on the monitored server. The AS/400 Knowledge Module has no remote software component and is therefore a pure proxy architecture without any remote presence.

Many of the major database monitoring Knowledge Modules have a proxying component, although their primary functionality is a non-proxy local architecture. A number of the database Knowledge Modules support remote monitoring of databases via the database vendor-supplied access methods. For example, the Oracle Knowledge Module can perform remote monitoring via SQL*Net to monitor Oracle instances even without an agent on the remote machine.

Although this remote Oracle monitoring is necessarily more limited than the normal non-proxy situation, it is a good solution for some Oracle-supported machine types where PATROL does not offer an agent. Other Knowledge Modules such as those for Informix, DB2, and Internet Servers have remote monitoring capabilities.

27.2.1 Proxying and the PATROL Architecture

The use of a proxy Knowledge Module architecture works well with the PATROL architecture. Almost all the advantages of the PATROL architectural infrastructure are automatically retained without additional Knowledge Module development effort.

Integration with all the various views of PATROL objects is smooth. The console or integration component talks directly to the agent that is running the proxy Knowledge Module. So the correct discovered objects and parameter metrics are automatically handled for a proxy Knowledge Module for the following views:

- PATROL Console—Operator or Developer, UNIX or Windows NT
- PATROLVIEW suite of integration components for enterprise framework consoles and help desks
- PATROLWATCH for Web Browsers—Netscape Navigator and Microsoft Internet Explorer web browser integration
- PATROL Command-Line Interface—a text-based console-like component
- PATROL Agent API—the published API used to access agent objects and parameters

As usual, the multi-console support for the agent is permitted. This means that all or any combination of the above views is possible on a single agent, in this case the proxying agent, without limitation or interference between the console views.

All the features of the agent apply even when it is acting as a proxy agent. Parameter history and event record storage and archival is unchanged and automatic. Notification via SNMP, pager, email, or through integration with a third-party notification product is also unchanged. The console views including graphs, multi-line charts, the Agent Query Tool, and both iconic and outline views are all available transparently without special Knowledge Module development effort.

The features of a Knowledge Module are also largely unchanged for a proxy Knowledge Module. As with any Knowledge Module, the result depends on the effort used to exploit these features, but they are all available. Features in this category include recovery actions, menu commands, info box commands, parameters, alarm range thresholds, online help, global channels, and annotated data points. The only difference here is that the underlying implementation of these features may involve remote access via the chosen proxy access method.

A proxy agent is not limited to this purpose. A PATROL agent can run both proxy and non-proxy Knowledge Modules at the same time. To the agent, both these classes of Knowledge Modules are identical, because the Knowledge Module developer has to add the proxy logic into

how the Knowledge Module operates. For this reason, a proxy Knowledge Module can run on an agent that is already monitoring a local system. Alternatively, the proxy Knowledge Module could run on an agent on the same machine as the console, if the console is on a separate machine.

27.2.2 Proxying Design Choices

You have two basic forms of a proxy Knowledge Module architecture. The first is a “pure” proxy architecture, where no remote executable is on the monitored system. The second is a hybrid proxy architecture, where a form of local subagent is used on the remote machine, but it is not a full PATROL agent. Both these options are viable, and the choice depends on the extra value that a subagent can provide. The AS/400 Knowledge Module is an example of the pure proxy approach, whereas the NetWare Knowledge Module uses a small subagent presence on the remote machine. First, examine the subagent-less architecture that forms a pure proxy approach.

A pure proxy architecture without any remote subagent relies on the presence of some non-PATROL method of remotely accessing the system. This service is usually part of the operating system or supplied by the monitored application itself. Some of the common methods that are candidates include:

1. Operating system daemons—ftp, ping, finger, etc.
2. Remote login capabilities—telnet, rlogin, rsh, rcp, etc.
3. Web-based HTTP access
4. SNMP access—get, set, and traps
5. Vendor-supplied remote access methods—e.g., SQL*Net for Oracle

These access methods all rely on some service or daemon that is already there. Therefore, the Knowledge Module development can be a one-sided process of accessing these existing daemons via an existing front-end or existing API. Because the remote service is already supported, no extra effort is required to build a special subagent to handle the proxy Knowledge Module requests. This reduced development impact is the advantage of using no subagent, provided that the capabilities for remote access via operating system or vendor methods are adequate for the Knowledge Module requirements.

27.2.3 AS/400 Knowledge Module Proxy Architecture

The AS/400 Knowledge Module is a Knowledge Module for monitoring the AS/400 machine that has been developed by Proxima Technology (<http://www.proxima.co.uk>), a long-standing BMC Software preferred partner. This Knowledge Module extends the power of PATROL into the AS/400 platform by providing monitoring of AS/400 operating system metrics, messages, and subsystems. The AS/400 KM permits monitoring of AS/400 machines despite the absence of a PATROL agent for the OS/400 operating system. Clearly, a proxy architecture is involved.

The AS/400 Knowledge Module has a proxy architecture using a proxy Knowledge Module running on a UNIX agent. No remote subagent is running on the AS/400 machine. The remote access method used by the AS/400 Knowledge Module is the wrapping of `telnet` to launch remote commands. Support for `tn5250` is under consideration for a future version of the Knowledge Module. The Knowledge Module gathers statistical data from OS/400 via a terminal network session based around scripted launching of a `telnet` client on the UNIX machine.

The use of shared global channels ensures that only one such terminal session is required for each monitored machine, and each class of AS/400 statistics shares the `telnet` session for performance reasons. Multiple AS/400 machines can be monitored from a single UNIX machine running the proxy Knowledge Module. Each monitored AS/400 machine involves an additional shared channel. In other words, an additional `telnet` session from the UNIX agent appears for each monitored AS/400 machine. This means a slight load per monitored AS/400 machine, but each UNIX machine was shown in tests to be capable of monitoring 20-30 AS/400 machines.

The performance of the AS/400 Knowledge Module is enhanced by its proxy architecture. By off-loading computational processing to the proxy UNIX machine, the load on the AS/400 machine is negligible. The main cost of monitoring the AS/400 is the CPU cost on the proxy UNIX machine. This is much cheaper capacity than the use of CPU on the AS/400 machine. This saving is one of the main advantages of a subagent-less proxy implementation as used by the AS/400 Knowledge Module.

27.2.4 Proxy Subagent Architecture

Another form of proxy architecture is the use of a subagent process on the remote machine. This involves specifically developing a subagent process to run on the remote machine that communicates with the proxying agent. This is similar to having a `PATROL` agent on the remote machine, except that the subagent need not be as sophisticated. The use of a subagent can be advantageous if the default remote access services are not adequate, since a subagent can usually provide additional features. Certainly this is the case in the NetWare Knowledge Module, where the subagent is based on the technology from the NetTUNE PRO performance tuning product.

The main downside to a proxy subagent compared to a pure proxy approach is the additional development effort for the subagent component. This is a non-trivial development effort, since the subagent and the proxying agent must be able to communicate over the network in some manner. Therefore, both the remote subagent and a local client to be launched by the local proxying agent need to be instrumented with communications code. Now examine how this need was addressed using the `PATROL` API as a communication layer in the NetWare Knowledge Module.

27.2.5 NetWare Knowledge Module Proxy Architecture

The NetWare Knowledge Module from BMC Software offers professional monitoring and management of NetWare servers. The NetWare Knowledge Module is implemented via a proxy Knowledge Module architecture. The NetWare Knowledge Module is executed on a Windows

NT or UNIX agent that acts as a proxy for the NetWare server. The subagent on the NetWare server is based on the BMC Software product called NetTUNE PRO. The communication method between the proxy agent running the NetWare Knowledge Module and the NetWare server's subagent is the PATROL API. Events are sent via the PATROL API in a two-way direction between the PATROL agent and the subagent.

The subagent running on the NetWare server is actually a combination of the PATROL API and an interface into the NetTUNE PRO engine. This is implemented as the PEMNETA NLM and works in conjunction with the NetTUNE PRO NLMs and NCF files. The use of the NetTUNE PRO agent allows powerful low-level technology to be employed for measuring and tuning server performance.

Multiple NetWare servers can be monitored by a single proxy agent. So a single agent could be used to monitor many NetWare servers. This proxy agent could be an already existing agent on another machine or could be on the console machine.

Performance impact on the NetWare server is negligible due to the proxy architecture. The NetTUNE PRO agent imposes minimal load on the server, and design features of the Knowledge Module ensure low network traffic. The parameter thresholds from the NetWare Knowledge Module are downloaded from the proxy agent to the NetTUNE PRO subagent on initial startup of the proxy NetWare Knowledge Module. Because data values are sent over the network, the collection periods when data is sent from the subagent to the proxy agent are reduced for performance reasons, unless the NetTUNE PRO agent detects an anomaly such as an exceeded threshold. In this way, both CPU and network resource impact is minimized.

27.2.6 Nested Instance Display of Proxy Icons

The PATROL data model does not have ideal support for the proxy idiom, at least not in PATROL 3.2 or earlier releases. However, a neat solution solves the problem of making a proxy solution visually appealing and consistent with local monitoring views. The main issue is that the discovered objects will reside in the machine object space of the proxying agent. So with the basic discovery via the PSL `create` function, the icons for the remote system or the remote applications will appear underneath the computer icon for the proxy agent. No automatically generated machine icon for the remotely monitored machine will appear.

Visually, the result is that one has to drill down into the proxy agent icon in order to see the proxy objects. This is clearly not a true representation of the host and application's physical or logical organization. Instead this is a representation of the monitoring architecture, not the monitored object's architecture.

Fortunately, the concept of "nested instances" can be used to solve the visual problem. Nested instances refers to the argument in PSL `create` that specifies a parent instance when creating a new instance. By careful choice of the parent instances for the proxy objects, you can go around the display limitations on the console.

The idea of "main map" icons in the PSL nested instances support can be used to advantage in a proxy architecture. Main map icons are the use of "/" as the special top-level instance.

The console display interprets this to mean “put the new instance on the map, where all the other computer icons are placed.” This use of PSL `create` with a “/” parent argument therefore makes the console put an instance icon in a place that it appears similar to a computer icon. This suggests the use of this method to create “fake” computer icons for the proxy machines. One solution is creating an application class, say “`PROXY_MACHINE`,” selecting a computer icon to represent it, and creating one instance of this class for every remotely proxied machine. Naturally, if you are proxying more than one application class on the remote machine, you should be sure to only create one of these `PROXY_MACHINE` instances.

Each proxy application instance can then specify this proxy machine object as its parent. On the console you will see a “`PROXY_MACHINE`” icon at the computer icon level, and then drill down into the computer-like icon to see all the proxy applications. Some of the visual cues will still exist that separate a computer icon from an instance icon. For example, on the `PROXY_MACHINE` icons, the built-in menus are different, as will be the built-in elements of the info box popup. However, you can certainly control the Knowledge Module-defined menus and info box items to make the rest of these items appear computer-like.

27.2.7 Design of Proxying

The simplest implementation of a proxy architecture occurs if a ready-made executable is available to perform the remote access. For example, you use the “`telnet`” client executable when using the `telnet` daemon as the remote server. The use of `telnet` becomes similar to launching any other executable on the local agent. Using `telnet` as an example, the basic steps to opening a remote session include:

1. Call PSL `popen` to launch the `telnet` executable (in interactive mode).
2. Use PSL `write` to send the command down the pipe to the `telnet` executable.
3. Use PSL `read` to read the results returned by the `telnet` executable.
4. Call PSL `close` to close the channel and thereby terminate the `telnet` process.

The next step is processing the results from the PSL `read` function call. The issues of results processing in proxy Knowledge Modules are not very different from the usual problems of interpreting output from any command, such as when using a global channel. Typically the output from the client, such as `telnet`, is not very “clean.” The output probably includes not only the command results, but also a welcome header, an input prompt prior to every command, and other noise. Extracting the true command output from this text is not always easy. PSL has no easy way to determine what strings are output from the command and what is text from other prompts.

Some clients have a silent mode to suppress prompts, which can be enabled to reduce the amount of output noise, but other clients do not. The welcome header is often different for different systems, making portable code difficult, and might be time-dependent. The task becomes a matter of determining what is the header end and what is the start of the command results.

Usually the first prompt occurs after the end of the header and before the command results, and this can be helpful in separating the text.

27.2.8 Synchronizing Multiple Commands

The example above is the simplest mechanism. Open the pipe to the remote machine via `telnet`, send the command, wait for the results, and then shut down. This is adequate for simple tasks, but is not adequate if a large number of commands are to be executed. In this case, you need to send multiple commands through one connection.

Sending multiple commands down a pipe is quite easy. Using PSL `write` for each command, you must ensure the correct command format and usually add a terminating newline. The remote client will typically process the commands one at a time in sequence.

Processing the results output is another story. Separating the output from each command is the main issue. No clean solution presents itself here. You cannot separate the PSL `read` calls based on commands because you have no guarantee that results will come back in separate blocks. In fact, this statement is particularly true of proxying architectures where the results will be buffered across the network. In some simple cases, you can distinguish which results come from which command on the basis of some textual pattern or prefix, but they are the exception rather than the rule. Another solution is necessary.

The most common solution to this problem is the use of some form of “separator” command to ensure a recognizable pattern between results. In the `telnet` example, it might be as simple as sending an “`echo SEPARATOR`” command down the pipe, to use the `echo` command to replay that text back between each real command. String processing of the PSL `read` return results can then break apart the queries.

The `echo` command could be even more intelligent by stating an identifier showing what command this starts or ends. This identifier might be some key, or might be a full copy of the one-line command.

An interesting point is that the separator idiom can be helpful to identify and remove the header lines of noise. By making the first command a separator command, you can then simply throw away the lines before the first separator.

27.2.9 Proxy Architecture Trade-Offs

The proxy architecture offers some performance trade-offs. The basic trade-off is increased network traffic from the monitored node traded off against reduced computation on the monitored node. The local proxying agent performs all the analysis of the results, rather than on the remotely monitored node. In some cases, this work is a performance advantage rather than a disadvantage. For example, in the AS/400 proxy architecture, you want to have the proxy UNIX host perform computation rather than using the more expensive AS/400 processor.

The proxy architecture is not a fully autonomous agent architecture. Some dependencies arise on the network connections between the proxy agent and the monitored machine. The remotely monitored node has no monitoring or auto-correction if the proxy node connection is

lost through network failure or machine failure. The proxying machine is solely responsible for monitoring and corrective actions. However, it does not depend on the presence of a console, thus retaining most of the advantages of an autonomous architecture. Therefore, autonomy of the agent is only partially lost in a proxy architecture. The console can still be absent and will not suppress monitoring or corrective actions. Corrective recovery actions can still occur independently of any console connection. However, they are affected if the proxy agent goes down or the network connection fails between the proxy agent and the monitored remote machine. In either of these cases, monitoring and corrective actions can be lost. However, to have the proxy Knowledge Module alert whenever this form of connection problem arises is trivial. This approach retains proactive monitoring notification.

Extensibility of a proxy Knowledge Module is different compared to a native agent, because the PSL functions do not run natively. Instead, a proxy Knowledge Module can be extended only by building on top of the PSL functions that are wrapping whatever proxy access method is used. This depends on the particular Knowledge Module implementation and is not generally using the basic PSL functions for operating system level access. A proxy Knowledge Module developer is responsible for documenting and externalizing the PSL API functions that are used to launch commands on the remote machine to permit user customizations.

KM Environment Variables

Every large application uses environment variables, meaning that the Knowledge Module monitoring that same large application must also deal with environment variables. PATROL provides a number of ways to deal with them. Environment variables become an integral part of managing the application.

The first level of Knowledge Module support for environment variables is the Knowledge Module attributes supporting them. At almost all levels of KM editing in the Developer Console is an “Environment...” button or menu. Via the Developer Console, you can define environment variables at the application, application instance, parameter, and recovery command levels.

The environment variables can be given a name and value via the GUI. This is the simplest way to deal with environment variables needed to manage the application. Decide which environment variables are needed, and manually enter the environment variable and its value via the GUI.

28.1 Child Process Environment Variables

The main need for environment variables is for launching child processes. The Knowledge Module has no need for environment variables unless it launches processes that require an environment variable. For example, they are needed if the application has a command-line interface that requires an environment variable. The environment variable is needed then for the child process launched by the PSL `popen` function launched to create the shared channel.

Environment variables affect processes launched by PSL `system`, `execute`, and `popen`. It also affects processes launched as an OS or non-PSL command type, either via PSL `execute` or directly by definition of these commands in the Knowledge Module.

28.1.1 Inheritance Path of Environment Variables

Environment variables can be defined at multiple levels of objects in the KM. When the KM launches a child process, it inherits the environment from whatever level at which it is running. For example, a parameter will merge the environment at the parameter, instance, and application levels. However, a menu command runs at the instance level and gets the values from the instance and application level.

In fact, the environment is actually also inherited from the computer classes and even the agent's startup environment variables. However, these computer class settings are not very useful to a KM, because they are things over which the KM has no control. The KM controls mainly its own application class, and this contains most environment settings used by a KM.

The most useful levels for the KM environment variables are the application level and the instance level. The application level is the top level affecting all processes launched by that KM application, its instances, menu commands, recovery actions, and parameters.

The instance level is useful when the environment settings affect each instance differently. For example, in the Oracle KM, each database server instance needs different environment variable settings. By installing environment variables at the instance level, each instance can have different values for the environment values.

28.2 PSL `getenv` Function

The PSL built-in function called “`getenv`” exists to query the environment. The value it receives should be the same as that which will be forwarded to a child process. PSL `getenv` is subject to the same inheritance hierarchy levels as child processes. This method of querying the environment is purely internal to the agent and is efficient. It is also supported across all agent platforms. PSL `getenv` can check only one variable at a time, and the code is simple:

```
out = getenv("MYVAR"); # Show one environment variable
```

Unfortunately, quirks in the agent make this equivalence between PSL `getenv` and child process environments not always the case. For example, in PATROL 3.1, the use of PSL `getenv` in discovery does not get application-level variables, nor does parameter-level execution get parameter-level environment settings. A more guaranteed way to test exactly what environment values are passed to a child process is by forcing a child process to report the environment values. The PSL commands to check the environment via a child process include:

```
# Show all environment variables
out = system("env"); # UNIX
out = system("set"); # Windows NT, OS/2
out = system("echo $MYVAR"); # UNIX
```

Put these calls in the same script as the child processes you are debugging. These simpler commands will receive the same environment as your other child process.

28.3 Dynamic Environment Values with %{} Macro Expansion

So far the discussion of environment variables has assumed static values entered via the Developer Console GUI. This situation doesn't really cut it. You definitely would not want to have to edit the KM manually to set up an environment variable. Although environment variable names may be common, their values will differ. Imagine having to edit the KM to change the environment attributes such as `ORACLE_HOME` for each Oracle database server instance.

For this reason, the KM provides a macro expansion method allowing dynamic values for the environment variables. The syntax is `%{macro_name}`. These `%{...}` macros are placed in the text for the environment value as set in the console. The macros are expanded by searching for the macro name in the agent's symbol table. This is the same symbol table as used by PSL `get` and `set`. Therefore, the value of an environment variable can be changed by combining PSL `set` and the `%{...}` macros.

The expansion of the macros also follows the inheritance levels from the current level up the tree. Therefore, the environment variable's value will receive macro values according to the symbol table execution context of the command. For example, menu commands and parameters will receive instance-level values if these macro names have been given values by PSL `set` at the instance level. Application discovery and pre-discovery will receive values only from the top-level variables.

The `%{...}` macros in the environment are expanded in real-time whenever a child process is launched. The text can also contain multiple `%{...}` macros that are all expanded. Usually, the symbol table contains a value for the macro name. Otherwise, when a child process is launched and the environment is macro-expanded, the `%{...}` macro without a value will generate an error. These error messages should be avoided, as should all spurious KM error messages. Therefore, if a variable is not needed, you should still use PSL `set` to create the macro name even if only with an empty string value.

28.3.1 Populating Environment Variable Values

The main method of setting a value of an environment variable is via the PSL `set` function and the `%{...}` macros. But how does the KM determine the value to store for the environment variable? Clearly, the answer depends on the application. Sometimes the KM can automatically determine the value. For example, the Oracle KM can determine the value for the `ORACLE_HOME` environment value directly from the "oratab" file. Not all applications are this easy. Sometimes you need to prompt the user for the value for the environment's value. Using PSL `response` to display a nice KM-specific GUI is still more elegant than having to manually edit the KM.

Once an environment variable's value is determined, either automatically or by user prompting, it can be stored to be reused. The easiest way is to use the PSL `pconfig` function for persistence. By using `pconfig`, the value is stored in the agent configuration database. The value becomes available via PSL `get`. The KM must then use PSL `get` to grab the value, and then PSL `set` to store it into the appropriate place. Typically, the PSL discovery script will

set the environment settings for each application instance just after creating the instance object with PSL `create`. This ensures that the settings exist for parameter or menu command scripts.

28.3.2 Dynamic Environment Variable Control

The main area of dynamic control over environment variables is to change the values. This is achieved via PSL `set` and the `%{ }` macros, as already discussed on page 413. Most other dynamic changes to the environment are not possible.

No way is available to dynamically add environment variables into the KM. This means you need to know before the fact the names of all the environment variables needed. You can dynamically control their values, but not their names. No PSL “`putenv`” or similar function exists. The only solution, if absolutely necessary, is to put “`set`” or “`setenv`” commands directly into the command text for PSL `system`, `execute`, and `popen`. However, usually the names of the environment variables are known ahead of time.

Also, you have no way to dynamically remove an environment variable from the group. The closest you can come is using PSL `set`, assuming that it is a `%{ }` macro, to the empty string. However, even then the environment variable will still show up in the environment of child processes and via PSL `getenv`.

28.4 PATROL Built-in Environment Variables

A few environment variables are handled specially by PATROL. Since PATROL is itself an application, it too has environment variables for various functions. Some of the PATROL environment variables appear in Table 28-1.

Table 28-1 PATROL environment variables

Environment Variable	Purpose	Agent Uses	Console Uses
PATROL_HOME	Main PATROL installation directory	Yes	Yes
PATROL_ADMIN	Username of file owner of configuration and data files that are created or written	Yes	Yes
PATROL_HISTORY	Location of parameter history data files	Yes	No
PATROL_LICENSE	Location of license file	Yes	Yes
PATROL_BIN	Directory of PATROL binary executable files such as <code>dump_hist</code> , etc.	Yes	No

Table 28-1 (cont'd) PATROL environment variables

Environment Variable	Purpose	Agent Uses	Console Uses
PATROL_GLOBAL_LIB	Global “knowledge” directory area override	Yes	Yes
PATROL_PSL_APPS	Global “psl” directory override	Yes	Yes
PATROL_CACHE	Directory containing the ‘local’ area	No	Yes
PATROL_LOCAL_PSL_APPS	Local “psl” directory override	No	Yes
PATROL_HELP	Directory containing help files	No	Yes
PATROL_DEVELOPMENT	Enable assertions and debugging code	Yes	Yes
PATROL_MACHINE_TYPE	Override the agent machine type	Yes	No
PATROL_BROWSER	Location of online help browser	No	Yes
PATROL_MIBFILE	MIB filename for SNMP support (used only if not set in agent’s configuration)	Yes	No
PATROL_LOG	Directory override of the “log” directory	Yes	No
PATROL_REMOTE	Directory override of the “remote” directory used for remote file sending	Yes	Yes
PSL_PROF_LOG	Agent PSL profiler log filename	Yes	No
PATROL_LANG	Language for console preference	No	Yes

In addition to these PATROL environment variables, a few specially handled variables exist. The “DISPLAY” environment variable is maintained on a per-console basis by the agent. During the initial handshake, the console tells the agent its DISPLAY value. This allows the agent to launch commands on the console’s X server using the DISPLAY variable. This trick is clearly UNIX-specific and does not work on a Windows NT console. However, no easy way is

available for the agent to determine whether it is connected to a UNIX or Windows NT console. Some of the common system environment variables that are also referenced by PATROL include those in Table 28-2.

Table 28-2 System environment variables

Environment Variable	Purpose	Agent Uses	Console Uses
LD_LIBRARY_PATH	Library loading path	Yes	No
TZ	Timezone	Yes	No
PATH	Executable loading path	Yes	No
HOME	Home directory	No	Yes
TMP	Temporary directory	Yes	Yes
TEMP	Temporary directory	Yes	Yes

All the external environment variables for the agent can be portably retrieved via PSL `getenv`. Some of these environment variables are available directly within the agent symbol tree hierarchy via PSL `get`. For example, `PATROL_HOME` can be found via PSL `get` on variable `"/patrolHome."`

28.5 Limitations of Environment Variables

Several pitfalls can occur in the use of Knowledge Module environment variables related to unexpected behavior. In a number of situations, environment variables are not picked up by either child processes or PSL `getenv`. Some of these problematic circumstances include:

1. Discovery scripts launching child processes do not inherit application-level environment variables defined in the application editing dialog. This statement is true of 3.2 and earlier agents.
2. PSL `getenv` in discovery scripts does not inherit application-level environment variables defined in the application editing dialog. This situation has changed in PATROL 3.2 agents, but not in pre-3.2 release agents.
3. Parameter-level environment variables defined in the parameter editing dialog are not found either by child processes or by PSL `getenv` in the parameter script or in recovery action scripts for that parameter. This statement is true of 3.2 and earlier agents.

4. Recovery action-level environment variables defined in the recovery action editing dialog are not found either by child processes or by PSL `getenv` in the recovery action script. This statement is true of 3.2 and earlier agents.
5. Instance-level environment variables via instance customization are not found by child processes or PSL `getenv` in some cases, such as parameter and recovery action scripts. This statement is true of 3.2 and earlier agents.

Some of the cases that are successfully handled are that application-level environment variables are inherited correctly in parameters, menu commands, info box commands, and recovery actions. Global environment settings in the computer class are also inherited correctly, but these are not so easy to affect when developing a KM application class.

The safest way to navigate these issues is to use only application-level environment variables defined in the application editing dialog. The problem with these variables and discovery scripts can be worked around by moving discovery logic that requires the environment variables into collector parameters or other places. The `%{ }` macros can be used to have smarter control over the environment variables if they are desired with different values in parameters, recovery actions, or menu commands.

Nested Instance Hierarchies

Nested instances are a visual feature of the PATROL Console where application instances can be nested beneath each other to create hierarchies. They can also be thought of as agent-side containers, where the agent controls the hierarchy of the console container objects. So when the console user drills into an instance object s/he can see other instance objects. These agent-side nested instance views are merged with console-side desktop view changes such as drag-and-drop moving of icons or creation of shortcuts.

The nested instances feature of PATROL 3.1 and later releases overcomes a limitation of PATROL 3.0 views that had only a three-level hierarchy: computers, instances, and parameters. This creates the opportunity to represent complex hierarchical object relationships within the KM representing an application.

Nested instances are desirable when the Knowledge Module needs to represent hierarchical information other than the default computer, instance, and parameter hierarchy. Some of the typical scenarios in which extra hierarchical information is best represented by nested instances include:

- Database objects—hierarchy of database servers containing tablespaces containing tables. The desirable visual behavior is to drill into the database server instance to see the tablespace, and then drill further down to see the individual tables.
- Proxying—when working on a proxy KM architecture, nested instances can be used to give some visual representation of the logical architecture to hide the true proxy architecture. Proxy architectures are covered in “Proxy Architectures” on page 402. The main issue here is the main map instances, as discussed in “Main Map Instances” on page 422.

Nested instances require a 3.1 agent and a 3.1 console or later releases. If for some reason you are developing a KM for the older 3.0 release of PATROL, you cannot use nested instances. PATROL 3.0 agents do not support the extra arguments to PSL `create`, and the 3.0 console cannot display the hierarchy, anyway.

29.1 Nested Instances via PSL `create`

The basic PSL primitive used to build inheritance hierarchies is the fifth optional argument to the built-in PSL `create` function. When called without the fifth argument, the PSL `create` function creates an ordinary non-nested instance. The full list of function arguments to PSL `create` is:

```
create(INSTANCE, LABEL, STATE, MESSAGE, PARENT);
```

where the parameters to PSL `create` have the following meaning:

- **INSTANCE**—name of the instance within the agent hierarchy.
- **LABEL**—name of the instance as shown on the console GUI display. Empty string defaults the label to the instance name.
- **STATE**—initial state of the instance when created.
- **MESSAGE**—message text stored in the event record associated with the instance creation. Can be the empty string to use the default message.
- **PARENT**—instance that is the parent of the new instance within the nested hierarchy.

Only by specifying a fifth argument can a nested hierarchy be created. The parent instance must already exist. The application class of the parent class can be the same class as the new instance or a different application class. Some examples of PSL `create` with nested instances include:

```
# Nesting of same class with relative paths
create("SERVER1", "", OK, "", "SERVER2");
# Nesting within another class
create("TABLE1", "", OK, "", "/SERVER/SERVER2");
# Using absolute paths
create("/TABLE/TABLE1", "", OK, "", "/SERVER/SERVER2");
```

Instances can be nested within the same application class or from different application classes. No limitations restrict what pattern of instance nesting is allowed. Also, no technical limitation controls the level of the nesting of objects, although practical console usability limitations call for a reasonable limit.

29.1.1 Common Nested Instance Pitfalls

The semantics of using PSL `create` for nested instances are not always as simple as they seem. Some common failures with PSL `create` for nested instances include:

- Using a parent instance that has not yet been created.
- Using instance names instead of full instance paths when using multiple application classes such as:

```
# WRONG! Full instance path required
create("TABLE1", "", OK, "", "SERVER2");
```

Assuming that `TABLE1` and `SERVER2` are in different application classes, then the above code has no chance of ever working, because it is not giving a full path but only instance names. Assuming that this code is running in the “TABLE” discovery script (otherwise, the `TABLE1` instance will not be created in the `TABLE` class, another common error), this is incorrect because it will look for a `TABLE` object named “SERVER2.” The correct code is to use a full path (either an absolute path or relative path) to the instance so that it looks in the correct application class for the parent object:

```
create("TABLE1", "", OK, "", "/SERVER/SERVER2"); # CORRECT!
```

The safest method is to fully path both objects so that no confusion arises about the application classes. The following code works regardless of where this code is executed from, whereas the code above assumes execution in the `TABLE` discovery script.

```
create("/TABLE/TABLE1", "", OK, "", "/SERVER/SERVER2"); # SAFEST
```

- Using the wrong path consisting of all parent instances in a multi-level hierarchy. The following code sequence exemplifies this error:

```
create("SERVER1", "", OK);    # Create database server instance
# Put tablespace instance under the database server instance
create("/TABLESPACE/TBSP1", "", OK, "", "/SERVER/SERVER1");
# Put the table instance under the tablespace instance ... WRONG!!
create("/TABLE/TABLE1", "", OK, "",
      "/SERVER/SERVER1/TABLESPACE/TBSP1");
```

Although this looks logical, the agent does not need the full inheritance path, and in fact, will fail. The correct path is a simple two-level path to the immediate parent instance object, ignoring other higher-level parents, regardless of how many levels deep the instance inheritance hierarchy is:

```
create("/TABLE/TABLE1", "", OK, "", "/TABLESPACE/TBSP1"); # CORRECT!
```

29.1.2 Create Icon for Class versus Nested Instances

A global application class setting called “Create Icon For Class” is in the application editing dialog in the Developer Console. This checkbox has some interesting interactions with the behavior of nested instances.

In the absence of any nested instances, the “Create Icon For Class” checkbox will create a class-level container into which it will place all application instances of the same class type. This is useful when you have an application class where many instances are likely to be. In effect, it is an automatic grouping into a hierarchy for a single type of application class’s instances. Examples include many of the operating system KM classes, such as PRINTER and FILESYSTEM in the Knowledge Module for UNIX.

If this checkbox is not set, all instances will be created at the same level immediately underneath the computer icons. This choice is really stylistic as to whether to place multiple instances under a default container or to have these instances at the same level as other instances. KM classes where the number of top-level instances is small, such as the Knowledge Module for Oracle, typically do not check this box.

When using nested instances, you are in control of the hierarchy. Therefore, you usually do not want the agent to automatically create this application-level container. In fact, the semantics of the interaction between nested instances and this container are complex, unintuitive, and rarely what you would want, anyway. Therefore, in my opinion, when using nested instances, this box is best left unchecked. Apparently the semantics are well-defined and merely place a new auto-created container level at any level where an instance of the class is created, but they have never seemed to work this way to me, although I have not tested it recently.

Note that the above recommendation applies to instances that will be the children of a nesting relationship but does not necessarily apply to the instances that will be the parents. You can check “Create Icon For Class” in the application class that are the parents.

The general guideline is therefore: Do not use “Create Icon For Class” in any application class where its instances might be nested instances under some other parent. For example, when nesting database tablespace instance icons under database server instance icons, you could check the box for the SERVER (parent) class, but not for the TABLESPACE (child) class.

29.1.3 Main Map Instances

The special parent instance named “/” is a special case in nested instances. This designates a “main map” object in the console display. Although the “/” instance is actually the computer instance for the current agent in other places, in this case with PSL `create`, it does not have this meaning. The use of “/” as the fifth argument in PSL `create` places an instance on the console’s main map. This means that the icon is placed at the same level as other computer icons representing agent connections. The syntax for the main map icons is the same as usual:

```
create("SERVER", "", OK, "", "/");
```

This will place the “SERVER1” instance on the main map of the console alongside all the computer icons for the agent machines. This feature has a few uses, including:

1. Application-specific views—By creating an icon representing all the instances of your application class, you can have an application-specific logical view of all instances of this type. For example, the Knowledge Module for SAP R/3 has this feature, creating an application logical view of an R/3 system regardless of whether it is distributed across multiple agent nodes.
2. Global application class logical views—The SAP R/3 KM can still have multiple icons at the top-most main map level, with one per R/3 system. The logical extension of this idea is to have one single main map instance to represent all the instances within a particular application class. For example, if the SYBASE KM used this idea, one global “SYBASE_WORLD” icon would appear as a main map instance, and all SYBASE database server instances would be nested beneath it. Drilling into any of the computer icons would then not find any SYBASE instance objects at all. This shows up one of the limitations of main map objects in that you cannot currently build up both logical and physical forms of the view in the same place from within the KM. Users can create console containers and shortcuts via GUI drag-n-drop to achieve this correlation, but it cannot be done automatically from within the Knowledge Module.
3. Proxy KM support—The use of a proxy architecture, as in the Netware KM and also the AS/400 KM, can benefit from the use of main map objects. By creating a computer-like application class and using main map instances, these proxy KMs can make their hierarchy look as if it were coming from a real physical agent.

29.1.4 Limitations of Nested Instances

Nested instances are mainly a visual feature and do not always properly interact with the rest of the PATROL feature set. Some of the known limitations of nested instances include:

- Nested instance parent instances must already exist when creating the hierarchy. You cannot create a nested instance before the parent instance exists, so you need to synchronize scripts such as discovery scripts when nested hierarchies are involved. On the positive side, this limitation prevents cyclical sequences of nesting.
- Parent instances cannot be changed after creation. The only place where the parent instance can be specified is via the PSL `create` statement, and the “parentInstance” variable is not writable from agent-side PSL. However, console-side reparenting is possible via icon drag-n-drop.
- Non-console views: Currently with PATROL 3.2, only the PATROL Console (Operator/Developer) on UNIX and Windows NT properly displays the nested instance hierarchy. Other views, including some PATROLVIEW integrations, PATROLWATCH for Web, and the `PatrolCli` command-line interface, do not fully handle nested instances. In the

case of PATROVIEW and PATROLWATCH, the hierarchy is flattened into the default three-level hierarchy in some of these views because not all of them support the nested instance hierarchy. If they are flattened, database server instances are placed at the same visual level as the database tablespace instances that are nested beneath them.

- Inheritance of a number of attributes does not follow the hierarchical path, as you can see in more detail in Table 29-1.
- Event handling of nested instances does not handle the hierarchy properly. For example, the events related to the nested instance contain no information
- Notification problems with determining the true origin parent arise from the limitations of the event handling of nested instances.
- No console container or shortcut object control comes from the KM: You have no way to control the console's creation of container objects or shortcut objects. This would be useful to create multiple hierarchies with the same object appearing in both a physical and logical view. Currently, the KM cannot do this, and can only place the nested instance within one hierarchy, with the console user having to create the second view if desired.

29.1.5 Inheritance Properties

Nested instances are mainly a visual feature of the console, and the inheritance properties of attributes within the agent do not always follow the nesting path. Some of these properties are identified in Table 29-1.

Table 29-1 Inheritance properties

Property	Inherited by Child Nested Instances?
Command Types (in parent's application class)	No
Environment variables (defined in parent application class or customized on parent instance)	No
Menu commands (in parent's application class)	No
Info box commands (in parent's application class)	No
Parameters (in parent's application class)	No
Alarm ranges (in parent's application class)	No
Recovery actions (in parent's application class)	No

Table 29-1 (cont'd) Inheritance properties

Property	Inherited by Child Nested Instances?
Destroying parent instance also destroys all child instance objects in a cascaded destroy sequence.	Yes
State change of parent instance cascades down to all child instance objects (e.g., PSL <code>change_state</code> to OFFLINE or a parameter alarm threshold exceeded).	No
Username and password customized settings for the application-level accounts.	Yes
<code>osUsername</code> and <code>osPassword</code> configuration variables for operating systems accounts.	Yes
PSL <code>blackout</code> on parent instance blacks out children in a cascaded blackout sequence.	No
Parameter properties in the parent instance parameters such as “active” or “interval.”	No

The opposite of inheritance is propagation. This is also less than ideal. For example, although the state of the child instance is propagated to the parent instance in the console, this propagation is not true for the agent’s built-in “status” fields.

29.1.6 Instance Name Collision Problems

Another problem with nested instances is that instance names can start to collide into one another. For example, consider having two database servers, `SERVER1` and `SERVER2`, both on the same agent machine. If both `SERVER1` and `SERVER2` have a tablespace called `TABLESPACE1`, then we would like to create an instance `TABLESPACE1` under both `SERVER1` and `SERVER2`. The code would appear to be simply:

```
create("TABLESPACE1", "", OK, "", "/SERVER/SERVER1");
create("TABLESPACE1", "", OK, "", "/SERVER/SERVER2"); # WRONG!
```

This fails and creates only one instance under `SERVER1`. The problem is that only one `TABLESPACE1` instance is allowed in the `TABLESPACE` class. Of course, not all such code

problems are this obvious, but if you are creating nested instances by name, they could get the same name requested.

The solution is simply to encode the parent path as part of the instance name to yield a unique name and avoid name collisions. The code would then become something similar to:

```
create("SERVER1_TABLESPACE1", "", OK, "", "/SERVER/SERVER1");
# CORRECT!
create("SERVER2_TABLESPACE1", "", OK, "", "/SERVER/SERVER2");
```

If you don't like seeing the name of the parent instance in the console, you can change the label argument in PSL `create`. The following code uses different instance names but the same label name:

```
# CORRECT! Labels differ from instance full name.
create("SERVER1_TABLESPACE1", "TABLESPACE1", OK, "",
      "/SERVER/SERVER1");
create("SERVER2_TABLESPACE1", "TABLESPACE1", OK, "",
      "/SERVER/SERVER2");
```

This will show the shortened name in the PATROL Console. However, other views are likely to still see the full text of the instance name. This statement is true of events in the event view, and typically of the PATROLVIEW and PATROLWATCH integrated views. This presence of the parent name in the object name can be a blessing too, in that it overcomes the problem that full nested paths are not shown in event views.

29.1.7 Parameter Nesting Issues and Limitations

You cannot control the nesting of parameters. These PSL `create` properties affect the nesting of instances but offer no control over parameters. The parameters under an instance icon when you drill into it are restricted to exactly those that are defined in the application class of which the instance is a member. So if you want to have a nested instance that you can drill into to see particular parameters, then those parameters need to be defined in the application class. You will see all the parameters of the application class, with no possibility to add any or exclude any from the view.

This limitation typically leads to the need for multiple application classes when using nested instances. If you want a particular parameter set under an icon, you will often need a new application class just for that list of parameters.

29.2 Grouping Parameters via Nested Instances

A common use of nested instances is to create parameter grouped views. For example, in your database server KM, you would often like to drill into the server and see icons representing different classes of parameters. For example, rather than immediately seeing 50 parameter icons,

you might wish to see a few containers such as “Availability Parameters,” “Performance Tuning Parameters,” and “Capacity Planning Parameters.”

Unfortunately, because of the limitations of nesting parameters, the only method of achieving this grouping view is to create new application classes with just those parameter definitions. For example, now in addition to a `SERVER` application class, you need `SERVER_AVAIL`, `SERVER_PERFORM`, and `SERVER_CAPACITY` classes. The `SERVER` discovery class becomes complicated by the need to create these three application instances nested under every `SERVER` class instance. The collector parameters populating all the parameters also need to be changed to find the correct instance to populate. That effort is a lot of work for a view of parameters, but it does achieve the desired result.

Beware when making this change that not all the views will look correct. Some `PATROLVIEW` and `PATROLWATCH` views will have a flattened view, and will not show the desirable hierarchy.

29.3 Determining Parent-Child Relationships

On the console, you can easily visually determine the parent-child relationships. On the agent, the task is not so easy. To determine the parent instance name of an instance, you have the special built-in “parentInstance” variable in the symbol tree. This variable can be accessed via PSL `get`.

```
parent = get("/TABLESPACE/TABLESPACE1/parentInstance");
```

The returned value from “parentInstance” is the full path of the instance name. For example, the above call might return “/SERVER/SERVER1” as the full path of the parent instance. If no parent instance exists, the returned value is the empty string.

Determining all the children of an instance is not simple in PSL. No special symbol lists all the child instance names. The only solution is the ugly, inefficient one of scanning all instances that might be children and determining whether they are under that parent. The code for that might look like that below, which finds all `TABLESPACE` instances nested under the `SERVER1` instance.

```
parent = "/SERVER/SERVER1"; # Full path, not instance name
children = "";
insts = get("/TABLESPACE/instances"); # Get the list of instances
foreach inst (insts) {
    if (get ("/TABLESPACE/" . inst . "/parentInstance") == parent)
    { children = children . inst . "\n"; } # Add it to the list
}
```

No representation of parent information appears in the MIB. Therefore, via `SNMP`, you have no way to gather parent-child information. This is one of the reasons that `PATROLVIEW` products tend now to use the `PATROL` API rather than `SNMP` to gather agent information.

Since the parent instance field is available in the symbol tree, the parent information can be gathered via the PATROL API. This allows PATROVIEW implementations via the PATROL API to show the relationships. You also have the possibility of executing remote PSL commands to gather this information using PSL scripts similar to the examples above.

PSL Libraries

PSL libraries are function libraries similar to those in other procedural languages. They permit user-defined functions to be grouped into a logical unit through combination into a physical library file. PSL libraries offer the typical advantages of code reuse, extensibility by additional abstractions, and some space usage reduction.

PSL library files have a suffix of “.lib” to indicate they are libraries. The format of a “.lib” file is a binary format based on PSL’s internal quad-code format. The format is not a native executable or DLL format. The binary format is platform-independent and does not suffer from portability issues such as “endian-ness” or network byte order constraints. This platform independence is achieved through the use of a byte code format specific to PATROL.

30.1 PSL Library Usage Syntax: requires

A PSL script loads a PSL library using the “requires” keyword. The `requires` keyword is followed by the library filename, either directly as an identifier or in double quotes as a string literal. Assuming the existence of a PSL library file named “myfuncs.lib,” the following `requires` statements should be identical:

```
requires myfuncs;  
requires "myfuncs";  
requires "myfuncs.lib";
```

However, the use of a suffix outside the quotes is not allowed:

```
requires myfuncs.lib; # ERROR!
```

Note that the string literal format was introduced in PATROL 3.1 and will fail with a compilation error on PATROL 3.0 agents and consoles. The format without quotes offers better compatibility with older PATROL releases, if this is a consideration.

On encountering a `requires` statement, the agent will load a PSL library file into memory, if it has not already been loaded (e.g., if it was required by another PSL script). The agent must find the library file in the `"/lib/psl"` or `"\lib\psl"` directory offset from the `"PATROL_HOME"` environment setting. The functions and global variables that have been exported by the PSL library are available to be used in the PSL script at any point after the `requires` statement.

This situation is also similar to when the PATROL Developer Console performs a syntax check of a PSL script containing a `requires` statement. However, the console will also look in the `"local"` directory area before the `"global"` area for the PSL library file when loading it.

30.2 PSL Library Definition Syntax: `export`

The main keyword in the definition of a PSL library is the `"export"` keyword. This keyword is used to explicitly export a PSL user-defined function or a PSL global variable. A function can be exported by the `"export function"` syntax:

```
export function myfunc;
```

A function can be exported before definition. A function cannot have the same name as a built-in PSL function name such as `"ntharg"` or `"grep."`

30.3 Exported Global Variables

A global variable in a PSL library can be exported using the `"export"` syntax:

```
export myvar;
```

A global variable can be exported before its first use. A function's local variable or parameter cannot be exported.

Exported global variables have a distinct value for each PSL process using the PSL library. Therefore, they are not exactly like a DLL architecture, which would have only one value for any global variable. For example, if two PSL scripts require a library and it exports a global variable `'x'`, then the two PSL scripts can set `'x'` to different values without interfering with each other. This possibility is usually convenient in preventing bugs, but can be limiting in that PSL processes cannot use PSL libraries to communicate with each other. On the other hand, the PSL `get/set` symbol tree hierarchy is more than adequate for PSL processes to communicate together, so this is not a major disadvantage.

Exported global variables cannot be easily initialized. They have the empty string value at startup of the PSL process, just like all PSL variables, but reliance on this fact can trigger a PSL runtime warning message if you have the `"Ps1Debug"` settings enabled. The initialization to

the empty string is well-defined and can be used provided that you disable `Ps1Debug` with simply `"Ps1Debug=0;"` to avoid an annoying message. Other than this empty string value, you have no easy way to give a global variable a value. A PSL library cannot automatically call an initialization function when it is loaded. Instead, the only solution is an explicit initialization function in the library that must be called by every PSL script that uses the library. Alternatively, this can be hidden by calling the initialization routine before any other library processing occurs, but this is less efficient, since it requires checking each time whether initialization has occurred yet.

30.4 Creating PSL Libraries

The PSL standalone compiler executable is called `"psl"`. This is the command that is used to create PSL libraries and binaries. The syntax to create a PSL library `".lib"` file is the `"-l"` option:

```
psl -l myfuncs.psl
```

This command will create a file `"myfuncs.lib"` in the current directory. This command must be executed in the current directory containing `"myfuncs.psl"`, and does not automatically find PSL files in either local or global directories.

The Developer console has no way to create or edit PSL libraries. The only method is the command-line method of using the `"psl"` executable.

30.4.1 Nested PSL Libraries

No limit exists on the level of nesting allowed in PSL libraries. A PSL script can require library A, which requires library B, which requires library C, and so on. In this case, B must exist before A will compile, and C must exist before B will compile successfully. Therefore, the compilation sequence has to start at the bottom leaf nodes of the `requires` hierarchy. The sequence of A requiring B requiring C would be:

```
psl -l C.psl      # creates C.lib
psl -l B.psl      # creates B.lib
psl -l A.psl      # creates A.lib
```

No implicit inheritance of the exported functions occurs. For example, if A requires B which requires C, then A cannot call the functions exported by library C unless A explicitly requires also library C.

Cyclical dependencies of PSL libraries are also not handled well. Although nothing explicitly prevents cyclical `requires` situations, they do create problems. For starters, how do you compile them? If A requires B and B requires A, then A must exist to create B, and B must exist to create A. It is like the chicken and egg. You can get around this conundrum through some tricks involving bootstrapping the library by compiling the first one without the cyclical

dependency and then re-compiling both libraries. However, since it can cause unpredictable agent behavior, why would you want to do this?

Unfortunately, PATROL 3.0 agents have a bug that causes unpredictable agent behavior with three or more library nesting levels. This refers to the situation of A requires B requires C requires D, or any deeper nesting. This problem is fixed in PATROL 3.1 agents but may affect compatibility of the KM to PATROL 3.0 agents. Fortunately, three levels of nesting are usually adequate for most KM needs.

30.4.2 Changing PSL Libraries

A number of difficulties occur when changing PSL libraries during KM development. Understanding the issues will help avoid pitfalls and save time.

The PATROL agent does not automatically load a new PSL library “.lib” file. Instead, it retains the older version of the library in memory, even after all PSL processes that required it have terminated. Therefore, when updating a new “.lib” file, you cannot simply unload a KM or kill all PSL processes. Unfortunately, the only method is to kill and restart the agent, or to use the “Reinitialize Agent” menu.

Getting the PSL library “.lib” files to the agent is another issue. PSL libraries are not automatically downloaded by the Developer Console, nor are they picked up by the “Commit KM” menu command (although in PATROL 3.2, you can use the `EXTRA_FILES` feature of the “ExtraFilesList” parameter for this). During development, PSL library files can be manually copied to the agent machine via FTP, NFS sharing or network drive mapping. Another possibility is to develop the PSL library directly on the agent machine and execute “`psl -l`” directly in the agent’s global directory. This can be effective, but is confusing if some files are created on the console and others on the agent, and this also does not work well when developing or testing on more than one agent.

Nested libraries also impose an annoying sequence when you change a library. When you change a low-level PSL library, you must re-compile all PSL libraries (or PSL binaries) that require that library. If library A requires B, then changing B will also force re-compilation of A. However, changing A does not force a re-compilation of B. So often when you change a low-level library, you have to re-compile many libraries and copy all these new library files to the agent’s global directory. An automated script or `makefile` is recommended to automate this compilation sequence of libraries.

Clearly, a number of steps are necessary when changing a PSL library. This causes an elongated cycle when developing a KM involving PSL libraries:

1. Change the PSL library text file via an external text editor (not the console).
2. Re-compile the PSL library via “`psl -l`” to create a new “.lib” file.
3. Re-compile any dependent other PSL library files with “`psl -l.`”
4. Re-compile any dependent PSL binary “.bin” files with “`psl -b.`”
5. Copy or FTP one or more changed “.lib” files to the agent’s global “`psl`” directory.

6. Copy or FTP any changed “.bin” files to the agent’s global “psl” directory.
7. Ensure that library file permissions are adequate to allow the default agent account to read the library files.
8. Bounce the agent (e.g., via the “Re-Initialize Agent” menu).

The cause of all this sequence is a design involving late linkage of the PSL library and binary format. Function and variable names are mapped to integer offsets too early, making PSL library loading efficient because this name mapping does not need to happen in the agent, but leading to these problems of inflexibility when the functions changed. Late linkage of PSL may appear in a future release, but for PATROL 3.2 and earlier, these problems are unavoidable.

The agent can offer a number of strange failures if some of these steps are missed. The agent does not cope well with inconsistent PSL libraries. Fortunately, it does not crash the agent, but it will offer a number of error messages such as “fatal inconsistency detected” and terminate the script.

Not all failures are this obvious. Some of the “quiet” failings involve the wrong PSL library function called or the wrong global variable accessed. When debugging a strange KM or PSL failing that appears to defy logic, consider the possible failings from PSL libraries. Did you restart the agent the last time you changed some libraries? Did you re-compile dependent PSL libraries when you changed just one of them? Did you copy the newly created “.lib” library files down to the agent’s global “psl” directory before restarting the agent? Did you do the same for any “.bin” files? In the face of strange behavior or a PSL coding bug you thought you had fixed, these PSL library issues could be causing you to chase ghosts instead of your real PSL coding bugs.

In PATROL 3.1, you have no way to detect the status of the libraries. However, PATROL 3.2 offers the built-in agent command `%DUMP LIBRARIES` to list the PSL libraries and when they were loaded. The 3.2 agent also has a warning on PSL script compilation if the agent has a library version in memory but the file on disk appears to be more recent. This helps to identify situations when the agent has not been bounced since a PSL library was changed. However, this does not help in the inconsistency problems involving nested library dependency problems.

30.4.3 Determining Library Dependencies

The PSL standalone compiler, “psl,” has a feature to report which PSL libraries are required by another PSL script or binary file. The “-R” option is used to report the dependencies for a file. Some examples would be:

```
psl -R myfuncs.psl    # check PSL source
psl -R myfuncs.bin    # check PSL binary
```

The output from the “-R” option is a report that no PSL libraries are required such as:

```
`myfuncs.psl' requires no PSL libraries.
```

Alternatively, if PSL libraries are required, the output will look similar to:

```
'myfuncs.psl' requires the following PSL libraries:
libA      DYNAMIC PSL
libB      DYNAMIC PSL
```

The columns in this report are the name of the library without the suffix, the loading status of the library, and the type of library. The status is either dynamic or static, depending on whether the “-s” option to “psl” was used to create a static linked library. Generally, all PSL libraries are dynamic. The type refers to whether the library is a normal PSL library, as discussed here, or an XPC-based DLL-like PSL library.

Unfortunately, the “-R” function is not supported on PSL library files, and fails with an error message:

```
psl -R myfuncs.lib    # check PSL library NOT SUPPORTED!
```

This limitation is of the “psl” executable and not a problem with the library file formats. One inelegant workaround for that may well be to copy “myfuncs.lib” to “myfuncs.bin,” and then try “psl -R myfuncs.bin.” Because of the similarity of “.bin” and “.lib” file formats, this will show the library usage dependencies.

30.4.4 Static Linked PSL Libraries

Static linking is rarely used in KM development. It has value only in obscure cases where you can reduce the number of “.lib” files or you need to avoid dependence on some standard PSL library. PSL libraries can be statically linked via the “psl -s” option. This possibility is useful if you have multiple levels of PSL libraries and want to merge them into a single “.lib” file. The command is similar to:

```
# Merge libraries into one static library
psl -s -l -o myfuncs.lib A.lib B.lib C.lib
```

However, doing this static library merging loses the advantages of shared memory usage for the lower-level libraries. These libraries are now statically linked rather than separately dynamically loaded. Some cost occurs and rarely any gain.

30.4.5 Performance Issues of PSL Libraries

No difference occurs in runtime performance in terms of CPU usage whether a user-defined PSL function is stored in a PSL library or directly in a main PSL file. The advantages of PSL libraries arise in coding practices rather than performance. Note that also no performance penalty occurs for the use of PSL libraries. All libraries are internally cached by the agent and have the same performance as other PSL scripts.

Some gain appears in terms of memory usage of the agent from the use of PSL libraries. The agent maintains only one copy of all the read-only data for a PSL library, including all the quad-code instructions, constants, and linkage data. This leads to space saving from using a PSL function in a library rather than copying it into many files, which is poor programming style also. The agent maintains multiple copies of PSL library dynamic information, such as the values of local, global, and exported variables, with one copy per PSL process using the library.

30.5 Debugging PSL Libraries

The PSL Debugger in the Developer Console has poor support of PSL libraries. No support is available for multiple PSL source file viewing in the debugger. Therefore, although you can step through a PSL library, you will be stepping blind without viewing the source code as you step. Problems also occur with using the “print,” “break,” and other debugger statements because the debugger does not fully handle library context, and may confuse the context of local or global variables in the PSL library with those in the main script.

The usual solution to debugging PSL libraries is to use debugging printout statements with PSL `print` or `printf`. This is a simple and effective solution. The debugging flags need to be externally controlled by the client using the PSL library. One solution is to export a global variable, say “myfuncs_Debug,” and have the client PSL script set this variable if debugging is desired. Another alternative is a PSL exported library function to control debugging settings.

30.6 Limitations of PSL Libraries

Although PSL libraries suffer from limitations, almost all large KM projects necessarily involve PSL library files. Some of the problems to be overcome include:

- Developer console KM downloading does not send PSL library files.
- Commit KM does not send PSL library files.
- “`psl -R`” does not work on “.lib” files.
- “`psl -R`” requires the libraries to exist and reports an error if not.
- “`psl`” only finds libraries in the current directory and has no `-I` or `-L` option to give an alternative directory. However, an environment variable named “`PSL_LOAD_PATH`” can be used.
- You cannot create libraries from the Developer Console KM editing GUI dialogs.
- PSL Debugger does not handle PSL libraries well. You cannot easily step through library source.

Despite these problems, the use of PSL libraries is good coding practice. It encourages code modularity and re-use and simplifies larger PSL coding problems. The problems with distribution are solved by including the PSL library files in agent-side installation routines. The majority of other difficulties occurs during development of the KM and do not affect the end user of the final packaged KM.

SNMP KM Development Features

The Knowledge Module developer needs to gather data about the monitored device/application. SNMP is a useful and valid source of data if the device/application is instrumented with SNMP support. A large number of modern hardware devices and software applications are instrumented with SNMP support. They contain a full SNMP agent that permits SNMP-based access to data about the device or application. Querying their MIB is a powerful way of gathering data about the device/application that can be used in a Knowledge Module.

The other direction of management is also possible via SNMP set capabilities. PATROL can use SNMP set to change a device or application, provided that the MIB for that device or application has writable values.

Trap receiving is another use of SNMP by PATROL to monitor devices/applications. PATROL can receive traps sent by the device/application and convert them to PATROL alarms. This standardizes all the detected problems, whether they are internally detected by the device/application itself (which then sends the trap), or whether PATROL detects them via other means. All failures raise a PATROL alert.

PATROL can also send SNMP traps. This powerful feature is related to notification about events to other third-party SNMP management consoles or event management consoles. This issue is more relevant to deployment issues surrounding PATROL, rather than Knowledge Module development, and is further discussed in “SNMP Integration” on page 239.

PATROL also has its own MIB that can allow access to PATROL agent data via SNMP. This is not of great interest for KM Development in most cases. It is useful as a possible inter-agent communication method in some cases, although the PSL remote-access functions largely supersede this usage.

31.1 Processing SNMP MIB Data via PSL

A number of PSL functions are important for the development of Knowledge Modules that read SNMP data from the MIB objects of other devices/applications. They can also read the PATROL MIB values, but this way to read an object that is already readily accessible within PSL itself is too long. Hence, Table 31-1 shows functions that are mostly useful for KM development tasks where the aim is to bring MIB data into the KM.

Table 31-1 SNMP MIB access PSL functions

PSL Built-in Function	Purpose
<code>snmp_open</code>	Open a new session to be used for SNMP “get/set/get-next/walk” operations.
<code>snmp_close</code>	Close a session used for SNMP “get/set/get-next/walk” access.
<code>snmp_config</code>	Report on sessions currently open.
<code>snmp_get</code>	Perform an SNMP “get” action on a MIB (using an open session).
<code>snmp_get_next</code>	Perform an SNMP “get next” action on a MIB (using an open session).
<code>snmp_h_get</code>	Perform an SNMP “get” action on a MIB using a brief session.
<code>snmp_h_get_next</code>	Perform an SNMP “get next” action on a MIB using a brief session.
<code>snmp_h_set</code>	Perform an SNMP “set” action on a MIB using a brief session.
<code>snmp_set</code>	Perform an SNMP “set” action on a MIB (using an open session).
<code>snmp_walk</code>	Perform an SNMP “walk” operation on a MIB using an open session.

31.1.1 SNMP MIB Data Processing

The simplest way to read MIB data is via the PSL “`snmp_h_get`” function. This creates a one-time session for SNMP requests, and gathers the data from the SNMP “get” request. An example of such PSL code is the following simple wrapper to the `snmp_h_get` built-in function:

```
function snmp_get_mib_value(hostname, mibvar)
{
```

```

local ret;
ret = snmp_h_get(hostname, mibvar) ;
if (errno) {
    # Error - failed to get the data
    print("ERROR: MIB get fail:", hostname, ": ", mibvar, "\n");
    return ""; # Return some failure code
}
return ret;
}

```

This simple code is effective for short data gathering expeditions. For longer sequences where many MIB values are gathered, you want to open a session via `snmp_open` and use `snmp_get` to get each MIB value before closing the session with `snmp_close`.

31.1.2 Advanced MIB Data Processing

The example below is a script that has some smarter index processing. This script will get an object by object id and index, or add a “.0” to the oid if no index is supplied. It also does some post-processing of the results returned by `snmp_h_get` in order to gather the real value of the variable and discard the origin information in the return code.

```

# returns a value for VAR.IDX on HOST
# example: snmp_h_get_value("mybox", ".1.3.6.1.2.1.1.1", "");
# returns the value of sysDescr on mybox

function snmp_h_get_value( host, mibvar, mib_index)
{
    local result;
    if (mib_index == "" ) { mib_index = 0;}
    result = snmp_h_get(host, mibvar . "." . mib_index);
    if (errno != 0 ) {
        result = "<error in response>";
    } else {
        # Get the real data fields
        result = ntharg(result, "3-", " ", " ");
    }
    return result;
}

```

The following example is a user-defined PSL function that creates a list of indexes for a given MIB object. It does so using the PSL `snmp_walk` function.

```

# gets a list of indexes for the given variable
function snmp_get_indexes (host, oid)
{
    local sess, i, ret;

```

```

    sess = snmp_open(host);
    if (sess == "") { return "" ; }
    ret = "";
    foreach line i (snmp_walk(sess, oid)) {
        if ( ret != "" ) { ret = ret . "\n";}
        ret = ret . ntharg(i,3);
    }
    snmp_close(sess);
    return    ret;
}

```

31.2 SNMP Ping: Detecting SNMP Support

You can detect whether a valid SNMP agent exists on a machine via a simple “get-next” request. The following PSL code should detect whether the local machine has valid SNMP support.

```

hostname = get("/hostname");
ret = snmp_h_get_next(hostname, ".1");
if (errno) {
    # SNMP support not valid
}
else {
    # Success gathering SNMP data
}

```

The `snmp_h_get` function can also be used in this manner. Both of these functions can clearly be used to detect the SNMP agent on remote machines, because they are not limited to the local machine. Another implementation of a “ping” style operation via SNMP is:

```

function ping(host)
{
    local sess, result;
    if(host == "" ) {
        return "snmp_ping: ERROR: Empty host\n";
    }
    sess = snmp_open(host);
    if( sess == "" ) {
        return("snmp_ping: ERROR: cannot open session\n");
    }
    result = snmp_get_next(sess, ".1");

    if( result == "" ) {
        result = host . " DOESN'T ANSWER";
    } else {
        result = host . " HAS SNMP AGENT";
    }
    snmp_close(sess);
}

```

```

        return result;
    }

```

31.3 Processing SNMP Traps from Other Devices or Applications

The KM developer can also add features where the PATROL KM can raise PATROL events and alarms when it receives a trap from some other SNMP-enabled device/application. The PSL functions that support this method of SNMP trap receiving and analysis appear in Table 31-2.

Table 31-2 SNMP trap receiving PSL functions

PSL Built-in Function	Purpose
snmp_trap_ignore	Ignore traps incoming to the PATROL SNMP subagent.
snmp_trap_listen	Start recording incoming traps to the PATROL SNMP subagent.
snmp_trap_receive	Read the traps accumulated from incoming traps to the agent.

31.3.1 Example: Agent Receiving Its Own Traps

A good example of how the agent can receive traps via PSL is to register for its own traps. The sequence of PSL code to achieve this is relatively simple:

```

# 1. Register interest to receive traps
# For this self-testing case,
# it is an internal registration with the agent itself via PSL

errno = 0;
# Add self as recipient/target host
ret = snmp_trap_register_im("localhost", "add");

if (ret != "OK") {
    exit; # Replace with whatever error handling is appropriate
}

# 2. Ensure that the SNMP agent is listening for incoming traps
E_PSL_SNMP_ALREADY_LISTENING = 98;
ret = snmp_trap_listen();
# Could return either OK or "" depending on if already going
if (ret != "OK" && errno != E_PSL_SNMP_ALREADY_LISTENING) {
    # Detected some failure, which was not a harmless indication
    # that the listener was already listening.
    exit; # Replace with appropriate error handling
}

```

```
# 3. Loop waiting for incoming traps
while(1) {
    errno = 0;
    ret = snmp_trap_receive();    # Wait for incoming trap

    if (ret == "" || errno != 0) {
        # Some failure to receive a valid trap
        exit; # Replace with appropriate error handling
    }
    # process the incoming received trap here
    print("TRAP DETECTED: ", ret, "\n");
} # while loop
```

This sample PSL code illustrates the basic ideas behind receiving SNMP traps via PSL. The first step is not typical, because registration for traps is internal to the agent. The KM is registering with the agent to receive traps from the agent. A more typical use of registration for SNMP traps from some external non-PATROL device or application will require some other steps, such as either an SNMP “set” operation via PSL or a PSL `system` call to launch a child process.

31.3.2 Registering with a Device or Application to Receive Traps

The PATROL agent must also register with the device/application in order to receive the traps. This action may involve other PSL SNMP functions to register via SNMP “set” or similar. The steps to register for interest in the SNMP traps depend on the device/application itself. Examination of the MIB for the device/application is required to determine how to register for traps in SNMP V1, although this is defined in SNMP V2. The sequence for trap reception is similar to the example code above, except that the first registration step is not internal to the agent:

1. Register with device/application for trap interest: e.g., SNMP set operation via PSL `snmp_h_set`.
2. Ensure that the PATROL SNMP support is listening for traps via `snmp_trap_listen`.
3. Receive traps with PSL `snmp_trap_receive` function.

31.4 Other SNMP Features

A number of other features are available in SNMP and PATROL. The majority of these advanced features are not needed for KM development tasks. Nevertheless, here are some of the other SNMP features:

- PATROL SNMP Toolkit
- Proxying via SNMP
- Inter-agent communication via SNMP

- Sending SNMP traps
- PATROL MIB

I now briefly examine each of these SNMP features in turn.

31.4.1 PATROL SNMP Toolkit

The use of SNMP data for monitoring a software application with a Knowledge Module assumes that the application has its own MIB and SNMP support. This is not always true for all software applications. For those without SNMP support, the PATROL SNMP Toolkit is an advanced suite of development tools to enable the instrumentation of the application with this SNMP and MIB support.

This toolkit is used in the more advanced cases of Knowledge Module development, where the application can itself be changed to be more manageable. This is also referred to as “intrusive instrumentation,” and this topic is fully covered in Chapter 16.

31.4.2 Proxying via SNMP

The SNMP protocol supports access over the network to machines other than the current one. Therefore, all the comments regarding SNMP MIB data gathering and SNMP trap receiving apply to the proxy architecture. You can gather data about a machine, a device, or an application on a remote machine without having a PATROL agent on the remote machine. This machine must have an SNMP agent on the machine. The agent can then gather the data via PSL about the remote machine using the SNMP PSL functions. Further discussion of the idea of a proxy KM architecture can be found in “Proxy Architectures” on page 402.

31.4.3 Inter-Agent Communication via SNMP

A special case of KM development is the development of KMs that use multiple agents. Since the PATROL agent supports both get and set access, SNMP can be used as a technology to communicate data between agents or to synchronize their actions. The SAP R/3 KM and the History Loader are examples of KMs that use multi-agent architectures. However, neither of these examples use SNMP, since inter-agent SNMP access is largely superseded in 3.1 by PSL and API inter-agent features. For further discussion of inter-agent KM issues see Chapter 27.

31.4.4 PSL SNMP Trap Sending

The main PSL function for sending traps is `snmp_trap_send`. An example usage is shown below using dummy values:

```
recipient = "some.host";
ipAddress = get("/ipAddress");
bmc_enterprise_id = ".1.3.6.1.4.1.1031";
trap_origin = ipAddress . " " . bmc_enterprise_id . " 6 255 1000";
```



```
mesg = "Whatever trap text message you like";
trap_text = ".1.2.3.4.5 string " . mesg;

ret = snmp_trap_send(recipient, trap_origin, trap_text);
if (ret != "OK") {
    print("ERROR: PSL snmp_trap_send returned '"
        . ret . "' with errno " . errno);
    # Other error-handling as appropriate
}
```

Sending SNMP traps is not normally the domain of the KM Developer. Sending SNMP traps from the PATROL agent is a general deployment and configuration issue that is usually handled by the PATROL administrator. A KM does not normally provide its own built-in SNMP trap sending support, since this can be added outside the KM for all KM alarms in the entire agent. Further discussion of SNMP traps is found in Chapter 19.

31.4.5 PATROL Agent MIB Data

The PATROL agent also has its own MIB. This allows external SNMP access to data gathered by the agent and limited set capabilities for some configuration parameters. The agent's MIB is a dynamic MIB that is dynamically populated by the KM classes, instances, and parameters as they are created. The KM developer need do nothing special for their objects to appear in the PATROL MIB.

Accessing the agent MIB data is not typical of a KM. After all, this data is all available internally via PSL `get`, and this approach is more efficient. The only KM development issue involving MIB data access is for KMs that have multiple agent architectures. These multi-agent Knowledge Modules are discussed in more detail in Chapter 27.

Advanced PSL String Operations

The string is the basic data representation for all variables in PSL. This leads to a powerful set of operators and built-in functions to operate upon string types. This chapter examines how to extend these string primitives even further to more advanced string processing problems that arise in real-world KM development. String data processing is fundamental to various common tasks of a KM, such as processing the output of a system command, processing the returned columns of a SQL script, or searching error log text files.

32.1 Basic String Operators and Functions

A large number of built-in operators and functions operate on strings. These operations are all efficient, with the majority occurring in a single atomic instruction.

The operators that apply to strings are shown in Table 32-1. The fact that all the normal relational operators apply to strings is indicative of the fundamental usage of strings in PSL as a data representation.

Table 32-1 PSL string operators

String Operator	Meaning
<code>x . y</code> (period)	String concatenation
<code>str =~ pattern</code>	Regular expression matches (Boolean operator)
<code>str !~ pattern</code>	Regular expression does not match (Boolean operator)

Table 32-1 (cont'd) PSL string operators

String Operator	Meaning
[x,y]	List operator (builds newline separated lists)
x == y	Relational equality operator: checks whether two strings are equal
x != y	Relational inequality operator: checks whether two strings are not equal
x < y	Relational less than (dictionary order)
x > y	Relational greater than (dictionary order)
x <= y	Relational less than or equal (dictionary order)
x >= y	Relational greater than or equal (dictionary order)

The built-in functions that are useful for string data processing appear in Table 32-2. None of these functions launch external system processes, not even `tail` or `grep`.

Table 32-2 PSL string processing functions

Function	Meaning
sprintf	Format strings according to complex specifications
grep	Search string for lines matching regular expression pattern
sort	Sort the lines in a string in various ways
unique	Remove duplicate lines from a string
index	Search string for pattern string (not regular expression)
rindex	Reverse search for pattern string (not regular expression)
length	Return number of characters in a string

Table 32-2 (cont'd) PSL string processing functions

Function	Meaning
lines	Return the number of newline separated lines in a string
substr	Cut a specified character range from string
isnumber	Test whether a string has a numeric form
tail	Cut the last lines from a string
tolower	Convert string into lowercase characters
toupper	Convert string into uppercase characters
ntharg	Cut columns from a single or multi-line string (adjacent separators as a single separator)
nthargf	Cut columns from a single or multi-line string (adjacent separators broken)
nthline	Cut lines from a string (adjacent separators as a single separator)
nthlinef	Cut lines from a string (adjacent separators broken)
difference	Set difference operator (sets are newline-separated strings)
intersection	Set intersection operator (sets are newline-separated strings)
subset	Set subset operator (sets are newline-separated strings)
union	Set union operator (sets are newline-separated strings)

In addition to these operators and functions, the PSL `foreach` loop is useful for analyzing strings by line or by word.

32.2 Testing Strings

Strings can be tested by the relational operators (i.e., `>`, `<`, `==`, `!=`, etc). This capability allows all the obvious forms of testing strings for alphabetical ordering such as:

```
if (s1 > s2) { ... }    # s1 is after s2 in alphabetical order
if (s1 == s2) { ...}    # s1 and s2 are identical strings
```

These relational operators are highly efficient. However, when testing multiple values, some opportunities for minor optimizations arise. Testing strings for multiple values could be coded simply as:

```
if (severity == "HIGH" || severity == "LOW"
    || severity == "MEDIUM")
{ ... }
```

This can be micro-optimized to a slightly faster PSL form using the PSL set-related function named “subset” to test whether the string is in a set of values:

```
if (subset("HIGH\nLOW\nMEDIUM", severity) { ...}
```

Another similar trick using PSL index would be:

```
if (index("!HIGH!LOW!MEDIUM", severity) { ...}
```

However, this use of `index` is slightly wrong, in that it can give false positives if the variable contains a subset of one of the values. The above code using `index` is wrong if “severity” is ‘MED,’ for example, although this may or may not be a failing in this example case.

32.2.1 Testing Single Characters

Testing whether a character is in a set of characters is easy using PSL `index`:

```
is_whitespace = index(" \t\r\n", ch);
```

The use of PSL `trim` is another possible method. As shown later, this idea also applies to strings, not single characters only.

```
is_whitespace = trim(ch, " \t\n\r") == "";
```

Other examples are not so elegant. No better way of testing for lower- or uppercase characters is available than:

```
is_lower = index("abcdefghijklmnopqrstuvwxyz", ch);
is_upper = index("ABCDEFGHIJKLMNOPQRSTUVWXYZ", ch);
```

Another bizarre idea that is more elegant and probably at around the same level of efficiency is to use PSL `toupper` to test lowercase and PSL `tolower` to test uppercase:

```
is_lower = toupper(ch) != ch;
is_upper = tolower(ch) != ch;
```

32.2.2 Reverse Testing Single Characters

In addition to the PSL `index` function, you also have the PSL `rindex` function. This function performs the character or string search in reverse, from the end of the string toward the beginning. We could just as easily have coded some of the above tests using the reverse searches:

```
is_whitespace = rindex(" \t\r\n", ch);
is_lower = rindex("abcdefghijklmnopqrstuvwxyz", ch);
```

32.2.3 Testing Strings Are Whitespace Only

The PSL `trim` function can be useful for testing whether strings contain only a particular character group. The idea is to remove these characters and see whether any other characters remain:

```
# Check whether string is all whitespace
function is_whitespace(str)
{
    return (trim(str, " \t\n\r") == "");
}
```

This idea can clearly be generalized to any character set:

```
# Check whether string is all from a set of characters
function is_characters(str, chars)
{
    return (trim(str, chars) == "");
}
```

32.3 Searching Strings

The PSL `grep` and `index` functions are the main functions used to search strings for text patterns. PSL `index` allows searches for plain text only. PSL `grep` allows fancy regular expressions, making it more powerful but usually less efficient. PSL `grep` also returns the matching lines, whereas PSL `index` returns the offset of the first match. Note that room for confusion exists, in that PSL `index` and `grep` have their search string and search pattern arguments in reverse order:

```
ret = grep(pattern, text);
ret = index(text, pattern);
```

Almost always, using PSL `index` is faster when performing a Boolean test (e.g., an “if” statement) as to whether a string contains a value. The following examples are equivalent, but the first is always faster if the text is a multi-line string:

```
if (index(text, "ERROR"))    # faster
if (grep("ERROR", text))    # slower
```

32.3.1 String Prefixes and Suffixes

The PSL `grep` function’s regular expressions make detecting particular prefixes and suffixes easy. Some code examples using `grep` with the special `^` and `$` regular expression characters:

```
if (grep("^ERROR", text)) # Found ERROR at start of line
if (grep("ERROR$", text)) # Found ERROR at end of line
```

A faster alternative for testing for a prefix is to use PSL `index` as follows:

```
if (index(text, "ERROR") == 1) # Found ERROR at start of line
```

We can also test for a suffix using the `rindex` PSL built-in function, which is the reverse of `index`. This allows code using PSL `rindex` and PSL `length` functions such as:

```
# Test for ".psl" suffix in filename
if (rindex(fname, ".psl") == length(fname) - 3) { . . . }
```

The magic number of 3 is one less than the length of the “.psl” suffix search string. The generalized code becomes:

```
function is_suffix(str, suffix)
{
    return (rindex(fname, ".psl")==length(fname)-length(suffix)+ 1);
}
```

Whether this use of `rindex` is faster than PSL `grep` is hard to say because of the additional calls to the `length` function. On small single-line strings, this function is probably worse than `grep`, but it is probably faster on larger multi-line strings.

32.3.2 Matching Exact Lines

The ideas for prefixes and suffixes can be generalized to match exact lines from the beginning to the end. An example using PSL `grep` with both `^` and `$` is simply:

```
if (grep("^ERROR$", text)) # Found ERROR line of text
```

Another equivalent piece of code is to use the set functions. Exact lines of text are considered to be members of a newline-separated set. This functionality allows the faster version:

```
if (subset(text, "ERROR")) # Found ERROR full line of text
```

32.3.3 PSL grep Compatibility

New functionality has been added to PSL `grep` in various agent releases. The changes all add new character flags to the third flags argument:

- 3.0: initial `grep` function added; “v” reverse test option available.
- 3.1: “t” text non-regular expression option added.
- 3.2: “i” ignore case option added; “b” bailout after first match option added; “c” count option added.

The purpose of all these PSL `grep` flags is as follows:

- “t”: Text matching rather than regular expression matching. Actually, the performance gain by using this method is negligible, owing to an internal optimization that does text searching automatically if no fancy regular expression characters are found.
- “v”: Reverse matching logic. Returns lines not matching the pattern.
- “c”: Counts the number of lines matching the pattern. Returns a number rather than lines.
- “b”: Bails out early after the first successful pattern match. Returns at most one line or a count of 1 if used with the “c” option.
- “i”: Ignores case of the characters when checking the match.

32.3.4 Searching Strings for Multiple Patterns

The PSL `grep` regular expression “|” character is the main method of searching for multiple patterns. Some simple examples are:

```
ret = grep("this\\|that", text); # Search for both patterns
ret = grep("^this$\\|^that$", text); # Search for 2 exact lines
```

The | character is preceded by two backslash (\\) characters. One is required because PSL `grep` requires this regular expression character to be escaped, and the second is required to get it past PSL tokenization during the compilation phase.

This feature of PSL `grep` gives rise to some common optimizations. One common use of multiple patterns is in searching an error log file for particular patterns. One implementation of this would be a simple `foreach` loop to perform multiple `grep` tests:

```
patterns = "ERROR1\\nERROR2"; # List of patterns newline-separated
foreach pattern (patterns) {
```



```

errors = grep(pattern, file_text);
    if (errors != "") {
        # Found matching error(s).. process it
        report_errors(errors);
    }
}

```

One way to remove the `foreach` loop is simply to change the newlines in the list of patterns into `\\|` sequences and use a single `grep` statement:

```

patterns = "ERROR1\\|ERROR2"; # List of patterns |-separated
errors = grep(patterns, file_text);
if (errors != "") {
    # Found matching errors .. process it
    report_errors(errors);
}

```

In some cases, the list still has to be newline-separated, such as when it is generated from some configuration settings. In this case, it can be automatically converted via `ntharg`. Here is the revised PSL code:

```

patterns = "ERROR1\nERROR2"; # List of patterns newline-separated
new_patterns = nthargf(patterns, "1-", "\n", "\\|"); # Make | list
errors = grep(new_patterns, file_text);
if (errors != "") {
    # Found matching errors .. process it
    report_errors(errors);
}

```

32.3.5 Character Case Handling

The main functions for treating upper- and lower-case are the PSL built-in functions `tolower` and `toupper`. For searching on the basis of character case, use also the “i” ignore case flag to PSL `grep` in 3.2 agents, but not in 3.1 or earlier.

```

text = grep("error", output_text, "i"); # Find all error messages

```

For compatibility with older releases, the following less efficient PSL code is possible:

```

# Portable version for 3.0/3.1 agents
text = grep("error", tolower(output_text)); # Find error messages

```

32.4 Removing Characters and Carriage Returns

All the PSL string operators and built-in functions assume that lines are separated by the new-line character. This character is represented by `\n` in PSL. This characteristic poses problems

for text processing on Windows NT, OS/2, and VMS agents. On these platforms, the output from commands or the contents of a text file may contain text separated by pairs of characters: carriage returns (`\r`) and newlines (`\n`).

The PSL `foreach` loop and functions such as `nthline` do not handle the `\r` character well. This fact can often lead to very confusing bugs, especially since `\r` is a hidden character when printing PSL variables. The simplest solution is simply to trim out the `\r` characters when receiving file or command text. The PSL `trim` function is simple to use:

```
text = trim(text, "\r"); # Remove \r characters for portability
```

This trimming of text to remove `\r` characters is desirable for the return values from the following PSL built-in functions: `read`, `cat`, `system`, and `execute`. Unfortunately, no automatic mode exists in PSL by which the agent will automatically do this.

32.4.1 Removing Whitespace and Other Characters

The PSL `trim` function is the main function used to remove whitespace. The way to remove annoying whitespace characters from a string is simply:

```
text = trim(text, "\t\r\n"); # Remove whitespace
```

Clearly, this can be generalized to remove any characters you might want to lose:

```
text = trim(text, "\r"); # Remove annoying carriage returns
text = trim(text,
    "abcdefghijklmnopqrstuvwxyz"); # Remove lowercase letters
text = trim(tolower(text),
    "abcdefghijklmnopqrstuvwxyz"); # Remove all letters (slower)
text = trim(text,
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"); # Remove all letters (faster)
);
```

32.4.2 Removing Leading Whitespace

No easy way is available to remove initial whitespace from a string. Unfortunately, the brute-force algorithm used below seems the only choice.

```
# Strip leading whitespace characters
function strip_leading_whitespace(str)
{
    local ch, i;

    i = 0;
    while (TRUE) {
        i++;
    }
}
```

```

        ch = substr(str,i,1); # Get first character
        if (ch == "" || !index("\r\n\t", ch)) {
            return substr(str, i, length(str));
        }
    }
}

```

This code can clearly be generalized to remove any characters. The string constant containing the whitespace characters need only be replaced with a different set of characters.

One attempt at a faster algorithm is to note that the PSL `ntharg` function ignores initial whitespace at the start of the string. The `ntharg` function does not consider it pertinent (although `nthargf` does). Therefore, `ntharg` can simply read the words again in a way that it discards the initial spaces. However, it replaces newlines and tabs with spaces, removes trailing whitespace, and discards multiple whitespace characters in the middle of the string. If these are not important, such as in a single-line string, here is a faster version of the above function:

```

function strip_leading_whitespace_fast(str)
{
    # This does **NOT** always work because the replacement
    # is always a single character space.
    # ntharg loses adjacent whitespace characters
    return ntharg(str, "1-", " \t\r\n");
}

```

32.4.3 Removing Trailing Whitespace

Trailing whitespace characters can be removed using a similar function. Apparently, nothing is better than a brute-force loop algorithm.

```

# Strip trailing whitespace characters
function strip_trailing_whitespace(str)
{
    local ch, i;

    i = length(str);
    while (i > 0) {
        ch = substr(str,i,1);
        if (!index("\r\n\t", ch)) {
            # Not whitespace, so we are finished...
            return substr(str,1, i);
        }
        i--;
    }
    # Got to start of string without finding non-whitespace!
    return str;
}

```

This code can clearly be generalized to remove any characters. The string constant containing the whitespace characters need only be replaced with a different set of characters. The faster version using `ntharg` for stripping leading whitespace also works to strip trailing whitespace, since it strips both. However, it has the same limitations, as already discussed.

32.5 Counting Characters and Text

The PSL `length` function is available to count the number of characters in a string. The usage is simply:

```
count = length(text); # Count number of characters
```

The number of a particular matching character can be testing by a reverse logic. The `trim` function can remove a particular character and count the number of lost characters. The following PSL code will count the number of whitespace characters:

```
# Count whitespace characters
count = length(text) - length(trim(text, " \r\n\t"));
```

32.5.1 Counting Lines

Counting lines is trivially achieved via the PSL “`lines`” function. This returns an integral number of lines separated by newlines:

```
count = lines(text); # Count number of lines
```

The PSL `grep` “`c`” option can be used to count lines matching a pattern in 3.2 agents:

```
count = grep(pattern, text, "c"); # Count matching lines (3.2)
```

The “`c`” option to `grep` is available only on 3.2 agents. One way to be backward compatible to 3.0/3.1 agents using slightly less efficient PSL code is to apply the PSL `lines` function to the output of a normal PSL `grep` function call:

```
count = lines(grep(pattern, text)); # Count matching lines, 3.0/3.1
```

32.5.2 Counting Words

Counting words has no simple PSL built-in function. One inefficient method would be to iterate through a variable with the PSL “`foreach word`” or “`ntharg`” statement and a counter. However, clever use of the `ntharg` function to convert whitespace to a newline and then apply the PSL `lines` function to count the number of lines allows more efficient word counting:

```
function word_count(str, delimiter)
{
```

```

# ntharg returns newline delimited return value
# Treats multiple delimiters as one, like ntharg
# i.e., this is not "nthargf"-like functionality.
return lines(ntharg(str, "1-", delimiter));
}

```

A variant of this uses the `nthargf` function instead of `ntharg`. This version differs on whether multiple delimiter characters have an empty zero-length word between them. The version below uses `nthargf` and counts these zero-length words, too:

```

function word_count_nthargf(str, delimiter)
{
    # ntharg returns newline delimited return value
    # Treats multiple delimiters as one, like ntharg
    return lines(nthargf(str, "1-", delimiter));
}

```

32.6 Converting Characters

The need to convert one character into another occasionally arises. For example, when dealing with filenames, you might want to convert all “/” characters to “\” characters. Another issue is that the PSL objects cannot contain the “/” character, and you might want to convert them to “-” or “_” characters, which are permitted in PSL object names. The versatile PSL `nthargf` function can be used for this purpose. The generalized idea of using `nthargf` to translate characters becomes:

```

function char_translate(target, char_source, char_dest)
{
    # Notes: convert all the characters to another character
    # Note: source string can NOT be more than one character
    # Note: destination string can be more than one character
    return nthargf(target, "1-", char_source, char_dest);
}

```

An example of converting “/” into “-” characters for use as instance object names to PSL create via `nthargf` appears below. This has the special feature of ignoring the first character if it is a forward slash.

```

function slash2dash(target)
{
    # If first character is /, then remove it.
    # Only changes all but the first character.

    if(substr(target, 1, 1) == "/"){
        # Remove first character
    }
}

```

```

        target = substr(target,2,length(target));
    }
    target = nthargf(target, "1-", "/", "-");
    return(target);
}

```

Note that you can use `ntharg` instead of `nthargf` if you want sequences of adjacent characters to be replaced by a single other replacement character.

32.6.1 Converting Strings, Numbers, and Hexadecimals

Nothing is difficult to say about conversions involving strings and numbers in PSL. Both of these data types are represented as strings. If the strings contain decimal values, you have nothing to convert. Simply use the string variable as if it were a number.

One caveat is also discussed in the PSL Reference Manual. No non-numeric characters must appear in a string value. If wrong characters appear, they may cause the PSL operators to perform string rather than numeric operations, such as relational comparisons (e.g., `<`, `>`, `==`, `!=`). One solution is simply to remove non-numeric characters from the strings via PSL `trim`.

You can easily convert decimal to hexadecimal values via PSL `sprintf`. A simple example is via the `%x` format to PSL `sprintf`.

```
hex = sprintf("%x", number);
```

However, you cannot easily perform the reverse conversion in PSL. Following is a brute-force algorithm using PSL string operations to perform the base conversion. Clearly, the base variable could be set to a number other than 16 to change the algorithm to another base, such as octal or binary.

```

function hex_to_decimal(hexnum)
{
    local result, mult, col, base;

    result = 0;
    sign = index(hexnum, "-");
    hexnum = trim(hexnum, "-+");
    col = length(hexnum);
    mult = 1;
    base = 16; # Hexadecimal

    while (hexnum != "") {
        digit = substr(hexnum, col, 1); # Get the right digit
        offset = index("0123456789abcdef", tolower(digit));
        result = result + ((offset - 1) * mult);
        # Prepare for next iteration
        col--;
    }
}

```

```

        hexnum = substr(hexnum, 1, col); # Remove rightmost digit
        mult = mult * base;
    }
    if (sign) { return ( -1 * result); }
    return result;
}

```

32.7 Removing Lines and Text

The PSL `nthlinef` function is used for extracting lines by their number. Some simple examples are:

```

ret = nthlinef(text, 1); # Cut first line
ret = nthlinef(text, "1,5,8-10"); # Get lines 1, 5, and 8-10

```

The PSL `nthlinef` function is almost always superior to `nthline`. The `nthline` function should be used only if you want to ignore blank lines in the count. The `nthline` function differs from `nthlinef` only in its handling of multiple adjacent newline separator characters. Note that trimming out `\r` characters is an issue for both `nthline` and `nthlinef`.

32.7.1 Reverse Removing Lines

The PSL `nthlinef` function counts line numbers from the beginning. Sometimes you want to extract lines from the end of a block of text. For example, this might be the end of an error log file with the most recent error messages. One way to get the end lines of a block of text is to use the PSL `tail` function.

```

ln = tail(text, 1); # Get the last line of text

```

Note that PSL `tail` is an internal string function. It does not launch any external executable. The semantics are similar to the UNIX `tail` command, but it does not launch this command on UNIX agents. This is an efficient internal string operation only.

The use of PSL `tail` works only if you want a fixed number of lines at the end. It does not work, for example, if you want to get the second-last line of text, but not the first. You would have to do the following:

```

# Get second-last line of text
ret = tail(text, 2); # Get last two lines
ret = nthlinef(lns, 1); # Get 2nd-last line

```

An alternative is to use some fancy arithmetic via the PSL `lines` function in combination with `nthlinef`. The code would be similar to:

```

count = lines(text);
ln = nthlinef(text, count - 1); # Get the second-last line

```

The efficiency of both these sequences is comparable. No major difference in performance occurs, because all these operations are internal string operations.

32.7.2 Removing Lines by Patterns

The “v” option to PSL `grep` is available to remove lines based on regular expression patterns.

```
text = grep(pattern, text, "v"); # Remove lines with a pattern
```

The PSL `difference` function can also be used as a more efficient method of removing an entire line (i.e., not just a pattern that is part of the line).

```
text = difference(text, full_line);
```

This idea is occasionally useful for advanced string handling. PSL `difference` can be used this way for associative arrays.

32.7.3 Removing Blank Lines

The inefficient way of removing blank lines (i.e., lines with not even whitespace on them) is via PSL `grep`:

```
text = grep("^$", text, "v"); # Remove blank lines
```

The efficient method is to use the PSL `difference` function:

```
text = difference(text, "\n"); # Remove blank lines
```

Note that both these methods may be hampered by `\r` characters if these are not removed first by PSL `trim`.

32.7.4 Removing Whitespace-Only Lines

The PSL `difference` function cannot work for the more general case of removing not only blank lines, but also lines with some whitespace characters. The removal of whitespace-only lines can be performed by PSL `grep` using a simple regular expression:

```
text = grep("^[ \t]*$", text, "v"); # Remove whitespace lines
```

32.8 Cutting Columns

The PSL `ntharg` function operates on multi-line strings for the exact purpose of cutting out particular columns. Some of the simpler forms include:


```
ret = ntharg(ln, 1); # Get first value on line
ret = ntharg(nthlinef(ln, 2), 3); # Get 3rd value on 2nd line
```

However, `ntharg` is far more powerful because it can operate on multi-line strings. This fact makes `ntharg` a very powerful function that can be used for many string processing needs. Some simple examples include:

```
col1 = ntharg(text, 1); # Cut column 1
cols = ntharg(text, "1-4,7"); # Cut columns 1-4 and 7
```

The returned string from `ntharg` is a newline-separated list that matches line for line and has each column separated by the separator character. This is perfect for cutting whole columns. The same multi-line behavior applies to `nthargf`. The main difference between `ntharg` and `nthargf` is in their handling of adjacent multiple separators. For example, in the simple case, this is multiple adjacent whitespace characters. The `ntharg` function treats these as a single column separator, whereas `nthargf` treats these as indicating empty columns between the separators. Also, different handling of initial whitespace separator characters occurs, in that `ntharg` ignores them, whereas `nthargf` treats them as indicating an initial empty column. The choice of `ntharg` and `nthargf` depends entirely on the behavior that is required of the code. In the majority of cases, multiple spaces can be treated as one (use `ntharg`), but in some cases a formal layout permits empty columns (use `nthargf` here). An example of the requirement to use `nthargf` is in a comma-separated data file format where multiple adjacent commas always separate empty fields.

32.8.1 Reverse Column Cutting

The following functions show how to perform a form of reverse argument cutting with features similar to `ntharg` and `nthargf`. Only the simple formats such as a single column number are permitted.

```
function reverse_ntharg(str, position, delimiter)
{
    local ret;

    # count is the number of delimited "words"
    if (isnumber(position) && !index(position, "-")) {
        # If a single number like "2" rather than using "-" notation
        # that ntharg allows as in 1-, -9, and 2-3, or commas.
        # No dashes and no commas in the pattern here:

        # ntharg turns delimiters into newlines, for easy counting
        ret = ntharg(str,"1-", delimiter);
        return trim(nthlinef(ret, lines(ret) - position + 1), "\n");
    }
}
```

```

    else {
        # General case of ntharg 'position' argument
        print("ERROR: general format unsupported.\n");
        return "";
    }
}

```

The version using `nthargf` is similar code:

```

function reverse_nthargf(str, position, delimiter)
{
    local ret;

    # count is the number of delimited "words"
    if (isnumber(position) && !index(position, "-")) {
        # If a single number like "2" rather than the "-" notation
        # that nthargf allows as in 1-, -9, and 2-3, or commas.
        # No dashes and no commas in the pattern here:

        # nthargf turns delimiters into newlines, for easy counting
        ret = nthargf(str, "1-", delimiter);
        return trim(nthlinef(ret, lines(ret) - position + 1), "\n");
    }
    else {
        # General case of nthargf 'position' argument
        print("ERROR: general format unsupported\n");
        return "";
    }
}

```

32.8.2 Cutting Columns by Character Position

No simple method of cutting entire columns of text on the basis of character positions is available. The `ntharg` function works only on column number, and not on character number. The PSL `substr` function works on character position, and the only solution appears to be a loop repeatedly using `substr` on every line of a block of text.

32.9 Sorting Text

The PSL `sort` function exists specifically to sort lines of text. The various options control the order of the sort (ascending or descending) and the type of comparison (numeric or string). This feature allows text to be sorted according to requirements.

```

text = sort(text);    # Sort in ascending alphabetical order
text = sort(text, "rn"); # Sort descending (reverse) numeric order

```

Also, a third argument to `PSL sort` specifies the character position at which the string is to be considered.

32.9.1 Sorting Text by Column Number

Sorting by column number is desirable when dealing with array-like representations of data. The `PSL sort` function has an option to sort on the basis of character position. A more useful approach often is to sort on the basis of column number, as used by the `ntharg` functions. However, no built-in support is available in `PSL sort` or any other function to fill this need.

One possible workaround would be to re-arrange the columns via `PSL ntharg` to bring the sorting column to the front. Unfortunately, although you can extract columns with `ntharg`, you have no way to paste columns together, and the idea fails for that reason.

32.10 Separating Lines by Patterns

A common string processing issue is to break a long command output at particular patterns. For example, when sending queries to a global channel, we want to find all the text up to a particular pattern that indicates the query is finished and a new output is beginning.

To get the string data up to a particular non-regular expression pattern is relatively easy. The `PSL index` function returns the offset location of the pattern, and the `substr` function can extract the desired text before that position.

```
function lines_up_to_pattern(str, pattern)
{
    local i;

    # Return all lines up to but not including a string
    # e.g., used to separate transactions from a global channel
    # Return does not include the pattern string
    i = index(str, pattern);
    if (i > 1) {
        return (substr(str, 1, i - 1));
    }
    else { return "";
    }
}
```

This code example extracts the first pattern. The more general case is a loop to process multiple text blocks separated by a line pattern.

This code works only for a plain text pattern. To separate lines on the basis of a full regular expression is much more difficult, because the `PSL index` function does not support this, and `PSL grep` cannot return the character or line number position of the pattern.

32.11 Date and Time String Representations

String handling can also help in dealing with dates and times. Two main representations of times are used in PSL: integral timestamp and string representations. Various PSL functions deal with both of these representations (see Table 32-2).

Table 32-3 PSL date and time functions

Function	Purpose
asctime	Convert timestamp integer to string time representation
convert_date	Convert string time representation to timestamp integer
date	Current time in string representation
time	Integral timestamp for the current time

Note that the PSL `convert_date` function was added to the 3.1 agent release, and is not supported by PSL in 3.0 agents.

Bibliography

- Aldrich, Susan E., *Web Discipline*, Distributed Computing Monitor, Vol. 11, No. 8, Patricia Seybold Group, April 1996.
- Allen, N., *The I₂O Initiative Promises Better I/O and Network Storage*, Data Center Strategies, Gartner Group, March 1997.
- Bair, J., and Brethenoux, E., *Agent Glut: A Solution Becomes a Problem*, Research Note, Gartner Group, May 1997.
- BMC Software, "PATROL Lowers the Cost of Managing SAP R/3," BMC Software, 1997.
- BMC Software, "PATROL and I₂O: Applications Management and the Impact on Underlying Technologies," BMC Software white paper, 1997.
- BMC Software, "Industry Overview and Trends," BMC Software, October 1996.
- BMC Software, Recreational Equipment Inc., and Open Systems Management, "The PATROL and COSMOS Products Reach the Summit at REI," Customer Success Stories, 1997.
- BMC Software and First State Computing, "FM for Open Systems," Business Profile, 1997.
- BMC Software, "PATROL architecture: Raising the application management standard," 1996.
- BMC Software, "PATROL architecture: Raising the application management standard (PATROL 3.2 updates)," 1997.
- BMC Software, "Proactive Application Management for SAP R/3," BMC Software white paper, 1997.
- BMC Software, "Cooperative Enterprise Management Solutions," BMC Software presentation, 1996.
- BMC Software, "Why the leading industry vendors are selecting the PATROL Management Suite," 1996.
- BMC Software, Associated Octel, "Managing The Reaction: How Associated Octel moved from a mainframe to a UNIX environment," Business Profile, 1996.
- BMC Software, Royal Holloway University, "Managing Information," Business Profile, 1996.
- BMC Software, Energis, "Managing Multiple Applications," Business Profile, 1996.
- BMC Software, HM Customs and Excise, "Breaking New Ground In A Distributed Environment," Business Profile, 1996.
- BMC Software, HBO and Company, "Staying Healthy," Business Profile, 1996.
- BMC Software, Abbey Life, "Moving Towards A Client/Server Environment," Business Profile, 1996.
- BMC Software, "Enterprise-Wide Analysis of Network-Based Application Performance," 1997.

- BMC Software, "PATROL. The Management Solution Chosen By More Industry Vendors," 1997.
- BMC Software, "Maximum Internet/Intranet Power," 1997.
- BMC Software, "Establishing Requirements for Application and Data Management," 1996.
- BMC Software, "Features and Benefits of PATROL," 1997.
- BMC Software, "XYZ Corporation: Enterprise Management Evaluation Checklist," 1997.
- BMC Software, "PATROL Application Management Overview," 1997.
- BMC Software, "PATROL Overview," 1997.
- BMC Software, "Automating Application Management: An Overview of PATROL," 1994.
- Borden, Jay, "HP to Become the Leading Framework Manager Within 12-18 Months," Giga Group, 1996.
- Burleson, Don, "PATROL Version 3.0.15," DBMS, April 1996.
- CCTA, "Service Level Management," IT Infrastructure Library, CCTA, London, 1996.
- CCTA, "Availability Management," IT Infrastructure Library, CCTA, London, 1996.
- CCTA, "Problem Management," IT Infrastructure Library, CCTA, London, 1996.
- Client/Server Computing, "DBA Tools Target Recurring Problems," January 1997.
- Colville, R., "Q & A on Service Desk Technologies, Part 2," Gartner Group, 1997.
- Cummins Engine, "Driving Network Management," Communications News, August 1996.
- Davis, Paul J., "The Systems Management Challenge: Getting Control of Enterprise Computing (The Internet, Intranets and Internetworking)," Robertson Stephens & Company, December 1996.
- Davis, Judith R., "Strategies for Managing Client/Server Databases," Open Information Systems, Vol. 11, No. 4, Patricia Seybold Group, April 1996.
- Feinberg, Donald, "New Age Database Management," presentation, Gartner Group, 1997.
- Foote, Steve, "The Application Management Wars," The Hurwitz Report, Hurwitz Group, July/August 1997.
- Foote, Steven, "Managing Applications," DBMS, October 1997.
- Fox, Ric, "Why Applications Management?," Tivoli Systems, 1997.
- Germann, C., Cone, K., Welch, M., Cunningham, A., Truglio, M., and Cicalo, P., "U.S. SWAMI '97: Architects for Change," Top VIEW, TV-000-404, Gartner Group, June 27 1997.
- Gliedman, Chip, "Enterprise Systems Management Spans Many Functions," Presentation from GigaWorld, Giga Group, June 1997.
- Gliedman, Chip, "Does BMC solve the problem of managing heterogeneous databases?," Giga Group, 1996.
- Gliedman, Chip, "From Frameworks to Suites: The Next Wave in Distributed Systems Management," Giga Group, May 1997.
- Greenemeier, Larry, "BMC Patrols Web Servers," Midrange Systems, June 13 1997.
- Hildebrand, Dr. Dietmar J., "Intelligent Agents as Part of the Enterprise Management Strategy," BMC Software, 1997.
- Howard, Brian, "PATROL Helps Manage Distributed Environment at Acxiom," DM Review, June 1997.
- Hurwitz Balanced View, "HP and Tivoli Announced Performance Management API," July 1996.
- Hurwitz Group, "Event Management Trends for 1998, Part One: Self-Tuning Agents, Event Horizons, Alarm Showers, and Event Profiles," 1997.
- Hurwitz Group, "Preventing Application Meltdown: Checklist of Solutions," Hurwitz Balanced View Research Bulletin, 1996.
- Hurwitz Group, "Managing the Software Deployment Process," Hurwitz Balanced View Research Bulletin, 1996.
- Hurwitz Group, "Applications Management: It's Not Just Technology: Organizational Issues in Developing an Applications Management Strategy," Hurwitz Balanced View Research Bulletin, 1996.

- Jeffress, Jeff, "PATROL Custom Integration," presentation, BMC Software, February 1997.
- Lowe, Doug, *Client/Server Computing for Dummies*, IDG Books, 1995.
- Mcnee, B., "Which IT Trends and Directions Are Key?," Gartner Group, 1997.
- McIntyre, Kim, "Year 2000 Compliance: Which Solution Will You Take?," techNEWS, BMC Software, Jan-March 1997.
- Malik, W., "Riding the Whirlwind: The Transformational Effects of Managing Distributed Systems," Gartner Group, 1997.
- Marshak, David S., "Mission-Critical Lotus Notes," Prentice Hall, 1996.
- Mattini, Max, and Hargrove, Dan, "Postcards from the Edge—Remote Event Logging in Networking Environments," techNEWS, BMC Software, April-June 1997.
- Martin, Richard J., and Weadock, Glenn E., *Bullet Proofing Client/Server Systems*, McGraw Hill, 1997.
- Microsoft, "Microsoft Management," Microsoft white paper, 1997.
- Parker, Jeff, "PATROL Implementation Guide," BMC Software, 1997.
- Parker, Jeff, "Beginning Policy for Enterprise Management—Getting Beyond the Baseline of PATROL," techNEWS, BMC Software, April-June 1997.
- Parker, Jeff, "Successfully Implementing PATROL: A Proven Approach," techNEWS, BMC Software, July-Sept. 1997.
- Price, C., "WBEM: A Standard in the Making," Gartner Group, 1997.
- Price, C., and Paquet, R., "WBEM: A Standard in the Making," Research Note, Strategic Planning, SPA-INT-315, Gartner Group, October 29, 1996.
- Salemi, Joe, *Guide to Client/Server Databases (second edition)*, Ziff-Davis Press, 1995.
- Schulte, R., "Middleware: One Size Does Not Fit All," Research Note, Gartner Group, April 1997.
- Schulte, R., "Overview of Six Modern IT Architectures, Part 1," Gartner Group, 1997.
- Schulte, R., "Overview of Six Modern IT Architectures, Part 2," Gartner Group, 1997.
- Scott, Donna, "BMC Patrol: A Tutorial," Research Note, Gartner Group, April 1997.
- Scott, Donna, "BMC Patrol: Ready to Cover More Ground?," Research Note, Gartner Group, April 1997.
- Scott, Donna, "IBM/Tivoli—Post Merger, Part 2," Gartner Group, 1997.
- Scott, Donna, "Positioning Unicenter and TME 10: Hype vs. Reality, Part 1," Gartner Group, 1997.
- Scott, Donna, "Positioning Unicenter and TME 10: Hype vs. Reality, Part 2," Gartner Group, 1997.
- Scott, Donna, "Client/Server Applications Management," Gartner Group, 1997.
- Scott, Donna, "Management Client/Server: Infrastructure vs. Application Management," Top VIEW Research Notes TV-000-287, Gartner Group, July 26 1996.
- Spuler, David, "PATROL SNMP Trap Notification and Integration Issues," BMC Software, 1997.
- Spuler, David, "PATROL 2.0/3.0 Agent Architecture," BMC Software, 1995.
- Spuler, David, "PATROL Large-Scale Deployment Guide," BMC Software, 1996.
- Spuler, David, "PATROL NFS Deployment Issues," BMC Software, 1996.
- Spuler, David, "PATROL Backward Compatibility Issues," BMC Software, 1996.
- Spuler, David, "PATROL Migration/Upgrade Issues," BMC Software, 1996.
- Spuler, David, "PATROL 3.0 Architecture," BMC Software, 1995.
- Spuler, David, "PATROL Fallback Console," BMC Software, 1996.
- Spuler, David, "PATROL Dial-Up Issues," BMC Software, 1996.
- Spuler, David, "PATROL Reliable UDP," BMC Software, 1996.

- Spuler, David, "PATROL Standards Issues," BMC Software, 1996.
- Spuler, David, "Achieving Total Management Coverage via Custom Application Management," presentation, BMC Software, 1997.
- Spuler, David, "PATROL Integration Points," techNEWS, BMC Software, July-Sept. 1996.
- Spuler, David, Kushner, Jeff, and Seppi, Kevin, "Empowering CEMS Via Agent Technology," techNews, BMC Software, 1996.
- Spuler, David, "Agent Query Tool on the Desktop," techNEWS, BMC Software, July-Sept. 1997.
- Spuler, David, Jones, Steve, and Fisher, Steve, "Proxy KM Architectures," techNEWS, BMC Software, Oct-Dec 1997.
- Spuler, David, "Anticipating Web-Based Enterprise Management," BMC Software white paper, February, 1997.
- Spuler, David, and Rios, William, "PATROL Pager and Email Notification Issues," BMC Software, 1997.
- The, Lee, "Patrolling Database Backups," Datamation, November 15, 1995.
- UniForum, "The MOSES Whitepapers: The Collected Whitepapers from the Massive Open Systems Environment Standard," revision 2.0, UniForum Technical Publication, 1994, ISBN #0-936593-29-6
- Wilson, Tim, "The BMC Factor: Making the Match for Applications Management," Data Communications OpenView Advisor, January 1996.
- Young, Tim, "Intelligent Agents in Systems Management: A Developer's Experience," BMC Software, 1997.
- Young, Tim, "Application Management: Commonly Asked Questions and Answers," BMC Software, June 1997.