Efficient and Portable Middleware for Application-level Adaptation

Deepak Rao

Department of Computer Science and Applications

Virginia Tech

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

Dr. Richard E. Nance, Chair

Dr. Benjamin J. Keller

Dr. Scott F. Midkiff

16th May 2001

Blacksburg, Virginia

Keywords: Application-aware Adaptation, Adaptation, Mobile Computing, Middleware

Copyright 2001, Deepak Rao

Efficient and Portable Middleware for Application-level Adaptation

Deepak Rao

(ABSTRACT)

Software-intensive systems operating in a shared environment must utilize a "request, acquire and release" protocol. In the popular client-server architecture resource-poor clients rely on servers for the needed capabilities. With mobile clients using wireless connectivity, the disparity in resource needs can force the consideration of adaptation by clients, leading to a strategy of self-reliance. Achieving self-reliance through adaptation becomes even more attractive in environments, which are dynamic and continually changing. A more comprehensive strategy is for the mobile client to recognize the changing resource levels and plan for any such degradation; that is, the applications in the mobile client need to *adapt* to the changing environment and availability of resources.

Portable adaptation middleware that is sensitive to architecture and context changes in network operations is designed and implemented. The Adaptation Middleware not only provides the flexibility for the client applications to adapt to changing resources around them, but also to changing resource levels within the client applications. Further, the Adaptation Middleware imposes few changes on the structure of the client application. The Adaptation Middleware creates the adaptations; the client remains unaware and unconcerned with these adaptations.

The Adaptation Middleware in this study also enables a more informative cost estimation with regard to applications such as mobile agents. A sample application developed using the Adaptation Middleware shows performance improvements in the range of 31% to 54%. A limited set of experiments show an average response time of 68 milliseconds, which seems acceptable for most applications. Further, the Adaptation Middleware permits increased stability for applications demonstrating demand levels subject to high uncertainty.

ACKNOWLEDGEMENTS

I thank Dr. Richard Nance for guidance and support in making this thesis possible. Without his support and his insights into problems, this thesis would not have seen the light of the day, I am grateful for his invaluable help. I take this opportunity to thank Dr. Keller and Dr. Midkiff for serving as members of my committee. I also thank Dr. Arthur for his suggestions and encouragement to make this thesis possible.

I would like to sincerely thank Dr. M. Satyanarayanan, Carnegie Mellon University, for laying the foundations in the field of application-aware adaptation. I also thank Dr. Brian Noble, University of Michigan, and Dr. Aline Baggio, Vrije Universiteit (Amsterdam), for their suggestions and comments on application-level adaptation.

I take this opportunity to thank Mr. Vikas Aggrawal and Mr. Balaji Krishnamachari-Sampath for their enthusiasm to use the adaptation framework and their help in developing the Tuple Space Manager application.

I am also indebted to my parents and the rest of my family for all the support they provided me. Last, but not the least, I thank all my friends and all others who have directly or indirectly helped me in my efforts.

TABLE OF CONTENTS

ABSTRACT .... **II**

ACKNOWLEDGEMENTS .... **III**

LIST OF FIGURES .... **VIII**

LIST OF ABBREVIATIONS .... **X**

CHAPTER 1 INTRODUCTION **1**

1.1 QoS vs. Adaptation 2

1.2 Types of adaptation strategies 3

1.3 Description of research 4

CHAPTER 2 LITERATURE REVIEW **6**

2.1 Operating system support for adaptation 6

2.1.1 Coda: Distributed file system *6*

2.1.1.1 Connected operations 7

2.1.1.2 Disconnected operations 7

2.1.2 Odyssey: Support for application-aware adaptation *7*

2.1.2.1 Structure of Odyssey 8

2.1.2.2 Odyssey API 8

2.1.2.3 Extensions to Unix for supporting Odyssey API 9

2.2 Support for adaptation from the middleware level: Cadmium 11

2.2.1 Objects and references *13*

2.2.2 Binding *13*

2.3 Design of adaptive applications 14

CHAPTER 3 ARCHITECTURAL-LEVEL DESIGN **17**

3.1 Classification of resources 17

3.2 Ideal adaptive middleware software architecture 20

3.2.1 Components of an adaptation middleware *20*

3.2.2 Effects of resources on sub-systems *21*

3.2.3 Constraints in implementing the ideal architecture *22*
3.3 THE REVISED SOFTWARE ARCHITECTURE 22

CHAPTER 4 COMPONENT-LEVEL DESIGN 25

4.1 STRUCTURE OF THE ADAPTATION MIDDLEWARE 25
4.2 MONITORING SERVICE 26
4.2.1 Processing narrative for Monitoring Service *26*
4.2.2 Description of interfaces in Monitoring Service *27*
4.2.3 Processing details of Monitoring Service *28*
4.2.3.1 Interface description 29
4.2.3.2 Algorithmic model for Passive Monitoring Service 32
4.2.3.3 Restrictions or limitations of Monitoring Service 33
4.2.3.4 Data structures used in Monitoring Service 33
4.2.3.5 Performance issues in Monitoring Service 33
4.2.3.6 Design constraints of Monitoring Service 34
4.3 FEEDBACK COMPONENT 34
4.3.1 Processing narrative for Feedback Component *34*
4.3.2 Description of interfaces in Feedback Component *35*
4.3.3 Processing details of the Feedback Component *37*
4.3.3.1 Interface description 37
4.3.3.2 Algorithmic model for Feedback Component 39
4.3.3.3 Restrictions or limitations in Feedback Component 40
4.3.3.4 Data structures used of Feedback Component 40
4.3.3.5 Performance issues of Feedback Component 41
4.3.3.6 Design constraints of Feedback Component 42
4.4 POLICY MANAGER 42
4.4.1 Processing narrative for Policy Manager *42*
4.4.2 Description of interfaces in Policy Manager *43*
4.4.3 Processing details of Policy Manager *45*
4.4.3.1 Interface description 45
4.4.3.2 Algorithmic model for Policy Manager 48
4.4.3.3 Restrictions or limitations in Policy Manager 49

4.4.3.4 Data structures used in Policy Manager 49
4.4.3.5 Performance issues of Policy Manager 50
4.4.3.6 Design constraints of Policy Manager 50
4.5 FLEXIBILITIES IN DESIGN 50

CHAPTER 5 PERFORMANCE EVALUATION 52

5.1 INTRODUCTION TO THE MONTE CARLO SIMULATION TECHNIQUE 52
5.2 DUAL PERSPECTIVES 53
5.2.1 Application perspective 53
5.2.2 System perspective 53
5.3 TUPLE SPACE MANAGER: A SAMPLE APPLICATION 54
5.3.1 Tuple Space Manager 54
5.3.2 Types of policies 55
5.3.3 Types of policy control mechanisms 55
5.4 APPLICATION PERSPECTIVE 56
5.4.1 Performance improvements of application with policy management 58
5.4.2 Comparison of performance for different policy control mechanisms 58
5.5 SYSTEM PERSPECTIVE 59

CHAPTER 6 CONCLUSIONS, SUMMARY AND FUTURE WORK 63

6.1 CONCLUSIONS 63
6.2 SUMMARY 63
6.3 FUTURE WORK 64
6.3.1 Adaptation middleware 64
6.3.1.1 On-the-fly introduction of policy control mechanisms 65
6.3.1.2 Hot-swapping of policies 65
6.3.2 Application-level adaptation 66
6.3.2.1 Domain-specific and domain-independent architectures 66
6.3.2.2 Cooperative adaptation 66

REFERENCES 68

APPENDIX A: REPORT ON THE UML DIAGRAMS 70

APPENDIX B: RESULTS OF PERFORMANCE EVALUATION 79

APPENDIX C: FRAMEWORK FOR AN ADAPTIVE APPLICATION 86

LIST OF FIGURES

FIGURE 1: RANGE OF ADAPTATION STRATEGIES. 3
FIGURE 2: ARCHITECTURE OF ODYSSEY. 8
FIGURE 3: HIERARCHICAL TOMES IN ODYSSEY. 10
FIGURE 4: CADMIUM DESIGN OVERVIEW. 12
FIGURE 5: A LAYERED VIEW OF AN ADAPTIVE APPLICATION. 15
FIGURE 6: CLASSIFICATION OF TYPES OF RESOURCES. 18
FIGURE 7: COMPUTING AND COMMUNICATING RESOURCES. 19
FIGURE 8: IDEAL PORTABLE ADAPTATION MIDDLEWARE SOFTWARE ARCHITECTURE. 21
FIGURE 9: REVISED PORTABLE MIDDLEWARE SOFTWARE ARCHITECTURE SUPPORTING APPLICATION-LEVEL ADAPTATION. 23
FIGURE 10: SUB-SYSTEM COLLABORATION DIAGRAM. 24
FIGURE 11: PLATFORM DEPENDENT MONITORING OF SYSTEM RESOURCES. 27
FIGURE 12: CLASS DIAGRAM FOR THE PASSIVE MONITORING SERVICE. 28
FIGURE 13: INTERACTION DIAGRAM — PART 1 — OF THE MONITORING SERVICE. 29
FIGURE 14: INTERACTION DIAGRAM — PART 2 — OF THE MONITORING SERVICE. 30
FIGURE 15: CLASS DIAGRAM OF FEEDBACK COMPONENT. 36
FIGURE 16: INTERACTION DIAGRAM OF THE FEEDBACK COMPONENT. 37
FIGURE 17: INTERACTION DIAGRAM FOR REGISTERING AND DEREGISTERING FOR RESOURCE CHANGE NOTIFICATIONS FROM THE FEEDBACK COMPONENT. 38
FIGURE 18: CLASS DIAGRAM OF POLICY MANAGER. 44
FIGURE 19: INTERACTION DIAGRAM TO REGISTER AND DEREGISTER POLICIES. 46
FIGURE 20: STATE DIAGRAM DEPICTING POLICY-CONTROL-MECHANISM-1 56
FIGURE 21: STATE DIAGRAM DEPICTING POLICY-CONTROL-MECHANISM-2 56
FIGURE 22: RATIO OF NUMBER OF TUPLES OF A PARTICULAR TYPE TO THE TOTAL NUMBER OF TUPLES OF ALL TYPES IN A SITE. 57
FIGURE 23: RATIO OF NUMBER OF REQUESTS GENERATED FOR TUPLES OF A PARTICULAR TYPE TO THAT FOR THE TOTAL NUMBER OF TUPLES OF ALL TYPES IN A SITE. 58
FIGURE 24: AVERAGE NUMBER OF TUPLES FETCHED OUTSIDE THE TSM. 59

FIGURE 25: RATIO OF POLICY CHANGES TO RESOURCE CHANGES FOR EXPONENTIAL DISTRIBUTION WITH MEAN = 1. 61
FIGURE 26: RATIO OF POLICY CHANGES TO RESOURCE CHANGES FOR NORMAL DISTRIBUTION WITH MEAN = 10 AND STANDARD DEVIATION = 5. 61
FIGURE 27: RATIO OF POLICY CHANGES TO RESOURCE CHANGES FOR UNIFORM DISTRIBUTION IN THE RANGE 70...90 61

LIST OF ABBREVIATIONS

AMS	Active Monitoring Service
API	Application Programming Interface
BSD	Berkeley Software Design
CMU	Carnegie Mellon University
CPU	Central Processing Unit
FC	Feedback Component
HWA	High Watermark Array
JVM	Java™ Virtual Machine
LDAP	Lightweight Directory Access Protocol
LWA	Low Watermark Array
MS	Monitoring Service
NDS	Netscape Directory Service™
PM	Policy Manager
PMC1	Policy-Control-Mechanism-1
PMC2	Policy-Control-Mechanism-2
PMS	Passive Monitoring Service
QoS	Quality of Service
RMI	Remote Method Invocation
TCP	Transmission Control Protocol
TCP / IP	Transmission Control Protocol / Internet protocol
TSM	Tuple Space Manager

CHAPTER 1 INTRODUCTION

This chapter provides a brief introduction to the field of application-level adaptation. The chapter argues why resource reservations may not be preferable in all cases; in particular, it compares a resource reservation policy in networks — QoS — with Adaptation. The chapter then describes the broad classification of types of adaptation. Finally, the chapter concludes with a description of the objectives of our research.

Typically, the design of a mobile client is resource-poor, compact and lightweight compared to a desktop client. Further, network connectivity through the wireless media is always error-prone and degraded in terms of available bandwidth, latency, reliability and cost. Also, power management issues like conservation of battery power requires many client actions to be avoided, slowed, or deferred. Finally, mobility of clients brings in new issues like lack of robustness and security.

As a consequence, the clients need to be adaptive to the changing environment around them and also to the changing resource levels within them. Adaptability is possible at various levels:

1. Physical and data link level: the adaptation mechanism can lead to appropriate channel selection, power control and error detection.
2. Network layer: routing methods should adapt to mobility; an example is the Mobile IP standard.
3. Transport layer: the protocols in this layer must distinguish between the wireless links and the wired links. They must further protect against the high error rates of the wireless links.
4. Application layer: the applications in this layer must adapt to varying bandwidth. Real-time applications, in particular the multimedia applications, must adapt to the variations in delay.

Adaptation behavior in the application layer is absolved from the adaptation to mobility of the wireless clients. Adaptation support for mobility is a network layer issue and not an

application layer issue. Hence, this thesis focuses in adapting the behavior of a wireless client to varying bandwidth, battery power and latency.

1.1 QoS vs. Adaptation

Quality-of-Service (QoS) refers to the packet delivery guarantees provided by the network architecture. Usually these are performance-related guarantees like bandwidth and delay. In mobile environments, even though the requested resources such as bandwidth can be available during the connection setup time, they can become unavailable due to the mobility of clients or greater interference in wireless communication. In such situations the applications must *adapt* to the changed environment.

Applications in mobile clients must be both QoS-aware and adaptive. One way of extending the QoS support to the mobile environment is to split the network, into the fixed part and the mobile part, and apply QoS support to each independently. With independent QoS support, the existing QoS support between the server and the base station can remain unchanged, while new QoS parameters and adaptation policies can be applied to the wireless link.

Traditional QoS parameters to capture the unpredictability in the mobile environment do not exist. New QoS parameters that need to be defined to capture this unpredictability are *graceful degradation of service, loss profile* and *probability of seamless communication* [6] .

- *Graceful degradation of service:* a characterization of the degree of degradation due to mobility.
- *Loss profile:* a specification of the order in which data can be discarded due to the inadequate bandwidth.
- *Probability of seamless communication:* the requirement of seamless connection stated in terms of probability (1 indicates absolute necessity of connection, while 0 indicates that the client process is independent of any server resource).

However, many networks offer a best-service model with no QoS guarantees. This is usually the case with a wireless network. With no resource reservation allowed for the mobile client, adaptation becomes a powerful and plausible alternative.

1.2 TYPES OF ADAPTATION STRATEGIES

Three types of adaptation strategies are possible (see Figure 1):

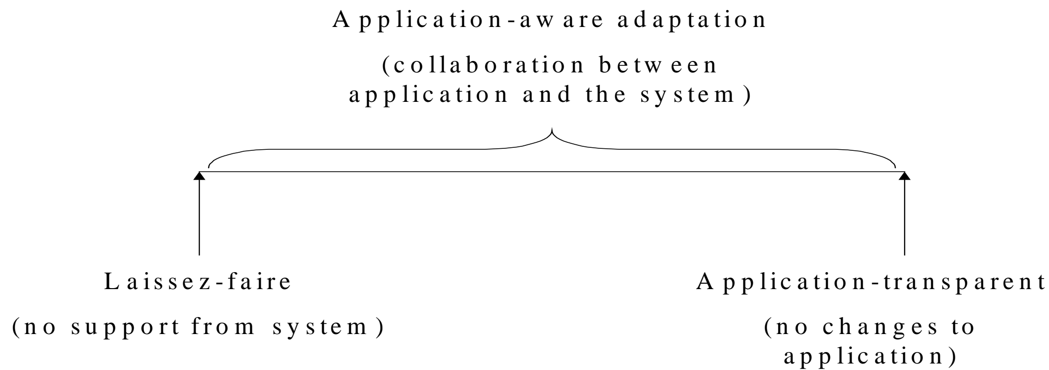


Figure 1: Range of adaptation strategies.

The *Laissez-faire* approach is independent of the system (operating system or middleware). It lacks the central resource arbitrator to resolve the resource demands of individual applications and also to enforce limits on resource demands of different applications. This makes the application code more difficult to write.

Application-transparent adaptation is one in which the system takes entire responsibility for adaptation. This approach provides backward compatibility with existing applications. But the adaptation policies are not only dependent on the type of data, but also on the application-type.

Between the two extremes lies the range of application-aware adaptation, which advocates a partnership between the application and the system to provide adaptation behavior.

We follow the approach of application-aware adaptation, wherein monitoring system resources and issuing notification for changes lie in the system domain. However, deciding

which algorithm to use with a changing environment lies in the domain of an application. Consequently, arbitration of resources lies outside the middleware and system domains.

1.3 DESCRIPTION OF RESEARCH

The primary goal of this research is to design and implement portable middleware capable of providing support for application-level adaptation. We propose an architecture that allows application-level adaptation in distributed applications in general and mobile clients in particular. Support for adaptation of systems specific to a certain class of applications, such as the following, are already present.

- Mobile databases: Bayou, Oracle Lite
- Mobile file systems: Coda, Seer, Amigos
- Mobile distributed objects: CoRe, Globe
- Real-time multimedia for mobile systems: OnTheMove

However, the need for application-independent support for adaptation is clear. This thesis concentrates on adaptation of a client at the application-level for generic applications and demonstrates how such a framework can be designed at the middleware level (of the software-architecture).

One of the problems with extending any of the systems discussed in Chapter 2 (to support applications like mobile agents) is to estimate the network-wide or global resource levels by knowing only the local resource level. Thus, the secondary goal of this research is to enable applications to estimate the network-wide availability of resources.

Currently, many systems providing support for mobile agents base their agent migration policies on the communicating resources in the network. Our intention is that applications should know the computing resources of the clients and servers to derive a more appropriate estimate of the cost needed to do the job. The policies in agent systems presently fail to arrive at a compromise between the priority of getting a job done (essentially dependent on the availability of the computing resources) and the cost involved in getting it done (dependent on the availability of the communicating resources). For example, reaching a node in the network may be less costly (that is, the communicating cost for reaching the node may be less), but the node may be committed to a high priority job thus leading to its high CPU utilization or the

required resource might already be in use. Selecting such a node unnecessarily adds the agent to the CPU queue, thus adding to the delay experienced in scheduling the job for CPU service rather than getting the agent to reach the client. Providing the computing resource levels of individual clients and servers in the directory service helps the agents to arrive at an improved choice of server.

Chapter 2 describes two existing systems providing application-level adaptation. It also looks into an architecture for adaptive applications with no support from the system (either the middleware or the operating system). Chapter 3 provides architectural-level design of the Adaptation Middleware. The chapter also describes the collaboration among various sub-systems of the Adaptation Middleware. Chapter 4 provides the detailed design of the Adaptation Middleware. The chapter concludes by noting the flexibilities in the design. Chapter 5 discusses evaluation methods for measuring the performance of the Adaptation Middleware. It also discusses the results observed in the evaluations performed. Finally, Chapter 6 provides conclusions and summary of our research. The chapter also provides some notes on improvisations for the Adaptation Middleware and some pointers for future research in the field of application-level adaptation.

CHAPTER 2 LITERATURE REVIEW

Application-level adaptation for generic applications is present in Odyssey and Cadmium systems. Both the systems provide application-aware adaptation, the essence of which is a collaborative partnership between the application and the system (middleware or operating system).

2.1 OPERATING SYSTEM SUPPORT FOR ADAPTATION

The role of the Operating System (in the discussion of this section) is to sense external events (like connectivity and physical location) and to monitor the resource availability (like network bandwidth, caching space, battery power and communication costs). On the other hand, the role of an application is to adapt its behavior to the environment information and resource availability indicated by the operating system. Please recall that the adaptation strategy is always dependent on both the type of data being considered and also the type of application.

2.1.1 Coda: Distributed file system

Since the late 1980's the Andrew File System on the CMU campus becomes very large. Server failures are occurring frequently; Coda [7] [12] proves to be an answer to this computing mayhem. Coda is used to allow *disconnected operations* on the file system. With the advent of mobile clients, such as laptops in the scene, the timing could not have been more appropriate. One of the strong aspects of Coda is the prefetching and caching policy it has incorporated.

Coda provides application-transparent adaptation to the file-system access. The heart of the Coda file system is its cache manager – Venus. Coda also has a powerful prefetching policy. Whenever an open() call to a file is issued, Venus fetches the entire file and stores it locally on the client. Only updates are communicated to the server.

Coda provides two modes of operations at the client: connected mode, where the server can be contacted at all points in time, and disconnected mode, wherein the server cannot be reached during the transaction, and all the updates have to be synchronized with the server whenever the connectivity becomes available again.

2.1.1.1 Connected operations

When an application requires data, the kernel queries Venus for the specific file/data, Venus caches all the information retrieved from the file server, if the requested data is not present in the cache then the server is contacted by Venus. Any updates made at the client are synchronously updated at the server also. If the server cannot be reached, transaction time-out occurs and an error is reported.

2.1.1.2 Disconnected operations

To support disconnected operations, when a transaction time-out occurs, Venus logs the update in the client modification log (CML) [19] . These operations become transparent to the user if the user switches to the disconnected mode. Upon reconnection to the server, the transactions in the CML are replayed on the server, bringing it up-to date.

If Venus detects a conflict, then application-specific-resolver (ASR) [17] is invoked to resolve the conflict. If ASR is successful, then the conflict is transparent to the user. Otherwise the conflict has to be solved manually with user intervention.

2.1.2 Odyssey: Support for application-aware adaptation

In the ideal world, mobile clients are resource poor and they depend on servers as the repository for all the data. Clients merely act as caching sites for the application. However in the case of real world mobile clients, the reduced connectivity between the client and the server forces the clients to adapt dynamically to the changing environment. Often the adaptation behavior of an application is dependent on the type of data. Further, even for the common (replicated) data used among applications, the individual application must decide on how best to use the data or even filter duplicate data. With these considerations in mind, Odyssey [18] [19] set out to provide application-aware adaptation, the essence of which is ‘a collaborative partnership between the system and the individual application’, in page 1 of [18] .

2.1.2.1 Structure of Odyssey

Odyssey is a set of extensions to Unix to support mobility. These extensions are provided both at the API level and as internals to the Operating system. Adaptation is achieved by allowing the applications to map between the resource levels and the fidelity levels. However, monitoring and arbitration of the system resources is done by the system (Odyssey).

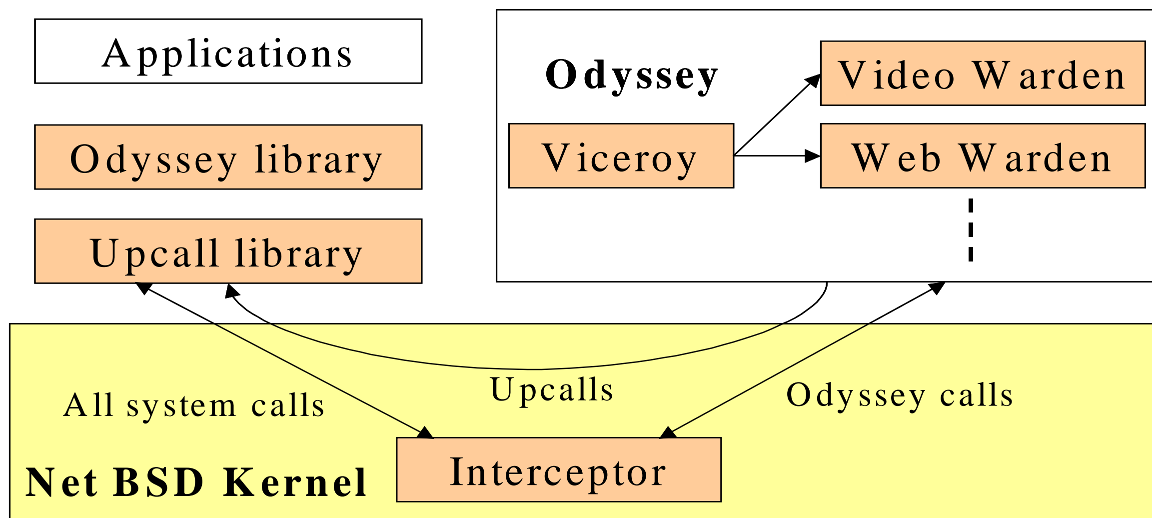


Figure 2: Architecture of Odyssey.

The Odyssey architecture [14] has two types of relationships between the applications and the system. The application is essentially dependent on the type of data (data-centric) when interacting with *viceroys* and *wardens* (to be defined in Section 2.1.2.3.2). However the Odyssey itself is independent of data but is action-centric, providing applications with control over the selection of fidelity levels.

2.1.2.2 Odyssey API

There are three components to the Odyssey API [14] :

- A way for applications to exchange salient features of the environment
- A mechanism for applications to track changes in environment
- A way to request a policy-change based on the changed environment.

The features of the environment can be either application-independent (for example, network bandwidth) or application-specific (for example, number of queries allowed to a web-server in a day). The resource availability is maintained as a single scalar value, the interpretation of which is left to the individual application. For example, a value of 20 in network bandwidth can refer to 20Mbps, in case of queries example, it can refer to 20 queries left.

The API allows applications to request monitoring [*ody_request*()] on a resource and also to cancel the monitoring [*ody_cancel*()]. The applications are also allowed to request policy changes that are 'type specific operations' [ody_tsop()].

ody_request: accepts resource interest, bounds of tolerance on resource availability, the reference item and an upcall procedure (notification handler routine). Whenever the resource availability crosses the bounds of tolerance, the upcall procedure is called.

ody_cancel: cancels the outstanding request.

ody_tsop: can be used to provide a richer set of interfaces than the simple file system interface provided by the native operating system. For example, items of type 'video' might support type specific operation like video_frame_read to read the entire variable sized frame in addition to the read system call.

2.1.2.3 Extensions to Unix for supporting Odyssey API

To support the Odyssey APIs the Unix file system was extended to hold the information on the type of data (codex) in addition to the data itself. Also the cache manager was divided into data-type-independent component (*viceroy*) and data-type-specific component (*warden*) [14]
.

2.1.2.3.1 Tomes and Codices

Since no universal adaptation policy satisfies all the types of data, the adaptation policy must be customized for each type of data. This is the reason why Odyssey stores type information along with the data in tomes (or typed volumes, adapted from the volumes in Coda). Each tome has a codex which refers to the tome type like 'Unix', 'SQL', 'Mpeg video', etc …Tomes are shared data namespaces, organized in a hierarchical manner, they are location transparent.

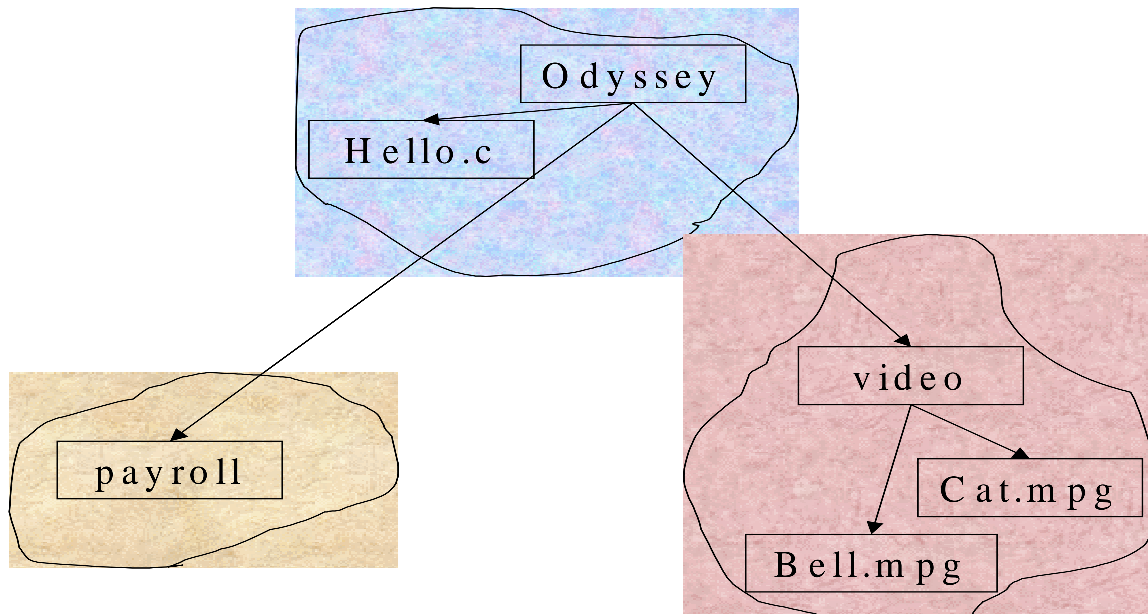


Figure 3: Hierarchical tomes in Odyssey.

2.1.2.3.2 Viceroys and Wardens

Viceroys and wardens are cache managers implemented at the client side. Both the cache-managers are implemented in the user space (rather than the kernel space) of the Unix operating system.

Viceroys are type-independent cache mangers that depend on the wardens, which are type-specific cache-managers, to do the job. Importantly, viceroys act as a single point of resource control in the system. The viceroys monitor the system resources, and inform the applications when the resource level crosses the tolerance bound. Finally, viceroys forward the request on any Odyssey object to the appropriate warden.

Wardens are type-specific cache managers that implement the access methods for specific Odyssey object types, in addition to the Unix operations on those types. The wardens also implement a number of fidelity mechanisms that allow the applications to choose among the different wardens.

2.2 SUPPORT FOR ADAPTATION FROM THE MIDDLEWARE LEVEL: CADMIUM

The middleware along with the OS forms the *system.* As discussed previously, the monitoring of the resource levels and the arbitration of the resources is done by the system. The applications are informed of the changed environment through an upcall. In the middleware approach, these supports are built as a layer between the application and the operating system; the internals of the operating system are left untouched.

Cadmium [2] [20] is a research project in the SOR group (Systèmes d'Objets Répartis, i.e. Distributed Objects Systems), that studies operating system support for information sharing in large-scale distributed systems. Cadmium has an application-aware adaptation model similar to Odyssey. The system, application and users can all be made aware of the currently available resources. The approach is a network-aware [3] model that can identify and use locally available resources, adapt to changes, and provide an appearance of continuity of service.

Cadmium provides continuous service in spite of weak connections, disconnections or reconnections. In order to provide such connectivity, Cadmium must ensure the availability of data at all times. The approach followed by Cadmium is to provide data and code replication (cache) on both sides of the network, fixed and mobile; additional support is provided to deal with the multiple copies of data.

Cadmium provides the information on the resource level and also makes an upcall (feedback) whenever resource levels cross the tolerance bounds. The applications are expected to adapt dynamically to the changing environment.

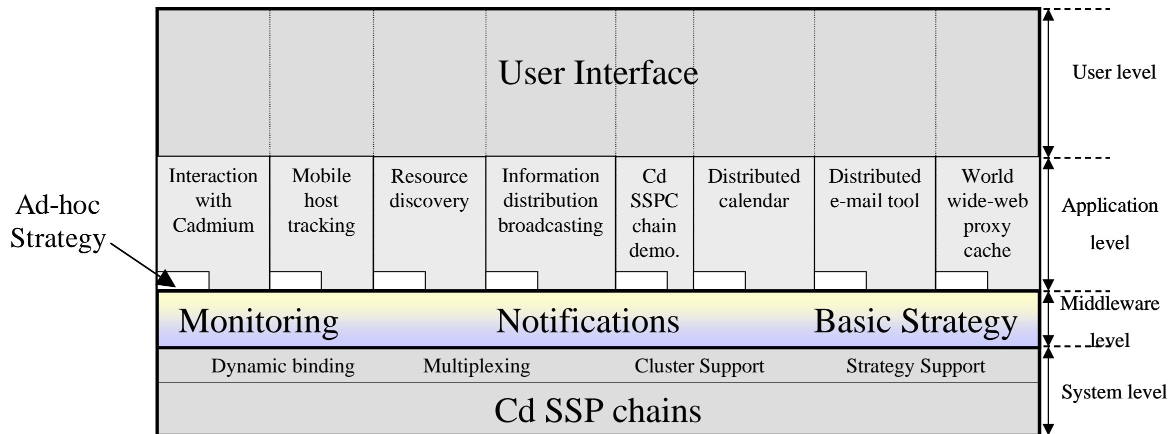


Figure 4: Cadmium design overview.

As seen in Figure 4, in Cadmium the mechanisms to enable adaptation are distributed over four levels [1] :

System level: this level includes abstractions of the systems as objects and references. References are used to retain meaning across disconnections. Due to replication, multiple copies of objects may be available; references are redirected using *flexible binding* [1] , which dynamically selects the 'best' available representation for the object.

Middleware level: Cadmium provides a set of software tools that reside between the operating system and the applications, without modifying the internals of the operating system. Applications benefit from the extended operating system features by using the services of the middleware. The middleware provides libraries of basic algorithms, called *strategies*, for access control, replication, consistency, conflict detection or resolution. A strategy can be used on a per-object, per-cluster or per-application basis. A strategy can be dynamically loaded into an application and replaced as the need arises. Strategies can be provided either by the system or the application.

Cadmium provides for environment monitoring and callback registration for notification of environment changes, just like in Odyssey. An application can either add new strategies or reuse the existing ones.

Application level: this level contains the application-specific strategies, that is, type or semantic specific strategy to be provided as part of the Cadmium libraries.

User level: in this level the user or application profiles are specified. These can be used by Cadmium as hints to the user/application needs.

2.2.1 Objects and references

Objects [1] are the collected data that can be modified as a unit. These are abstractions provided by the system level; adding a method to their class, that is, modifying the behavior, can modify objects. The powerful feature of the objects is abstraction, for example, the printer object in the client might print the job when it is strongly-connected (when there are lesser interferences in the wireless communication) to the printer or it might queue the job for print during the disconnected mode. Objects can also migrate among different machines and connections. Objects can be grouped into a cluster; the way a cluster is formed is dependent on the type of application.

Objects are identified by references, two users having the same reference are said to share the corresponding object. References can be temporarily unavailable or disconnected. References [1] also support object migration (location transparency) through the use of modified Stub-Scion Pair Chains (SSPCs).

2.2.2 Binding

Binding acts as a validation process for the references. Only after the binding, can the objects communicate. Cadmium provides for flexible binding wherein the mobile client and server can negotiate the binding. This allows a mobile client to tear down an existing binding (maybe to an unreachable server) and to re-establish a new one for another server object. This network-aware flexible binding protocol allows the negotiation to have parameters like network bandwidth, access privilege, Quality-of-Service and interface compatibility.

Cadmium allows two kinds of flexible binding: Quality-of-Service based flexible binding and flexible reference redirection binding.

- Quality-of-Service based flexible binding allows an application to negotiate the QoS of the link to the server. This binding is used by the application for adaptation purpose.
- Flexible reference redirection or flexible reference binding allows the mobile client to locate the ‘best’ object among different replicas available.

As opposed to Quality-of-Service based flexible binding, flexible reference binding is not concerned with the QoS issues, it rather concentrates on the finding an object for a particular criteria. It is not an application-level binding but a system-level binding, providing only object selection as the support for adaptation.

2.3 DESIGN OF ADAPTIVE APPLICATIONS

An Adaptive application is one in which the application changes its behavior according to the perceived constraints in the environment, so as to maintain the semantics (or expected behavior) of the application for the user.

Often adaptation to resource constraints is considered a system's problem, the adaptation is visible to the user as changes in utility, and hence the user must be involved in designing the application. The adaptation policy followed is that of *Laissez-faire* as discussed in Section 1.2. The architecture of adaptive application [11] discussed in this section is demonstrated in Figure 5.

The first step in the application design is to identify the resource constraints and the application semantics. Next throughout the design, the application semantics should be maintained for changes in the application behavior. The application changes behavior trying to adapt to the changing environment.

In order to realize the layered approach, see Figure 5, we need to make a number of design choices like providing an open implementation (OI) of the underlying technology that exposes the network resources of that implementation. The adaptation controller can adjust the network requirements of the components, thus allowing the component to adapt to the prevailing network condition. Usually an adaptation controller can be implemented as part of a component.

Providing implementations of components that are orthogonal to each other such that they can reduce the resource usage in different ways. By providing orthogonal implementations we can increase the degree of freedom of the application. For example, by storing the tuples *(implementation choice, resource usage, application utility)* the designer can choose the implementation choice that is the best compromise between the resource constraints and application utility.

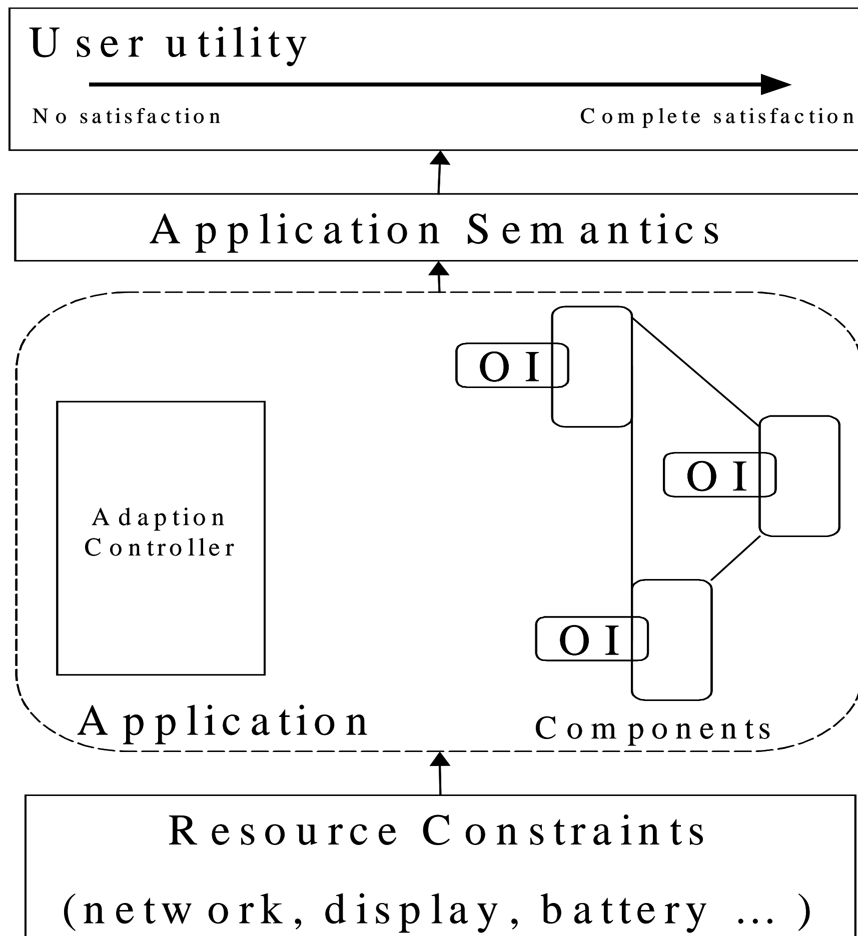


Figure 5[1]: A layered view of an adaptive application.

Many users are not concerned about the subtleties of the network and display characteristics – they are more interested in having the software work. For such software-illiterate users, some fixed behavioral paths (trajectories or paths of execution) should be determined at design time. This clearly requires the users to be involved in the design-time.

The choice on whether or not to include the user intervention during adaptation depends on the resource in consideration and also the adaptation behavior. The user's intelligent intervention can be sought in cases like battery monitors querying the user if the application can

[1] OI seen in the components is an acronym for Open Implementation

shut down. If user's intervention is accepted then the user must also be educated to handle the adaptation behaviors.

To deal with a majority of resource constraints, the application can be allowed to execute in a pre-determined trajectories or paths. All the contexts in which an application will execute cannot be determined at the design time. Hence, selection of the degradation behavior of an application should be allowed at the run-time. That is, a user should be able to override default profiles depending on the context of the application execution.

Resource constraints should be interpreted to a metaphorical situation that the user is already familiar with. This aids the user in proper interpretation of the situation and aids the user to take corrective decision.

Finally, there has to be an interface like Media Policy Language (mpl) to specify the policy of degradation at the run-time [11] .

CHAPTER 3 ARCHITECTURAL-LEVEL DESIGN

Applications depend on one or more resources during their lifetime to carry out their tasks. In some situations, particularly in the mobile environment, reservation of a resource may not be possible. For instance, a resource such as bandwidth may be available during the resource reservation period, but may become unavailable during the lifetime of the application (due to the relative mobility of clients and servers). In such situations, applications need to monitor the resource levels and adapt to the available levels of resources. Dependency on resources is not limited to applications executing in mobile environments alone, the problem is more relevant to distributed applications in general. Also at times the very nature of the resource may not allow any kind of reservations to be imposed on it; the applications might be required to use resources as and when they become available.

This section explains the ideal architecture of a portable adaptation middleware. The section also presents some constraints in implementing such an architecture followed by a modified architecture that can be implemented as portable middleware.

3.1 CLASSIFICATION OF RESOURCES

The usefulness of an adaptation layer is more pertinent to distributed systems in general and not limited to mobile environments alone. One of the fundamental problems with Odyssey (see Section *2.1.2*) and Cadmium (see Section *2.2*) systems is their focus on data availability for mobile clients. The author's view of adaptation is more general and not specific to any application domain. This section analyzes types of resources. Further, the section also discusses the interaction among sub-systems in the architecture.

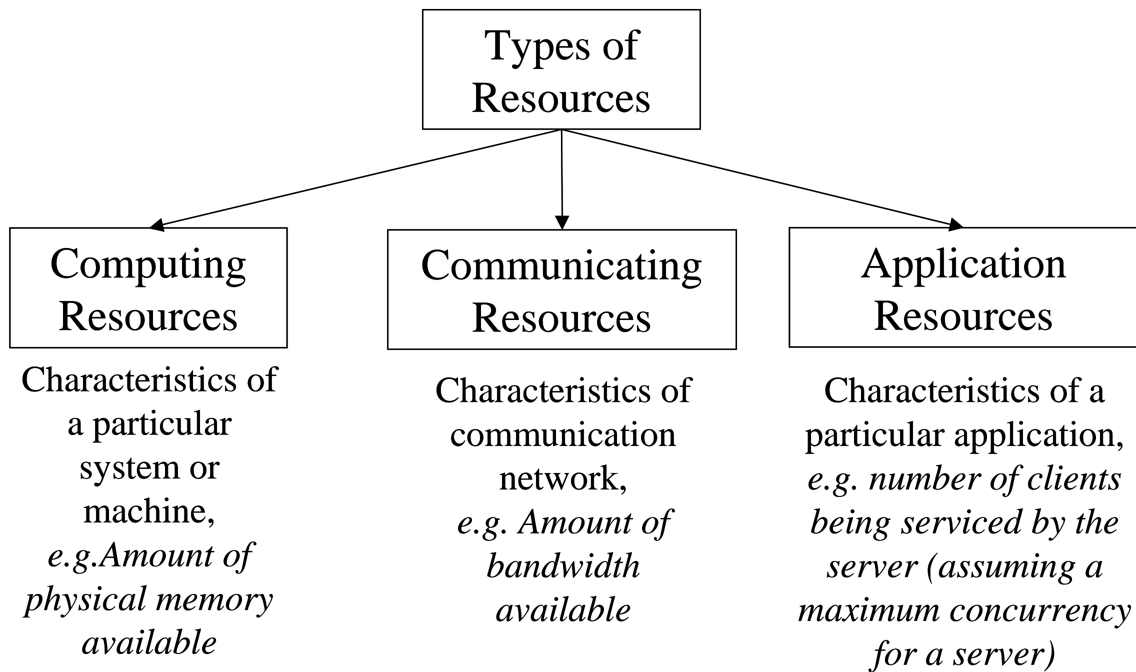


Figure 6: Classification of types of resources.

We begin the analysis of the adaptation problem by looking at the categories of resources[2], see Figure 6. A client has two categories of system resources, ones that are concerned with its computing capability and the others that are concerned with its communicating capability. Each resource can be static or change dynamically during the lifetime of the client, see Figure 7. Lack of a particular kind of resource can cause extra demands to be placed on some other kind of resource; for instance, in the case of a client-server multimedia application, lack of communicating resources introduces overhead on the computing resources. In particular, poor bandwidth availability makes it necessary for the server to compress the multimedia data before transmission; and in turn, the clients need to decompress the data on reception. Since the resource needs of a system and resource requests by a system are dependent on each other, it is frequently necessary to compromise on one category of resource due to the scarcity of the other.

---

[2] The terms *resource* and *attribute of a resource* are used interchangeably. Specifically, the value (or level) of a resource refers to the value associated with a particular attribute of a resource.

	Computing resources	Communicating resources
Static resources	▪ Number of processors ▪ Clock speed of the processor. ▪ Capacity of random access memory (RAM) ▪ Physical disk capacity	▪ Peak available bandwidth
Dynamic resources	▪ Current disk queue length ▪ Available disk space ▪ Number of active processes ▪ Processor queue length ▪ CPU utilization ▪ Number of page frames and size of each page-frame	▪ Battery power ▪ Jitter and latency ▪ Error rate ▪ Network connectivity (strong connection, weak connection or sleep mode) ▪ Traffic characterization (running average of number of bytes sent and received, number of failed sessions) ▪ Cost of link utilization in US dollars ($)

Figure 7: Computing and communicating resources.

Besides computing and communicating system resources, many applications have entities within them that act as providers of resources. A simple example is the number of clients being serviced by a server, which represents the load on the server. The more clients being serviced, the greater the load on the server. A client program should be able to choose to interact with a lightly loaded server that is presently servicing fewer clients. Thus the notion of a resource is specific to an application.

3.2 IDEAL ADAPTIVE MIDDLEWARE SOFTWARE ARCHITECTURE

This section presents the components of adaptation middleware, followed by ideal software architecture for portable middleware that supports application-level adaptation. The section further modifies the ideal architecture to present a more realizable high-level architecture supporting application-level adaptation.

3.2.1 Components of an adaptation middleware

An adaptive application by definition needs to change its behavior with changes in resource levels. The essence of this adaptation system is to have a sub-system for monitoring[3] resource levels, a subsystem for providing feedback, and a sub-system for convenient introduction of policy changes, see Figure 8. These can be implemented in the middleware layer with little (or at times no) changes to the operating system. A fundamental problem in extending any of the systems discussed in the previous sections (to support applications like mobile agents [8] [9]) lies the estimation of network wide (or global) resource levels by monitoring only local resource levels. An associated concern with adaptation middleware is portability. In this section, we describe portable middleware architecture to provide knowledge of network wide resources that is necessary to support applications such as mobile agents.

---

[3] *Monitoring a resource* refers to querying the status of a resource — fetching a value of an attribute of the resource — and not to synchronize the access to the resource.

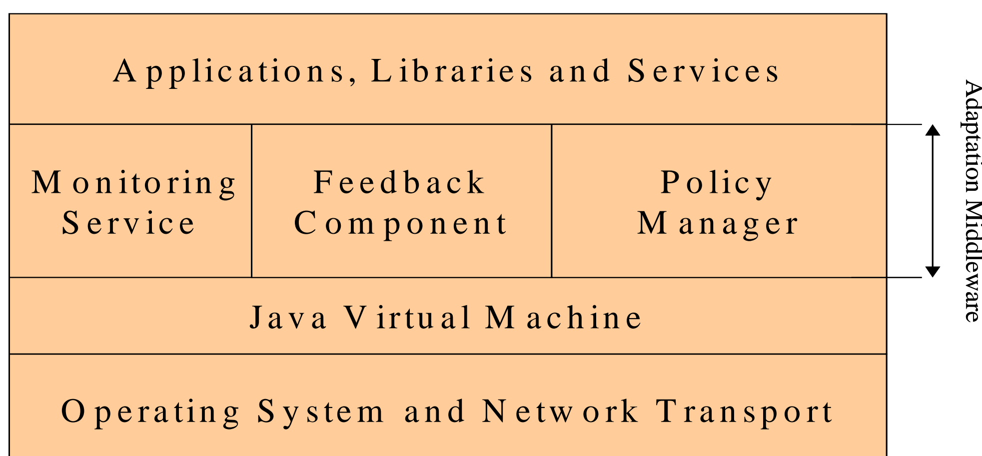


Figure 8: Ideal portable adaptation middleware software architecture.

The components of an adaptation system (or adaptation middleware) have to be sandwiched between the virtual machine (to provide portability) and the applications (and services). Since static resources do not change their attribute levels, the Feedback Component — which informs clients of changes in resource levels — will not be able to notify the attribute values of static resources to the clients. Optionally, a Directory Service can be provided to enable the estimation of static resource levels present network-wide.

3.2.2 Effects of resources on sub-systems

The *Monitoring Service* can actively monitor the system resources. But, active monitoring[4] of application resources is not possible; for example, the number of clients being serviced by the server. Resources associated with applications necessitate a passive monitoring component that can periodically be informed by the application regarding the levels of its resources. The Monitoring Service thus acts as a *funnel* for all the resource changes occurring in the system.

[4] A service that periodically queries the status of a resource is considered to be actively monitoring the resource. In the case of passive monitoring, the resource or the application updates its status with the monitoring service.

All applications may not require changes in all resources. Even for a given resource of interest to an application, the application may not require all resource level changes. The *Feedback Component* acts like a *filter* for resource level changes received from the Monitoring Service. The Feedback Component should be able to notify applications selectively across the network.

The resources on the other hand do not concern the *Policy Manager*. The Policy Manager acts as a *storehouse* for policies. Policy control mechanisms, developed by an application developer or a library developer, can choose the appropriate policy to be used to enable the adaptive behavior.

3.2.3 Constraints in implementing the ideal architecture

We choose Java™ as the language of implementation of the portable middleware components. Java™ is widely popular, possessing a rich set of language features allowing for implementation of a wide range of applications. The language support for data structures and advanced object-oriented (and component based) programming makes Java™ an obvious choice for developing portable adaptation middleware.

However, there are some inherent disadvantages in using Java™ as middleware development language. One of the chief disadvantages is the lack of support in Java™ Virtual Machine (JVM) for run-time space and processor analysis. Further, the implementation of the JVM varies from one computer to another, forcing the monitoring of system resources to be non-uniform across machines. Implementers of virtual machines are always presented with an inevitable conflict between language portability and support for finer granularity access to system operations. Implementation specifications of JVM abstract the characteristics of the system in favor of language portability.

3.3 THE REVISED SOFTWARE ARCHITECTURE

In order to overcome the constraints specified in Section 3.2.3, the revised architecture has the monitoring component implemented over the native operating system instead of over the JVM (see Figure 9).

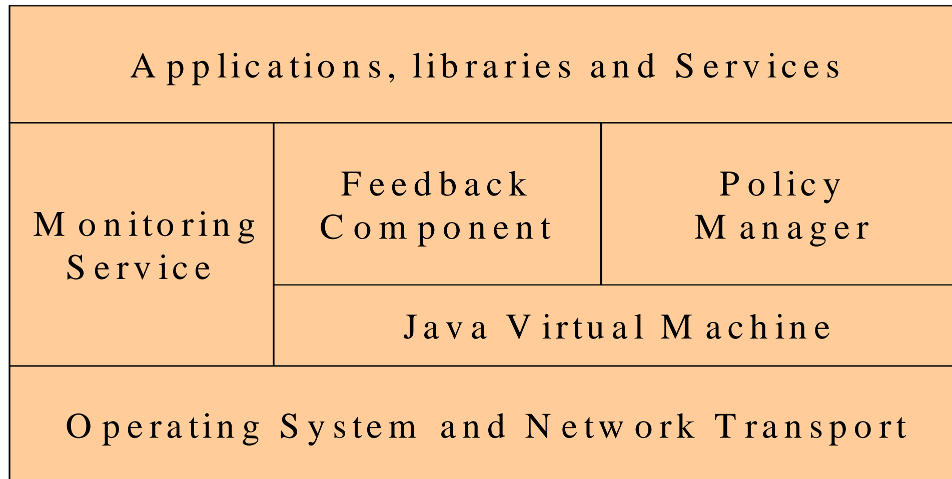
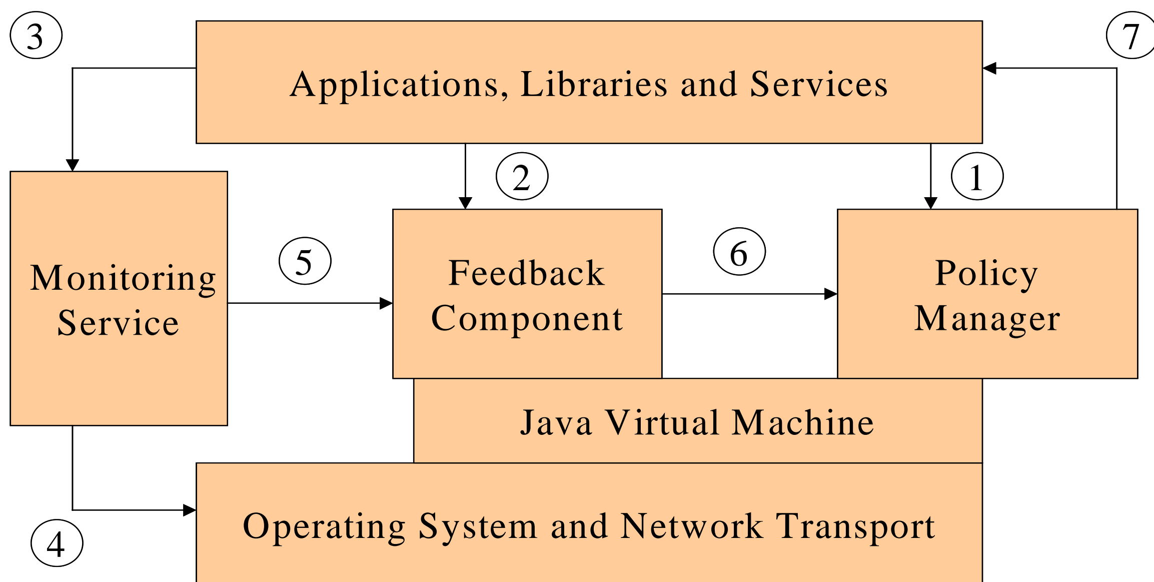


Figure 9: Revised portable middleware software architecture supporting application-level adaptation.

The Monitoring Service is split into two services, an Active Monitoring Service and a Passive Monitoring Service. Since the Passive Monitoring Service does not query the resource levels, it is not specific to any platform. The revised architecture thus isolates the platform-dependency problem to the Active Monitoring Service alone. In order to inform more than one application, the Adaptation Middleware needs to store the resources values. Alternatively, the Monitoring Service can store the resource values in a Directory Service. For convenience and to improve execution efficiency, inclusion of Directory Service can be made optional. The Feedback Component informs applications about change in resource-levels across the network.

Collaboration among subsystem components in the revised version of adaptation middleware is shown in Figure 10.



1. Application (or library developer) registers policies (or library) with the Policy Manager

2. Application (or performance tuner) registers its policy control mechanism, *i.e.* when to switch policies

3. Passive monitoring: Applications inform monitoring service about changes in their resource levels

4. Active monitoring: Service monitors change in resource levels of the system (network and OS)

5. Monitoring Service informs Feedback Component about change in resource levels

6. Feedback Component requests Policy Manager for changes in policy

7. Policy Manager switches current policy, effects of which are visible in the application

Figure 10: Sub-system collaboration diagram.

CHAPTER 4 COMPONENT-LEVEL DESIGN

This chapter describes component-level design of the Adaptation Middleware. The chapter first describes the structure of the Adaptation Middleware, followed by the design of the three individual sub-systems. For each sub-system the design of data structures and interfaces is described. Finally, the chapter concludes with a brief look at the flexibilities in the design.

4.1 STRUCTURE OF THE ADAPTATION MIDDLEWARE

The architecture of the middleware follows the brief introduction in Section 3.2.1. The functional behavior of Adaptation Middleware is defined by the joint operation of the Monitoring Service (MS), the Feedback Component (FC) and the Policy Manager (PM).

MS has two components: the Active Monitoring Service (AMS) that is platform dependent, and the Passive Monitoring Service (PMS) that is platform independent. The AMS can be implemented as a daemon process, which periodically queries the operating system and network resources for their status or resource levels. The PMS in contrast is platform independent. The primary responsibility of PMS is to funnel the changes in resource levels, supplied by the AMS and the application, to the FC. In this respect, PMS can be implemented as a single thread waiting for resource levels inputs.

FC acts as a filter for all resource level changes in the system. In order to notify applications of changes in resource levels, the component has to allow applications to register for such notifications. An implicit requirement is that each application 'deregisters' for such notifications. Thus the FC can be implemented as an RMI server in Java™. The process of registration will require a feedback object, or in particular a Java™ object implementing a particular Java™ Interface, to be registered that can be notified of relevant changes. This registered interface (or object) should implement the policy control mechanisms, which requests the PM to change the current policy in use whenever required, to enable the adaptation behavior.

Registration for notifications with the FC does not cause the resources required by an application to be monitored. Monitoring a resource is purely the function of the MS. Due to the platform-dependent nature of the AMS, the applications (or users of applications) need to have an external means of initiating the monitoring process. For resources that are not dependent on

the AMS, such as application resources, the PMS will automatically funnel the changes to the FC.

The PM acts as a storehouse of policies with additional abilities to load new policies and to change the current policy of an application. The primary function of the PM is to allow applications and libraries to register policies — methods implementing various algorithms — with it. The PM must allow applications to request use of certain policies and also to change the current policy in use. Adaptation behavior is possible due to changes in the use of current policies. For this reason, the PM must accept requests to change polices by any policy control mechanisms present either in the FC or in the application itself. Thus the PM is implemented as an RMI™ server in Java™.

4.2 MONITORING SERVICE

This section describes the MS in more detail. It explores the issues of data structures and interfaces used within the sub-system, and some of the limitations and constraints in design.

4.2.1 Processing narrative for Monitoring Service

The Section3.1, describes various classes of resources. Among them, the resources associated with computing and communicating capabilities of the client are platform specific. Presently, JVM has no (or very little) support to monitor such system resources. Hence, the MS needs two components, one that is platform dependent and another that is platform independent.

As seen in Section 3.3, many platform dependent components might be needed to enable the monitoring of system resources. These components need to periodically query the resource levels from the operating system. Additionally, these platform dependent components can also have some applications executing as daemon processes to collect necessary performance data. Further, an already existing notification service like SENS — System Event Notification Service — can be integrated, via a Write API, into the Adaptation Middleware (see Figure 11).

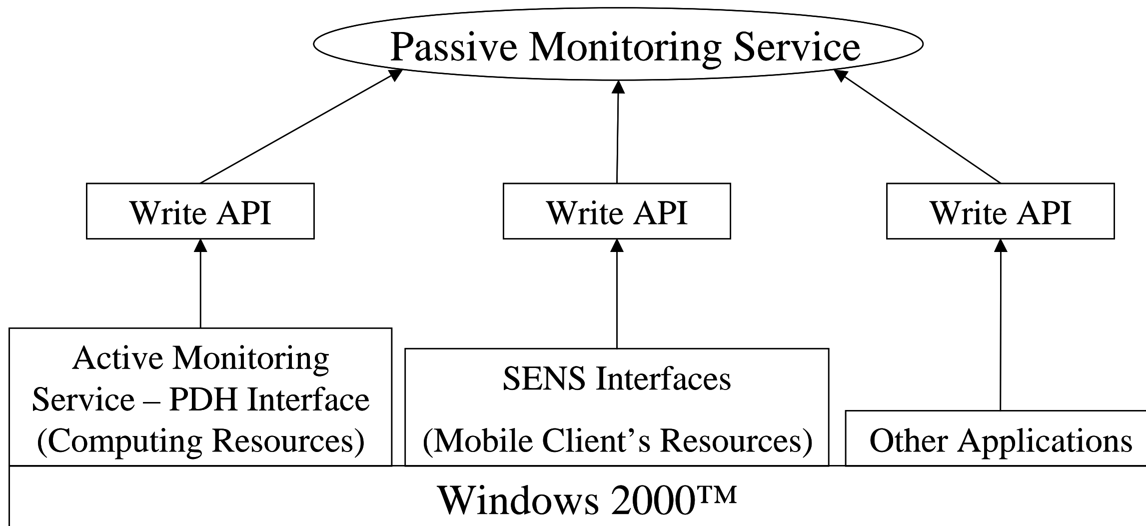


Figure 11: Platform dependent monitoring of system resources[5].

Thus the PMS collects all the changes in the system, forwarding the information asynchronously to the Feedback Component, as described in Section 3.3.

4.2.2 Description of interfaces in Monitoring Service

The AMS is platform specific, it has no interfaces to be informed of any resource level changes. However, the PMS has a socket interface to notify resource changes. Neither the AMS nor the PMS can store resource levels internally; hence they have no output interface to query the level of any resource.

[5] PDH is an acronym for *performance data helper* interface provided by Windows 2000™ operating system. Also, SENS is an acronym for *System Event Notification Service* provided by Windows 2000™ operating system, which enables applications to be notified of any changes in resource levels such as battery power and bandwidth.

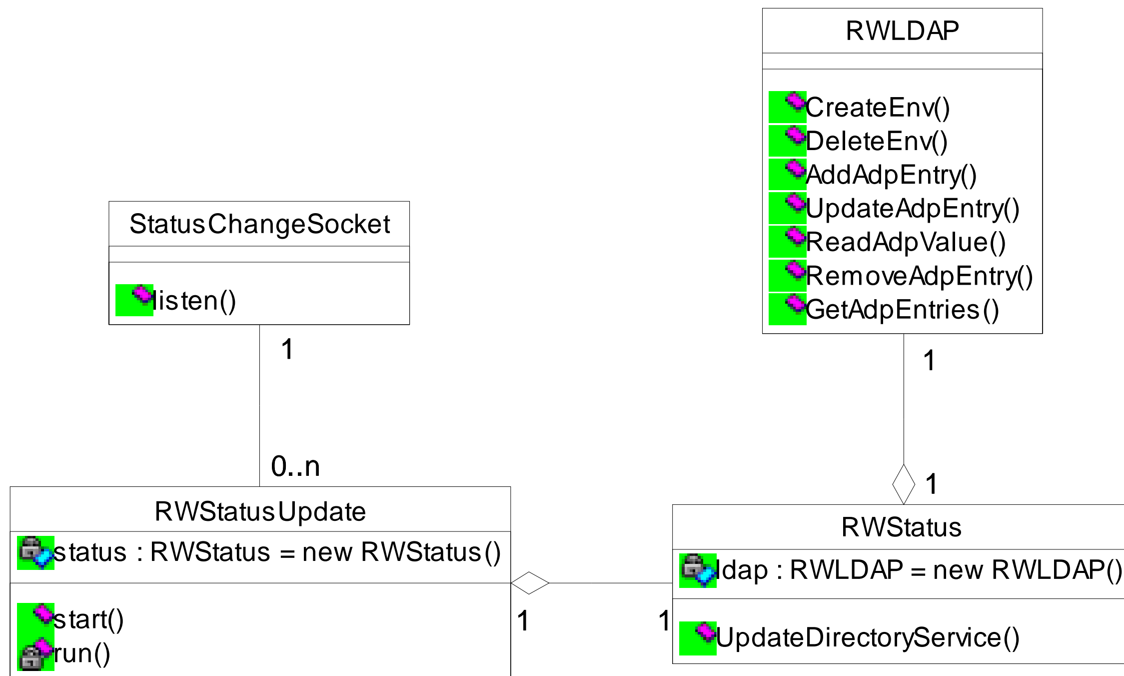


Figure 12: Class diagram for the Passive Monitoring Service.

An API is provided to query the stored resource level in a client. The PMS stores the value of a resource level in a portable Directory Service such as Netscape Directory Server™ (NDS) supporting the Lightweight Directory Access Protocol™ (LDAP) — see RWLDAP.java in Figure 12. On access, a time value in milliseconds is specified; the value is retrieved if it is more recent than the time specified, this ensures that the user is not returned an obsolete value.

As previously noted, the AMS and applications need to inform the PMS regarding their resource levels. Since PMS lies at the boundary of platform dependent and platform independent components, this communication needs to take place via a file system abstraction or a BSD socket abstraction that is supported on all platforms.

4.2.3 Processing details of Monitoring Service

The following sections presents a detailed algorithmic description of the Monitoring Service, a description of data structures used and design constraints.

4.2.3.1 Interface description

This section describes the representation of a resource and interpretation of its value. Also from *Section 4.2.1*, the PMS sits at the boundary of platform-dependent and platform-independent components. This section describes the protocol for applications and the AMS to inform the PMS of their resource level changes. The section also describes the *RWLDAP* interface to read and write resource values in the Directory Service (see Figure 12). Interaction among interfaces is shown in Figure 13 and Figure 14.

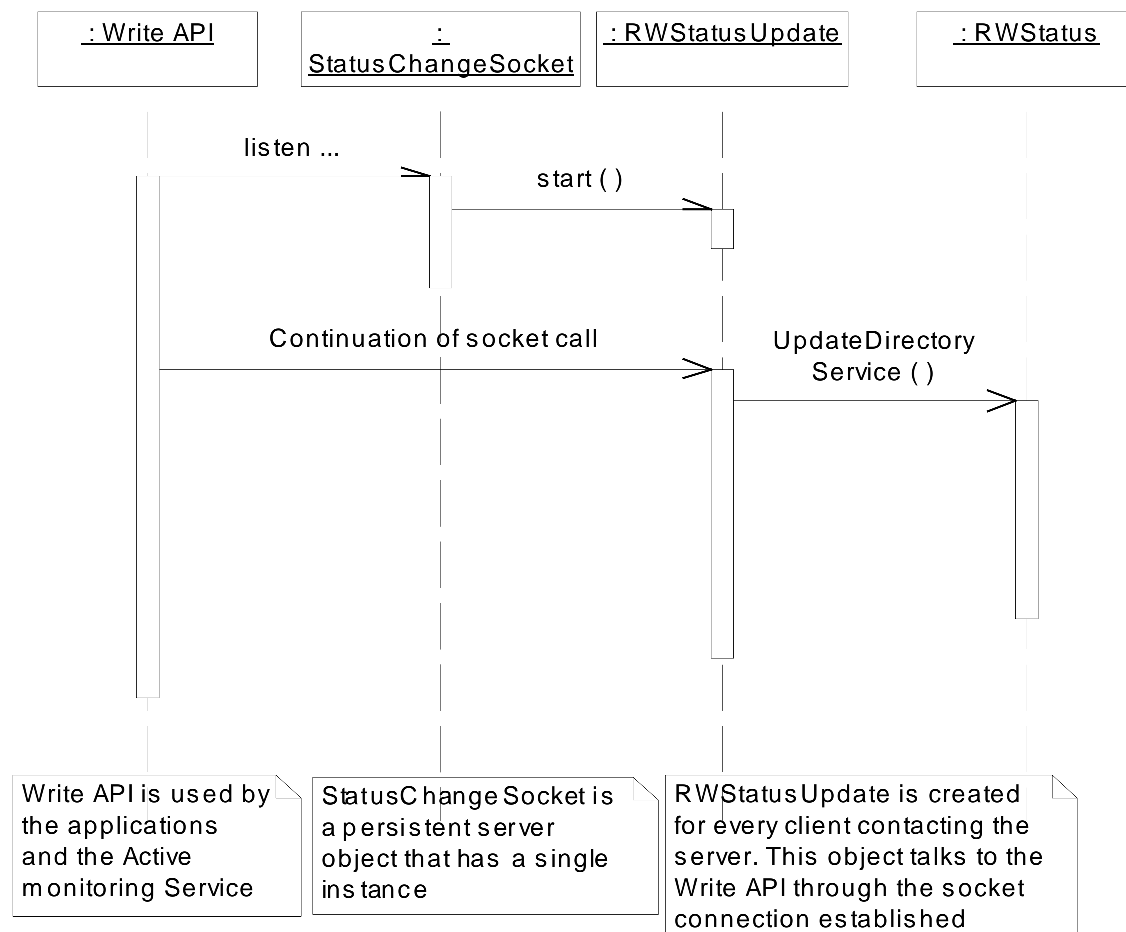


Figure 13: Interaction diagram — Part 1 — of the Monitoring Service.

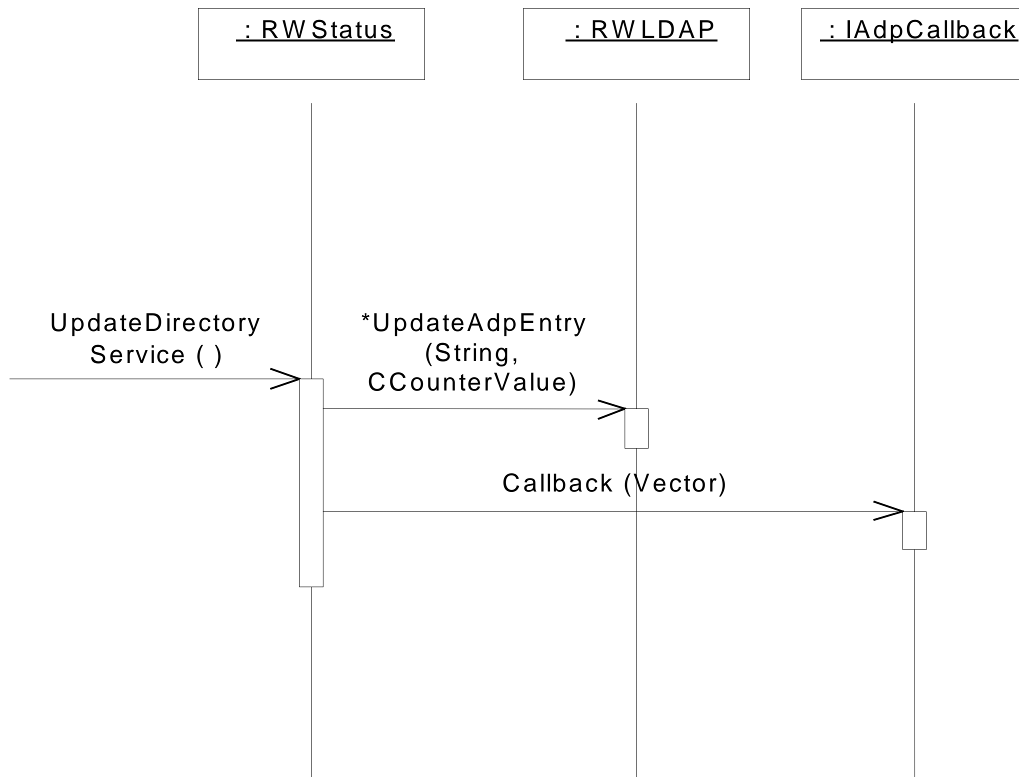


Figure 14: Interaction diagram — Part 2 — of the Monitoring Service.

4.2.3.1.1 Representation of a resource

A resource is usually measured with the help of a counter. The value in a counter is incremented or decremented to represent the status of a resource. Similarly, each counter is associated with a particular resource and more often represents just one of the attributes of that resource. For instance, a resource like virtual memory can have more than one attribute associated with it, such as: (1) the total amount of virtual memory[6] in the system, and (2) the amount of virtual memory available. Thus a resource like virtual memory needs at least two

[6] Virtual memory refers to the physical memory allocated by the operating system for segmentation and paging purposes. Hence *amount of virtual memory* actually refers to the amount of physical memory allocated for the purpose of paging.

counters to be associated with it. Similarly, each individual counter represents the status of a particular attribute of virtual memory available to a client.

Hence to represent a resource level by name, we require the name to have two components: the name of the resource and the name of the attribute (or counter). The generic representation can be ***<resource name> \ <resource counter>,*** for instance — "System \ Context switches per second" or "Virtual Memory \ Total Memory".

The above representation can be extended to represent a resource level associated with a particular machine. That is, we can represent a hierarchy in the name. A simple representation would be ***<machine name> \ <resource name> \ <resource counter>,*** for instance — "BattleBot \ Battery \ Battery life remaining" or "My Machine \ Virtual Memory \ Memory Available".

Further, the value of a resource is treated as a 32-bit scalar quantity. Interpretation of the value is left to the application. For instance, a 32-bit value of 1000 could either mean 1000 KB of virtual memory when associated with "Virtual Memory \ Total Memory" or 1000 context switches per second in a CPU when associated with "System \ Context switches per second."

4.2.3.1.2 Read and write values of resource levels

Applications or users (like administrators) may require the MS to store resource values. Alternatively, the MS can store data values in a Directory Service so that the data values are available for all the machines in the network. Portable middleware necessitates a portable Directory Service; Netscape Directory Service™ serves this purpose. Further, Netscape Directory Service supports Lightweight Directory Access Protocol™ (LDAP) API to allow platform independent browsing of the Directory Service. To keep a manageable load, reading and writing of resource levels are not supported directly through the Monitoring Service.

The RWLDAP API (see RWLDAP in Figure 12) provides methods to write (or read) resource values to (or from) the Directory Service. The RWLDAP API has two important methods:

UpdateAdpEntry: It accepts the name of a resource counter, the value of the resource counter and a string specifying a short description of the counter. This method returns a *Boolean* result indicating success (or failure) of the operation. The descriptive string needs to be specified only the first time the method is called; during the first call to the method, if an entry

representing resource counter is not present in the Directory Service, then this method creates an entry.

ReadAdpValue: It accepts the name of the resource counter, a variable to return the value of resource level and a time in milliseconds. This method returns zero or a value — when the value stored in the Directory Service is more recent than the specified time value. This method also returns a *Boolean* result indicating success (or failure) of the operation.

4.2.3.1.3 Write API to specify resource levels to the Passive Monitoring Service

A *Write API* provides specific methods to enable communication between the platform-specific components and the Passive Monitoring Service (see *WriteAPI* in Figure 12). The *Write API* has two important methods:

GetWriteObject: This method returns a *Write API* object if one exists, else it constructs a new one for the application. The newly constructed object is returned on successive calls to the method. A single *Write API* object ensures a single point of communication among different components and the Passive Monitoring Service. That is, it enables different points of update in a single application.

UpdateValue: This method accepts a string representing the resource counter and a 32-bit value that is the resource level value. This method has no return types.

4.2.3.2 Algorithmic model for Passive Monitoring Service

The *Write API*, as seen in *Section 4.2.3.1.3*, shields the application from the protocol involved in communication between the platform-dependent and platform-independent components. For this reason the means of communication should employ abstractions such as file systems or BSD sockets that are available on all platforms. A protocol involving BSD sockets is described below.

1. PMS listens for updates on TCP port #2598.
2. *Write API* (or some other application) sends a 32-bit number, indicating the number of resource level updates, to the PMS through a TCP socket connected to port #2598.
3. Each resource level value is sent in the following format:
   <Length><resource name>\<resource counter>\<resource value>
   where <Length> and <resource value> are 32 bit numbers and <Length> indicates the length of the sequence — <resource name>\<resource counter>\<resource value> — in

bytes. Further, any name hierarchy as described in *Section 4.2.3.1.1*, can be incorporated. The PMS reads the last four bytes of the sequence to interpret them as the resource value. Remaining bytes between <Length> and <resource value> are treated as the name of the resource counter.

4. After reading all the resource levels, the client closes the socket connection, allowing other clients to connect to the server.

As mentioned earlier, the PMS funnels the changes in resource levels, received from the AMS and other applications, to the Feedback Component.

4.2.3.3 Restrictions or limitations of Monitoring Service

There are no restrictions or limitations in this model as long as the level of a resource being monitored is measurable. The only restriction is that the resource counter value be a 32-bit number. To simplify the restrictions on a resource, we specify the need for a hierarchical representation. Also, we refrain from interpreting the value of the resource level. Further, having the resource levels in a Directory Service enables static resource values to be retrieved whenever necessary (see Figure 7).

4.2.3.4 Data structures used in Monitoring Service

The *Write API* uses a *hashtable* data structure to the store resource counter values. Multiple changes in the value of a resource before its communication to the PMS results in the last value.

The PMS receives a series of strings representing the name of the resource counter and a 32-bit number specifying the value of the resource counter. PMS does not store the counter values, but instead re-constructs the *hashtable* data structure to pass it to the Feedback Component.

4.2.3.5 Performance issues in Monitoring Service

In the software design, *Write API* is designed to provide a periodic update to the Passive Monitoring Service. This allows for a set of values to be communicated to the Passive Monitoring Service, rather than communicating each individual value as and when they become available. The resulting periodic communication reduces the communication overhead. As a side effect, rapid changes in the resource levels are filtered; thereby introducing stability in adaptation behavior. This design for *Write API* can incur noticeable delays before applications adapt to

changes in resource levels. To keep such delays to a minimum, the *Write API* communicates the resource levels every second to the Passive Monitoring Service.

If the performance overhead of one second is not acceptable for an application or resource being monitored, custom APIs can be developed to communicate the resource levels asynchronously or at a smaller period. The PMS by itself does not introduce any delays in informing the Feedback Component.

4.2.3.6 Design constraints of Monitoring Service

The design of PMS is constrained by the need for platform independent BSD sockets to be present in all platforms. The port #2598 assigned for the PMS is tentative; if the port is not free, the PMS can listen at some other port or a reserved port can be assigned for this purpose.

The overall design constraint for the Adaptation Middleware is the same as that of Java™. The MS is developed specific to Java Virtual Machine™ (JVM) and Netscape Directory Server™ (NDS). NDS itself is implemented in Java™ and hence the entire MS is constrained by the portability of Java™.

4.3 FEEDBACK COMPONENT

This section describes the FC in more detail. It explores the issues of data structures and interfaces used within the sub-system, and identifies some of the limitations and constraints in design.

4.3.1 Processing narrative for Feedback Component

As mentioned in Section 3.3, the FC acts as a *filter* for all resource level changes in the system. Applications tend to be selective about the resources they seek to monitor. For a given resource an application might not require notifications for all the resource level changes. Applications often require the range in which a resource level lies. The determination of a resource level crossing a *threshold* can be important (either in the increasing or the decreasing direction). For instance, consider the case of a client-server multimedia application interested in adapting to varying bandwidth conditions. The multimedia application needs to use different levels of compression at the client and the server depending on the currently available bandwidth. The policy for adaptation is to identify the ranges of bandwidth and the

corresponding compression algorithms to be used. The application is now requires the range in which the current bandwidth lies, rather than the exact value of bandwidth, to use the appropriate compression algorithm. To find the range, thresholds have to be identified, and it is sufficient for the application to be notified whenever current bandwidth crosses the identified thresholds. In the Adaptation Middleware, we refer to these thresholds as *watermarks*.

Thus the FC should have a mechanism to identify the resources and the applications. The FC should also maintain the mapping among resources, applications and their designated watermarks. Further, to reduce the overhead involved in informing the applications, FC needs to convey all the relevant changes in resource levels to an application. If changes in status occur for an application in more than one resource, then the FC communicates all changes required by the application in a single notification.

4.3.2 Description of interfaces in Feedback Component

As mentioned earlier in *Section 4.1*, the FC needs to allow for registration of interface (or callback objects) that need to be informed of resource level changes required by the application. Since each application may require different values of a resource level, they need to register their watermarks with the Feedback Component. Each watermark is an array of 32-bit values. For each resource of interest, the application needs to register two watermark arrays, a *low watermark array* and a *high watermark array*. The FC informs an application when the resource level crosses an entry, in the decreasing (or increasing) direction, in the low (or high) watermark array.

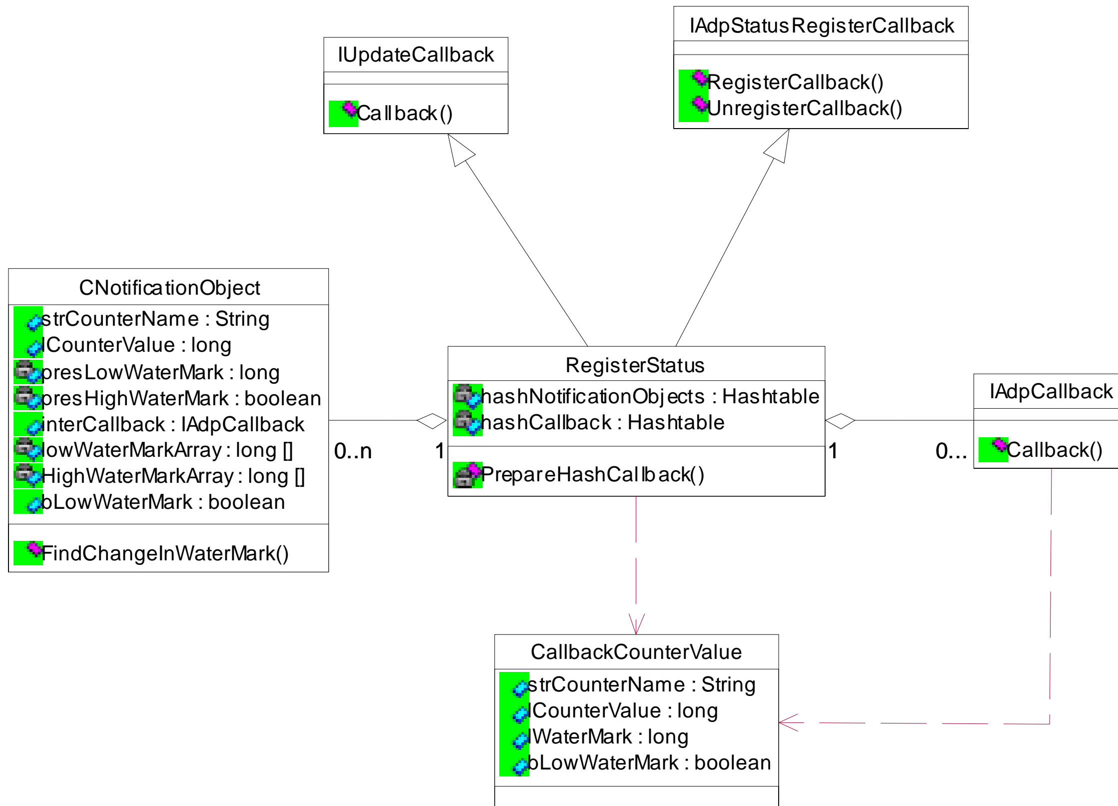


Figure 15: Class diagram of Feedback Component.

The FC also needs to allow the PMS to inform it of changes in resource levels. The PMS maintains a hashtable of resources whose levels have changed and their corresponding values. A reference to this hashtable is passed to the Feedback Component implemented by *RegisterStatus* (see Figure 15). Implementing FC as an *RMI server* in the same process space as the PMS removes the need to marshal[7] the argument (hashtable); still retaining the flexibility to implement it in a process space different from the Passive Monitoring Service, should such a need arise.

[7] Marshaling is the process of converting the data structures (sometimes nested data structures) to a flat network byte order that can be transmitted to a receiver. The receiver needs to unmarshal the flat array of bytes received into the intended data structure that can be used.

4.3.3 Processing details of the Feedback Component

The following sections presents a detailed algorithmic description of the FC along with a brief description of data structures used and design constraints / limitations.

4.3.3.1 Interface description

As mentioned in *Section 4.3.2*, the FC needs to be implemented as a *RMI Server* primarily supporting two interfaces: *IUpdateCallback* to allow the PMS to communicate any changes in resource levels and the *IAdpStatusRegisterCallback* to allow registration of policy control mechanisms. Interaction among interfaces is shown in Figure 16. While the process of registration and deregistration of policies is shown in Figure 17.

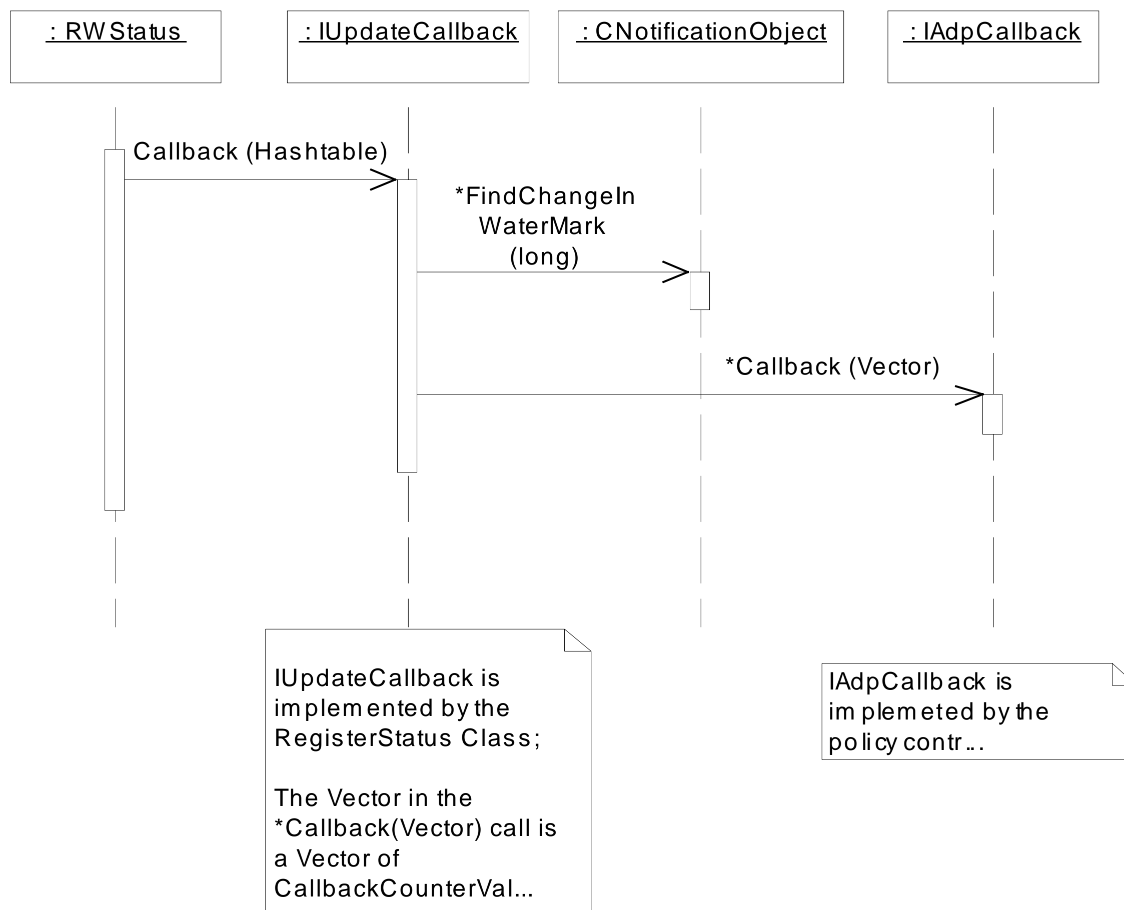


Figure 16: Interaction diagram of the Feedback Component.

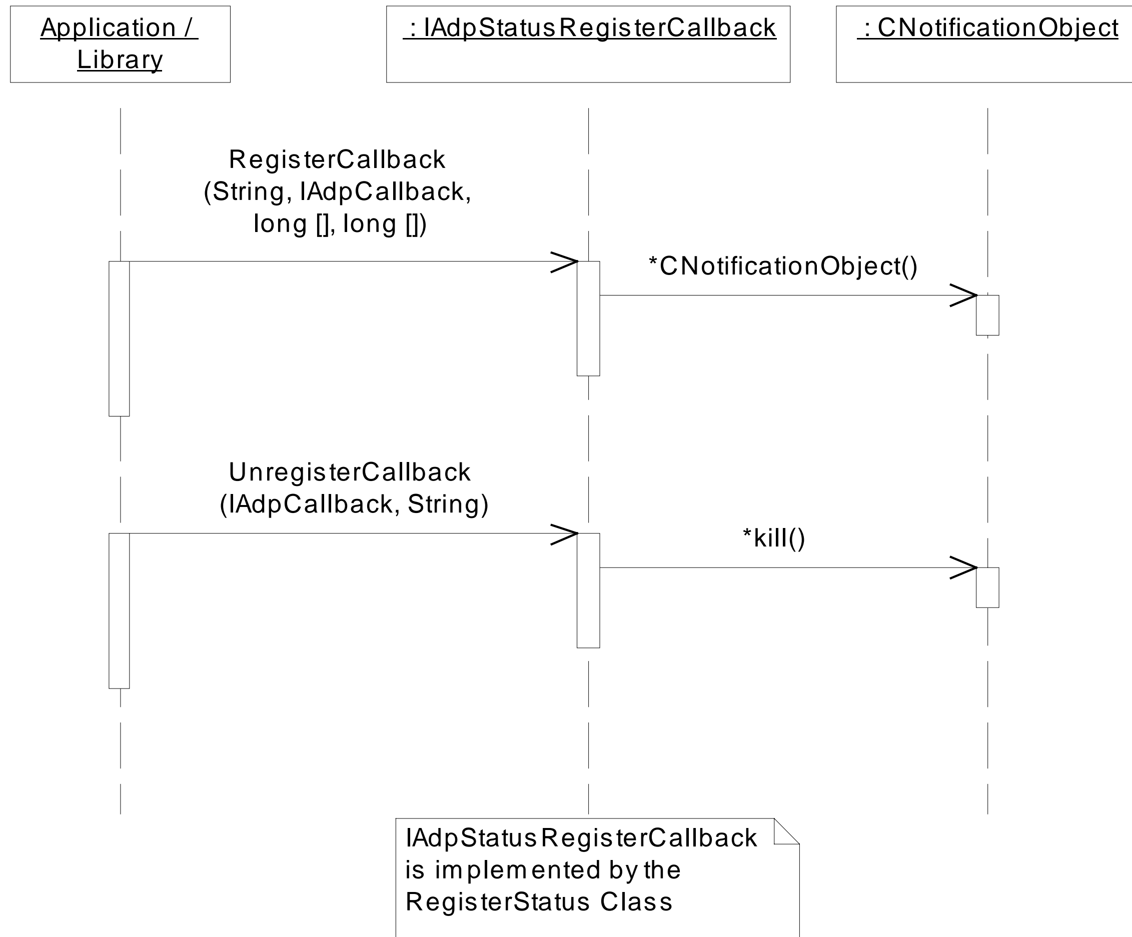


Figure 17: Interaction diagram for registering and deregistering for resource change notifications from the Feedback Component.

4.3.3.1.1 An API to specify changes in resource levels

The Java™ interface IUpdateCallback, allows the PMS or any other application to convey changes in resource levels to the Feedback Component (see Figure 16). Specifying changes is done through the following method:

Callback: This method accepts an argument of hashtable with the name of resource counter (as explained in *Section 4.2.3.1.1*) as the key and the value of the resource counter as a 32-bit value. If the user of API executes in the same process space as the FC, then little

overhead is incurred for (un)marshaling the hashtable since only the reference to the hashtable needs to be passed as an argument. The method returns a Boolean result indicating the success or failure of the operation.

4.3.3.1.2 An API to register policy control mechanisms

The Java™ interface IAdpStatusRegisterCallback allows for registration or deregistration of policy control mechanism (see Figure 17). The interface has two methods:

RegisterCallback: This method accepts the name of resource counter (as explained in *Section 4.2.3.1.1*) an object implementing the IAdpCallback interface, the low watermark array, the high watermark array and a string specifying a context. The context identifies an instance of the application, its resources of interest, its policies and the policy control mechanism in use. That is, the context string is the glue that holds the policy user, policy manager and policy control mechanism together (see *Section 4.4.1*). Context strings are particularly useful to dynamically change the policy control mechanism in use. If the resource being registered is already being monitored, then RegisterCallback method returns the current value of the resource otherwise it returns zero.

UnregisterCallback: This method accepts the name of resource counter to be deregistered along with the string specifying the context. This method returns the latest value of the resource level before deregistering.

4.3.3.1.3 Policy control mechanisms

Objects implementing the Java™ interface IAdpCallback serve as policy control mechanisms. The *IAdpCallback* interface has a single method:

Callback: This method passes a single argument that is a vector of *CallbackCounterValue* objects (see *Section 4.3.3.4*). The Callback method has to be implemented by the application developer to introduce the desired policy changes. In effect this method implements the policy control mechanism.

4.3.3.2 Algorithmic model for Feedback Component

As mentioned earlier in *Section 4.2.1*, the FC has to filter the resource level changes received from the PMS and the applications, and dispatch them appropriately to the policy control mechanisms. The algorithm can be briefly described as:

1. Extract all the resource counter names and their resource values from the hashtable received from the Passive Monitoring Service.
2. For each resource name retrieved in step 1
   a. The notification objects (see *Section 4.3.3.4*) associated with the resource are checked to find if the new value crosses any watermarks (low or high).
   b. If the value crosses any of the watermarks, an object of *CallbackCounterValue* (see *Section 4.3.3.4*) is created and appended to the vector (of *CallbackCounterValue* objects) mapped to the appropriate policy control mechanism.
3. If the vector mapped to a policy control mechanism is non-empty, then notify the policy control mechanism of any relevant changes by passing the vector of *CallbackCounterValue* objects.
4. Individual policy control mechanisms now need to interpret the CallbackCounterValue objects to deduce the status of a resource. Based on such deductions, the policy control mechanism may choose to change the current policy in use.

   The change in policy will bring about the adaptive behavior of the application.

4.3.3.3 Restrictions or limitations in Feedback Component

Specifying the low-level and high-level watermarks might not be possible in some situations. For instance, if a user is interested in every change happening with a particular resource, then the size of watermark arrays can grow prohibitively large for registration purpose. In some cases a user may only be interested in relative increase or decrease with a resource value, say an increase of 10% or decrease of 20%, presently the watermark arrays do not provide the capability for relative changes.

4.3.3.4 Data structures used of Feedback Component

The *CNotificationObject* stores the watermark arrays specified by the client, the value of resource counter during previous update and the enclosing (that is both the low and high) watermarks for the stored resource counter value. This class has one important method:

FindChangeInWaterMark: This method accepts a 32-bit resource counter value parameter. It returns true if the new value has crossed the enclosing watermarks else it returns

false. On every call it also determines the enclosing watermarks for the new value of the resource counter.

The *CallbackCounterValue* has four fields to help in identifying the change in resource level: the resource counter name, 32-bit resource counter value, watermark crossed, a boolean indicating the direction of crossing. If the boolean value is set to true (or false), then the watermark crossed corresponds to the one in the low (or high) watermark array. This class has no methods (see Figure 12).

For the purpose of filtering the relevant resource change notifications, the FC maintains two specific map data structures:

- The first one contains the mapping between resource counter names and objects of *CNotificationObject* class.
- The second one contains the mapping between process control mechanisms and a vector of *CallbackCounterValue* objects relevant to them.

Notification objects are created during policy control mechanism registration. A notification object exists for every combination of policy control mechanism and its associated resource.

4.3.3.5 Performance issues of Feedback Component

The design of Adaptation Middleware can be improved with efficient use of low and high watermark arrays. By specifying the high watermarks slightly high than the low watermarks can shield the application from policy thrashing. For instance, a low watermark array of (5, 10, 15) and a high watermark array of (7, 12 17) will give more stability to adaptive application; if the application is notified when the resource level falls below watermark 5, then the application is not informed until the resource level increases beyond 7. After an increased-beyond-7 notification, the resource level has to fall further below 7, that is 5, before the application is notified again. Hence a resource constantly changing between values in the range (4…8) will produce far lesser policy changes with the afore-mentioned watermarks.

FC can be a potential bottleneck when the number of resources and client registrations for notifications increase. Hence to improve the performance of the algorithm in *Section 4.3.3.2,*

thread pooling[8] can be used at two places: searching the vector of notification objects and also for informing the individual clients. Each client executes the policy control mechanism in the context of the thread from the thread-pool.

4.3.3.6 Design constraints of Feedback Component

The overall design constraint for the Adaptation Middleware is the same as that of Java™. The FC is constrained by scalability of data structures like maps, vectors and hashtables in Java™. Similarly, scalability of threads in Java™ determines the scalability and the concurrency behavior of the Feedback Component.

4.4 POLICY MANAGER

This section describes the PM in more detail, exploring the issues of data structures and interfaces used within the sub-system, and some of the limitations and constraints in design.

4.4.1 Processing narrative for Policy Manager

The PM acts uses a storehouse of policies, see *Section 4.1*. Each policy implements an algorithm. For instance, finding the entry to be replaced in a cache can evoke a number of cache replacement schemes such as replacing the least-recently-used-entry, least-frequently-used-entry, first-entry-that-was-cached, etc. The process of finding the next 'replaceable' entry in the cache depends on the algorithm used. Each such algorithm, in a collection, can be implemented as a separate policy.

The Policy Manger uses a context string to identify the association between a client and the collection of policies of interest to the client. Hence both the policy control mechanism (or the Feedback Component) and the client must indicate the *Context* in a request to use or change current policies. Usage of the context string allows the clients to reuse a single PM and a single policy control mechanism — for different or at times a combination of policy-collections — to change more than one current policy in use.

---

[8] A thread-pool is a collection of pre-spawned threads that can execute concurrently without imposing the overhead of creation / termination of the threads involved.

On receiving a request to use a collection of policies, the PM 'loads' if necessary, the policies in a separate namespace [5] . The PM also creates the mapping between the context and the collection of policies. The clients can begin to use the loaded policies as a normal method call, unaware of which exact policy (or algorithm) is currently in use. (For details see Appendix C: Framework for an Adaptive Application).

The policy control mechanism helps in changing the current policy in use. Policy control mechanisms are triggered by changes in resource counter value leading to a watermark crossing. The set of resource counter values in effect represent the execution environment of the application. The application is oblivious to the changes in its environment, the corresponding change in its behavior and to adaptation in general. Hence, an application developer has no overhead while developing an application. Developer need concentrate only on the semantics of the application and need not to be concerned with the adaptation behavior.

4.4.2 Description of interfaces in Policy Manager

As mentioned earlier in *Section 4.1*, the PM must support two important interfaces: *IPolicyManager* to allow clients to register and deregister policies and *IpolicyManagementUser* to allow clients to request of policy and also to change the client's current policy.

The client identifies a collection of policies (or collection of algorithms) by a name. Policies belonging to a collection need to implement a Java™ interface that defines the methods supported by those policies. Preceding the name of policy-collection by letter "I" gives the name of interface to be implemented by those policies. The policy interface itself has to inherit a dummy *IPolicy* Java™ interface having no methods (see Appendix C: Framework for an Adaptive Application).

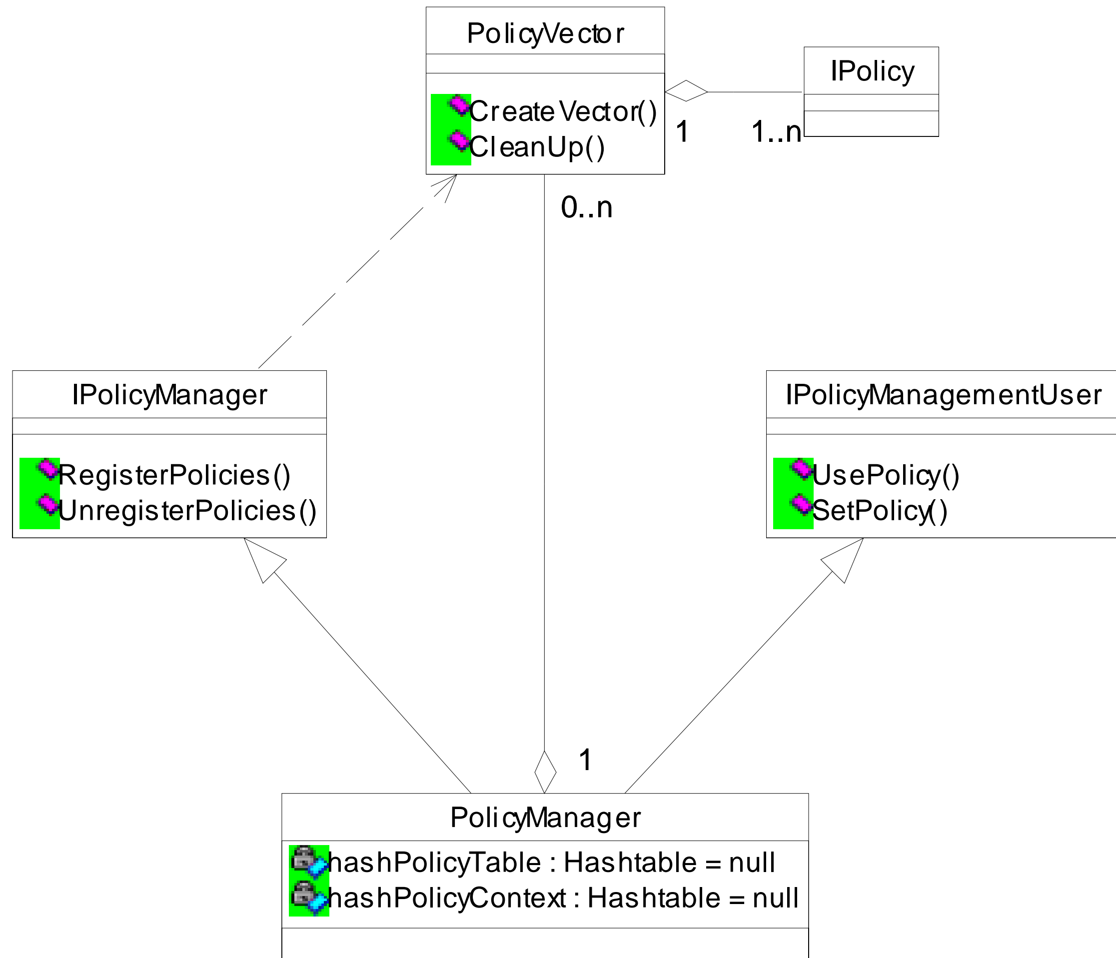


Figure 18: Class diagram of Policy Manager.

The Java™ interface *IPolicy* is necessary for the uniform handling of policies. Creation of a policy-vector is needed to store the set of policies belonging to the same policy-collection. Since policies implement custom interfaces that are independently defined by the policy (or library) developer[9], PM imposes the requirement that all policies should implement *IPolicy* interface. As *IPolicy* by itself has no methods in it, no restriction is made on the number of

[9] Development of algorithms might also be the job of a more specialized *library developer*. The application developer need only to use the algorithms or policies without concern for the implementation details.

methods, the names of those methods or the parameters and return types of methods in a policy. Hence, an application developer involved in developing the algorithms is free to define any Java™ interface with no restrictions except that it has to inherit the *IPolicy* interface.

4.4.3 Processing details of Policy Manager

The following sections present a detailed algorithmic description of the PM along with a brief description of data structures used and design constraints or limitations.

4.4.3.1 Interface description

Implementation of PM imposes more restrictions on the development of policy-collections than those specified in *Section 4.4.2*. As mentioned in *Section 4.4.2*, each collection of policies is identified by a name. Polices in a collection need to implement a Java™ interface that specifies the method names and their parameters to invoke a policy. The name of the interface is derived from the name of the policy-collection by prefixing the letter "I." All such interfaces defining the methods of a policy need to inherit the *IPolicy* Java™ interface. In addition to defining a policy interface and implementing the policies, the policy developer needs to implement a proxy (or reference indirection) class; a sample indirection class is shown in Appendix C: Framework for an Adaptive Application. Prefixing the letter "C" to the name of policy-collection derives the name of the proxy or the reference indirection class. Details on packaging of policies can be seen in Appendix C: Framework for an Adaptive Application. Interaction among interfaces is shown in Figure 19.

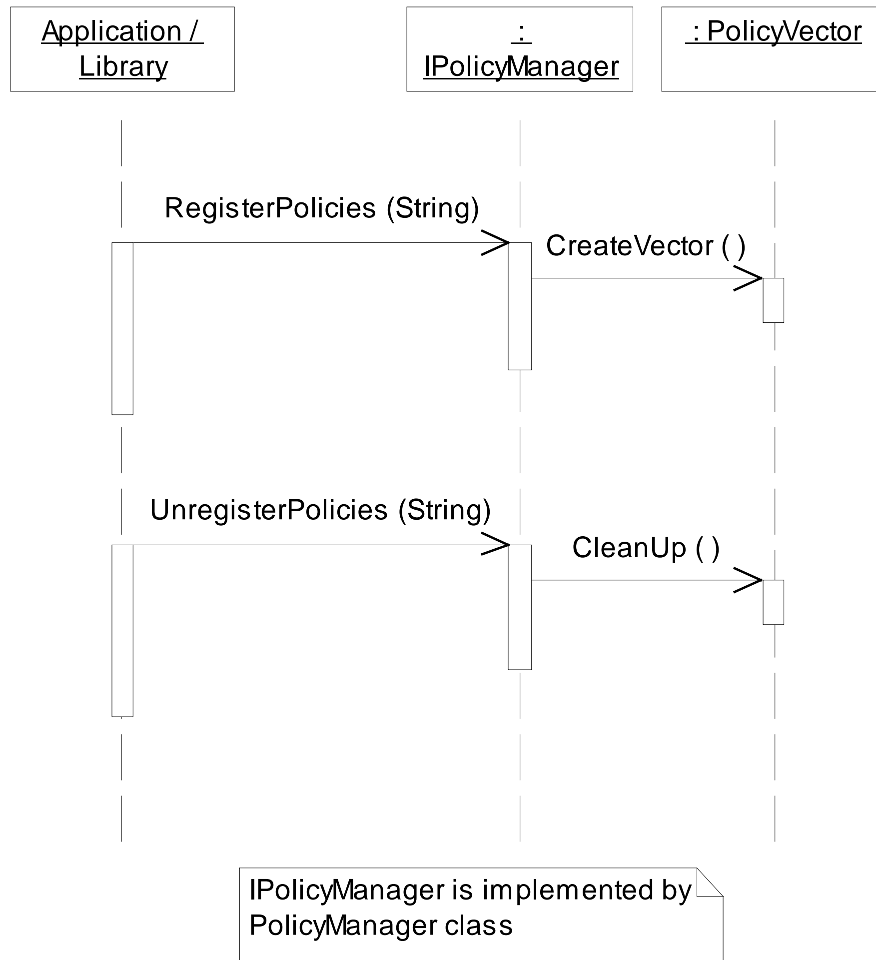


Figure 19: Interaction diagram to register and deregister policies.

4.4.3.1.1 The reference indirection class to be implemented by the policy developer

The reference indirection class needs to implement all methods defined in the policy interface. Each implementation is a trivial forwarding call to the *forwarding interface reference* held. In addition to the methods defined in the interface, the indirection class needs to implement two methods that are not defined in any Java™ interfaces, but whose signatures are fixed. These methods are:

SetPolicyPointer: This method accepts the policy interface (defined by the policy developer) as the first argument and the policy number[10] as the second argument. The function needs to set the forwarding reference to the first argument passed and also store the policy number in use. The method has no return values.

CleanUp: This method accepts no arguments. *CleanUp* should set the forwarding reference to null value and if hot-swapping[11] needs to be supported it should invoke the garbage collector that is specific to the proxy. The method should return the latest policy number in use.

The PM must support two important interfaces: *IPolicyManager* to allow clients to register and deregister policies and *IpolicyManagementUser* to allow clients to request usage of policy and also to change their current policy.

4.4.3.1.2 An API to register and deregister policies

The Java™ interface *IPolicyManager* allows for registration and deregistration of policies (see Figure 19). It has two methods:

RegisterPolicies: This method accepts a string parameter specifying the name of policy, described in *Section 4.4.2*, that refers to a collection of policies. The PM loads the specified collection of policies if they are not already loaded in a separate namespace. The method also creates a mapping between the name of the policies loaded and the namespace in which they are loaded. The method returns a Boolean result indicating the success or failure of the operation.

UnregisterPolicies: This method accepts the name of the policy-collection as a parameter. The method removes the mapping between the name of policy-collection and the name-spaces, allowing the name-space to be garbage-collected when no outstanding references to it remain. The method also returns a Boolean result indicating the success or failure of the operation.

---

[10] Policy number is the index of the policy in a collection that is sorted in the alphabetical order. For instance, if a policy-collection has three policies: Policy1, Policy2 and Policy3, then the policy numbers associated with them are 1, 2 and 3 respectively.

[11] Hot-swapping is the process of loading new or changed versions of policies transparent to the application.

4.4.3.1.3 An API to request usage and change of policies

The Java™ interface *IPolicyManagementUser* allows a client to request the usage of a policy and also to change the current policy (see Figure 18). It has two methods:

UsePolicy: This method accepts two parameters, the name of the policy-collection and a context string (see *Section 4.4.1* for discussion on context strings). This method returns a reference to the proxy object (see Appendix C: Framework for an Adaptive Application for an example of proxy object that implements indirection). The *Section 4.4.3.2* explains the details of the indirection mechanism.

SetPolicy: This method accepts two parameters, the context string and the policy number selected for change. This is the method that must be used by the policy control mechanism or the application to change the current policy. The method returns a Boolean value indicating the success or failure of the operation.

4.4.3.2 Algorithmic model for Policy Manager

The reference indirection class needs to maintain a forwarding reference of the policy interface type. The goal of the reference indirection class is to isolate the lifetimes of policies and policy user (or the client). Maintaining different name-spaces for each of the policy-collections and forwarding the entire client calls to policies through a forwarding reference achieves a loose coupling among the clients and their policies.

During registration of policies, the PM accepts the name of the policy-collection and loads all the classes in the *<CLASSPATH>\<policy-collection-name>*[12] directory. The PM also maintains a mapping between name of the policy collection and the policies.

On the request by a client to use a policy, the PM instantiates an object of the proxy or the reference indirection class and returns the reference to proxy to the client. The client being oblivious to the presence of a proxy makes method calls to the proxy object, which are duly forwarded to the appropriate policy.

---

[12] CLASSPATH is a system environment variable that stores the directory path to all "*.class*" and "*.jar*" files. A Java™ class file in the CLASSPATH directory is loaded by the default classloader in JVM. Enabling custom loading of policy classes requires that the classes not be in any directory mentioned in CLASSPATH.

4.4.3.3 Restrictions or limitations in Policy Manager

PM in particular and adaptation in general impose the need for policies to be stateless. That is a policy (or algorithm) cannot internally store, the state[13] of an operation between calls. Since the policies being used can change between calls by the client, policy implementations should be stateless. State information can be stored in an abstraction (like a file) outside the policy.

The interface for a policy is not restricted by the number of methods, method names, parameters or return types. The only requirement being that arguments and return values be serializable, in other words, they need to inherit the Java™ *java.io.Serializable* interface that has no methods to implement. As mentioned earlier, the interface for policies must inherit the *IPolicy* interface (as shown in Appendix C: Framework for an Adaptive Application).

From the point of view of clients, the context strings need to be unique across instantiations. Although the context string has no restriction, it needs to be predictable to enforce policy changes through policy control mechanisms. A simple automated generation of context strings could be effected by appending user-name to a fixed string or a string input from the user.

4.4.3.4 Data structures used in Policy Manager

The PM has a *PolicyVector* class that stores the list of policies in a policy-collection. The *PolicyVector* also has a custom classloader for each policy-collection. The name of the policy-collection is passed to the *PolicyVector* in its constructor. The *PolicyVector* has two methods:

CreateVector: This method accepts no argument. It creates a custom classloader and loads the policies in a policy-collection. This method has no return types.

CleanUp: This method is used to release the custom classloader created in the *CreateVector* call. The method has no return types.

The PM needs to maintain a map of context string and the indirection object for the policy. When a call to *SetPolicy* is made either by the application or by the policy control mechanism, the PM uses the Java™ Reflection API [10] to call the *SetForwardingPointer* method of the proxy object with appropriate policy number.

---

[13] State of an operation between calls is usually stored in *static* members within the method. Policy implementations cannot have static members.

The PM also needs to maintain the map of name of policy-collection and the list of policies loaded from that policy-collection, through the *PolicyVector*. This map is useful to return the appropriate policy object given the policy number. Retrieval of policy given the policy number is necessary when the client requests a policy for use or when the policy control mechanism specifies a change in current policy.

4.4.3.5 Performance issues of Policy Manager

In the above discussion we have advocated that policies be stored in the Policy Manager. Some compromises in performance result with this model: every call to a policy needs to be an inter-process communication. However, this design provides the conditions for hot-swapping that might be introduced in later versions (see *Section 6.3.1.2*).

4.4.3.6 Design constraints of Policy Manager

As mentioned earlier, the overall design constraint for the Adaptation Middleware is the same as that of Java™. The PM is constrained by scalability of data structures like maps, vectors and hashtables in Java™. Similarly, robustness of inter-process communication in Java™ will determine the robustness in execution of policies.

4.5 FLEXIBILITIES IN DESIGN

The following section describes some of the flexibilities in using the Adaptation Middleware. The flexibilities can be viewed in the perspective of an application development, policy (or library) development and application tuning.

The purpose of the Adaptation Middleware is to provide support for application-aware adaptation, as explained in *Section 1.2*. A objective is to shield the application developer from many complexities, while representing a manageable load for the system. There are two important flexibilities afforded to the application developer:

The developer can treat system resources and application-specific resources with no distinctions. Thus the applications are not limited to adapting to environment changes around them, but also to resource changes occurring within them.

The change needed in a client's code is limited to a *UsePolicy* method call. The client is then unaware of adaptation. It uses the reference obtained by the *UsePolicy* call as just another

reference to a remote object; in effect allowing the application developer to focus on the application semantics rather than adaptation.

Policies are separated from applications, which means they can be developed as libraries (or Java™ packages) and integrated with the Adaptation Middleware. Further, policies can be implementations of any algorithms that can execute with user privileges. The middleware imposes no restriction on either the number of methods, their names, arguments or return types. At times, the developer of a policy will be a good judge to decide when to use a particular policy. In such situations, the policy developer can also supply the correct policy control mechanisms.

The policy control mechanism at the server can control the policy being used in the client. This property is especially useful in coordinating policies among clients and servers in a real-time application like video-on-demand. These applications need policy changes at both ends of a communication channel. For instance, depending on the available bandwidth and error rate, the server can choose to change the encoding scheme or the compression algorithm used. This change has to be effected at both ends of a communication channel. The design of PM as an RMI server makes this policy change possible.

CHAPTER 5 PERFORMANCE EVALUATION

This chapter briefly describes the Monte Carlo simulation technique and two perspectives on evaluating the performance of an adaptation system: Application perspective and System perspective. The middleware being simulated is predicted to lead to a more intelligent cost estimation for applications like mobile agents. One of the objectives of this chapter is to verify this prediction.

5.1 INTRODUCTION TO THE MONTE CARLO SIMULATION TECHNIQUE

In many applications of Monte Carlo [4] [13] , the physical system is directly simulated, that is, the differential equations describing the behavior of the system are not written. The only requirement is that the system should be described by probability density functions (pdf's). Monte Carlo simulation then proceeds by random sampling from the pdf's. Multiple trials are performed and the desired result is found as an average over the number of observations, similarly the variance in average result is also calculated.

Monte Carlo methods can be used to simulate random processes, since these can be described by pdf's. For cases having no apparent stochastic content (or randomness), the desired solution is derived in terms of pdf's. This representation allows the system to be *treated* as a stochastic process for the purpose of simulation; Monte Carlo methods can be applied to simulate the system. Hence, Monte Carlo methods include all methods that involve statistical simulation of an underlying system, whether or not the system represents a real physical process. The essential characteristic of Monte-Carlo simulation is the use of random sampling techniques to arrive at a solution to the physical problem.

For the Adaptation Middleware under consideration, differential equations cannot be written to describe the behavior. Input data for the system includes the changes in resource levels of the objects / resources required by the application. Since the clients can specify the threshold levels for a given resource, not all changes are applicable to all clients. Hence capturing the behavior of the entire system through differential equations is not practical. Some experiments are done to observe the changes in resource levels of certain resources. Most of the observations agreed upon a combination of one or many of exponential / normal / uniform distribution systems.

5.2 DUAL PERSPECTIVES

This section presents two perspectives in evaluating the performance of an adaptation system: Application perspective and System perspective.

5.2.1 Application perspective

Most of the systems providing support for mobile agents base their agent migration policies on some resource level, for instance the communication cost required to reach a node. The policies in agent systems presently fail to arrive at a compromise between the priority of getting a job done (essentially dependent on the availability of the computing resources) and the cost involved in getting it done (dependent on the availability of the communicating resources). For example, reaching a node in the network may be less costly — the communicating cost for reaching the node may be less — but the node may not have the required resources to accomplish the job. Selection of such a node unnecessarily leads to further migration of the agent to another node; thus, adding to the delay experienced in accomplishing the task rather than getting the agent to reach the client. Providing the computing resource levels of individual clients and servers allow the clients to arrive at a more optimal choice of server. The agent first hops to a node depending on the availability of resource and minimum communication cost (if communication cost is also considered). If the request is unsatisfied at the migrated node, the agent further migrates to the next node depending on its migration policy.

A parameter that indicates the performance is the number of hops the agent makes in order to accomplish the job at hand.

5.2.2 System perspective

Performance of the middleware depends on the type of system. However, a few concerns must be supported by all adaptation middlewares. Some such issues are scalability, portability of the middleware to a new platform, flexibility (in adding new policies, in changing the policy control mechanism, possibility of hot-swapping of policies…), fault-tolerance, load balancing and security.

One other such concern is *agility* [16] . Ideally, the client must be able to detect change in the resource levels instantaneously. However, a lag always occurs between the time at which

the resource level changes and the time at which the client becomes aware of the change. *Agility* of the adaptation system is the speed with which the client becomes aware of the change in the resource level. We measure this time lag through a series of Monte-Carlo simulations.

5.3 TUPLE SPACE MANAGER: A SAMPLE APPLICATION

This section describes briefly the Tuple Space Manager (TSM), a sample application developed to measure its adaptation. This section also describes the types of policies used in the application.

5.3.1 Tuple Space Manager

Tuple Space Manager (TSM) on each host stores two tables. First is a Hashtable storing the tuples of the object types and the second is a vector table storing the number of objects of each type on other hosts (as per the latest information on this host).

When an application requests a tuple T, TSM checks in its local tuple-space for a match. If a match is not found, then it finds the remote host that has to be queried, depending upon the current policy. It then forwards the request to the TSM on the remote host via the Agent Architecture (AA), which creates an agent and sends it to the remote host.

When a host receives an agent, the AA on the remote host launches the agent in a separate thread. The agent then forwards the request for tuple T to the remote TSM (on the migrated node). In case a match is found, a reply-agent is sent back to the requesting host. Otherwise, the agent is further migrated, based on the current policy in the remote TSM. The remote TSM stores information regarding the unsatisfied tuple T and the requesting host.

On finding a match for the tuple T, the TSM removes the tuple T from its tuple-space. The table containing the *number of tuples* for the type of tuple T is updated, the table now reflects the change in number of tuples in the tuple-space.

Since a trail of the request-agents is left at remote TSMs that do not contain matching tuples, a possibility exists that the tuple becomes available at the remote host after the agent has migrated to another host. In the meantime, the agent would have migrated to another host that contains the requested tuple. Now, all remote-hosts that satisfy the request will send back the matching tuple to the source host. In this case, the source host will receive more than one tuple. It uses one of them to satisfy the query and makes all other tuples as its out-tuple (that is to store

them in its tuple space); it also modifies an entry in a table to reflect the changes in the number of tuples. It is possible that the requested tuple becomes available in the tuple-space of the source host, and then the source will convert all the received tuples as out-tuples.

The tuple-space is designed in hierarchical manner. A collection of TSMs forms a *cluster* that has one of them functioning as a *cluster-head*. If no TSM in the tuple-space is able to satisfy a request, then the request is passed to the *cluster-head*, which in turn passes the request to other clusters known to it. The request also stays alive in the TSMs of the present cluster.

5.3.2 Types of policies

Policies in a TSM are mainly those concerning agent migration, that is, where to start looking for a particular tuple. The availability of resource (in our case the tuple) should dictate the policy. In this respect, we have identified two types of policies:

- *Object count policy*: Agent migrates to the node having the maximum number of tuples of a particular tuple-type T.
- *Queries satisfied policy*: Agent migrates to a node that has satisfied the maximum number of client requests for tuple-type T.

The *default policy* involves an agent migrating to hosts in a fixed order. The performance of above policies should be better than the *default policy*. The number of hops the agent makes to satisfy a request is a fair indication of how a policy can affect the performance.

5.3.3 Types of policy control mechanisms

Following are the policy transitions (or policy control mechanisms) that are introduced in the TSM:

With no policy control mechanism: The agent always migrates to a fixed next-node using *default policy*.

Policy-Control-Mechanism-1: Agent migration policy — to fetch a particular tuple (T) — changes from *default policy* to *object count policy* when the number of tuples of type T, in a remote TSM, exceeds ten. *Policy-Control-Mechanism-1* has no policy transition back to the *default policy* (see Figure 20).

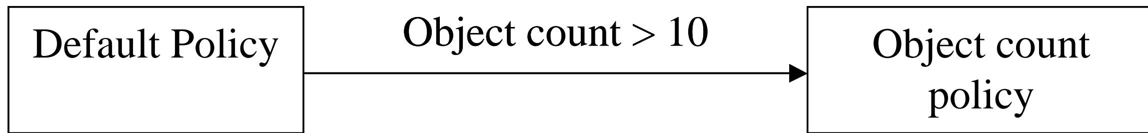


Figure 20: State diagram depicting Policy-Control-Mechanism-1

Policy-Control-Mechanism-2: Agent migration policy — to fetch a particular tuple (T) — changes from *default policy* to an *object count policy* when the number of tuples of type T, in a remote TSM, exceeds ten. The policy changes to *queries satisfied policy* if the number of requests satisfied by a TSM exceeds ten. *Policy-Control-Mechanism-2* has no policy transition back to the *default policy* (see Figure 21).

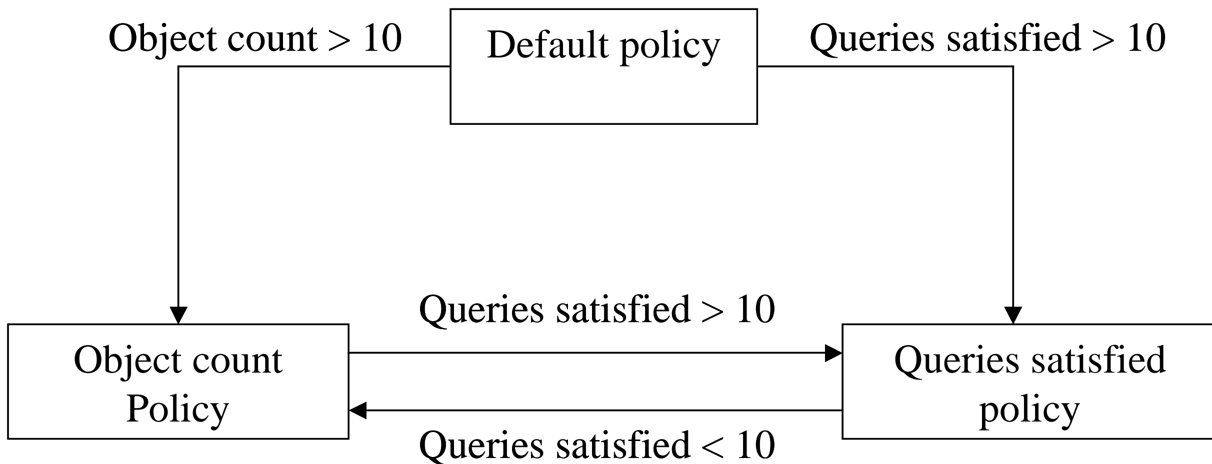


Figure 21: State diagram depicting Policy-Control-Mechanism-2

5.4 APPLICATION PERSPECTIVE

This section discusses the performance of a sample application — TSM, with and without policy control mechanisms. The two policy control mechanisms used are explained in *Section 5.3.3*.

As mentioned earlier in *Section 5.2.1,* for an agent-based system (or application) the number of hops required by an agent is a fair indication of the performance of the system. In particular, lesser the number of hops required by an agent to finish the job at hand, the better is

the performance of the system. TSM being developed on the Adaptation Middleware uses agents to fetch tuples from other TSMs. The metric *hop count per tuple* — the ratio of total number of hops to total number of tuples fetched outside the TSM (or host) — required to satisfy all the requests of an application that uses the TSM is an indication of performance of the TSM.

The performance of TSM can be simulated by developing a consumer-application that generates request for tuples along with another producer-application that introduces new tuples into the tuple-space. Producers and consumers of tuples can also be separate threads in the same process. Performance evaluation involves simulation of three TSMs in a network. Each application being developed on the TSM is both a producer and a consumer. For the purpose of simulation, three types of tuples: T1, T2 and T3 can be considered; the TSMs can be identified as sites (or TSMs) A, B and C. Figure 22 shows the distribution of various types of tuples in a site. Also, introduction of new tuples to a TSM and generation of requests for tuples occur concurrently.

	Site A (Cluster-head)	Site B	Site C
Tuple of type T1	0.20	0.30	0.50
Tuple of type T2	0.50	0.20	0.30
Tuple of type T3	0.30	0.50	0.20

Figure 22: Ratio of number of tuples of a particular type to the total number of tuples of all types in a site.

For *default policy*, the next hop for an agent in Site A is Site B, for an agent in Site B is Site C and for an agent in Site C is Site A. Figure 23 shows distribution of number of requests generated for a particular type of tuple in each site. Site A also functions as a *cluster-head*.

	Site A (Cluster-head)	Site B	Site C
Tuple of type T1	0.50	0.30	0.20
Tuple of type T2	0.20	0.50	0.30
Tuple of type T3	0.30	0.20	0.50

Figure 23: Ratio of number of requests generated for tuples of a particular type to that for the total number of tuples of all types in a site.

5.4.1 Performance improvements of application with policy management

Simulation runs — replications or trials — for different policy control mechanisms were conducted for *1200* productions of tuples and *1000* requests for tuples at each site (for types of policy control mechanisms, see *Section 5.3.3*). Chart 9 in Appendix B: Results of Performance Evaluation shows ten observations made for each policy control mechanism. The metric *hop count per tuple* for *Policy-Control-Mechanism-1* shows improvement in performance — **6.5%** and **1.18%** for **Sites B** and **C** respectively — when compared with those for the *default policy*. Results also show good improvement in performance — **14.3%** at **Site A** — for the *cluster-head* when compared with that for the *default policy*.

Similarly, when the performance metric — *hop count per tuple* — is compared between *default policy* and *Policy-Control-Mechanism-2* there is a significant improvement in performance at all the three sites. The performance improvements are **31.25%** at **Site A** (*cluster-head*), **53.75%** at **Site B** and **33%** at **Site C**.

From the observations made above we can conclude that right choice of policies and policy control mechanisms improve the performance of an application.

5.4.2 Comparison of performance for different policy control mechanisms

From Charts 12 & 13 in Appendix B: Results of Performance Evaluation we can conclude that Policy-Control-Mechanism-2 shows improved performance when compared with Policy-Control-Mechanism-1. The performance improvements in terms of hop count per tuple

are **19.8%** at **Site A** (cluster-head), **51%** at **Site B** and **32%** at **Site C**. As seen in Figure 24, the performance improvements in terms of number of tuples fetched from other sites are **26.9%** at **Site A** (cluster-head), **26.7%** at **Site B** and **26.6%** at **Site C**.

	Site A (Cluster-head)	Site B	Site C
Without policy control mechanism	972	991	995
Policy-Contol-Mechanism-1	970	987	986
Policy-Contol-Mechanism-2	709	723	724

Figure 24: Average number of tuples fetched outside the TSM.

An interesting observation is the standard deviation in the number of tuples fetched other TSMs (observed for *Policy-Control-Mechanism-2*), **288** for **Site A** (*cluster-head*), **295** for **Site B** and **271** for **Site C**. In the case of *Policy-Control-Mechanism-2*, the distribution for the number of tuples fetched from TSMs outside the current machine is bimodal. The first mode — in the range 950 to 1000 — is comparable to that of *Policy-Control-Mechanism-1* and *default policy*. But there is a significant reduction in number of tuples fetched from outside, in the case of second mode: range 300 to 500.

To investigate the reduction in number of tuples fetched from outside the TSM, more replications were conducted with lesser number of tuples — 500 and 750. In both the cases, the distribution of number of tuples fetched from other TSMs is bimodal. No apparent reason can be attributed to this behavior. Even though *Policy-Control-Mechanism-2* is more efficient in terms of *hop count* (and the number of tuples fetched from other TSMs), it lacks deterministic behavior and hence it lacks the confidence of the application user.

5.5 SYSTEM PERSPECTIVE

The performance of the Adaptation Middleware can be measured in terms of *agility* —lag in time between the change in resource level and client becoming aware of the change — in adapting to changing resource levels. Studying the performance of the Adaptation Middleware

for input data following the exponential, uniform and normal distributions will give a better understanding of the behavior of the middleware.

Simulation is carried out for 8 replications of 1000 resource level changes for each type of distribution followed. The time lag between the change in resource level and corresponding notification to the policy control mechanism is determined. The performance of the FC — and hence the Adaptation Middleware — depends on the number of client registrations for resource-level change notifications[14]. To minimize the overhead, writing the resource values to the Directory Service can be skipped.

A sample scatter-chart depicting the time lag for 88 policy changes (spread over 200 resource level changes) is shown in Chart 10 in Appendix B: Results of Performance Evaluation. There is no correlation among the data sets in the scatter-chart. The time lag varies between 70 milliseconds (0.07 seconds) and 1128 milliseconds (1.128 seconds) with the average time lag around **568 milliseconds** (or 0.568 seconds) and a standard deviation of **305 milliseconds**.

The *WriteAPI*, see Section 4.2.3.1.2, performs a periodic update — every one-second — to the Passive Monitoring Service; the average delay introduced by the *WriteAPI* should itself be approximately 500 milliseconds (without accounting for the delay caused by the protocol to inform the PMS). The average delay introduced by the PMS and FC is approximately **68 milliseconds**. This delay includes the inter-process communication involved in informing the resource changes to the Passive monitoring Service.

We simulate the middleware to measure the effect of damping. Damping is introduced by specifying the watermark arrays (see Section 4.3.1). We simulate the middleware with two different sets of watermark arrays — LWA and HWA[15] — for the exponential, normal and uniform distributions of input data. Results show a reduction in the number of policy changes (see Charts 1-6 in Appendix B: Results of Performance Evaluation).

---

[14] Simulation replications were conducted with three registrations for resource-level change notifications.

[15] LWA = Lower Watermark Array and HWA = Higher Watermark Array.

Exponential distribution (mean = 1)	Ratio of policy changes to resource changes
Without damping LWA = {1, 3, 5}, HWA = {1, 3, 5}	0.228
With damping LWA = {1, 3, 5}, HWA = {2, 4, 6}	0.086

Figure 25: Ratio of policy changes to resource changes for Exponential distribution with mean = 1.

Normal distribution (mean = 10, standard deviation = 5)	Ratio of policy changes to resource changes
Without damping LWA = {5, 10, 15}, HWA = {5, 10, 15}	0.441
With damping LWA = {5, 10, 15}, HWA = {7, 12, 17}	0.37

Figure 26: Ratio of policy changes to resource changes for Normal Distribution with mean = 10 and standard deviation = 5.

Uniform distribution (range: 70 … 90)	Ratio of policy changes to resource changes
Without damping LWA = {75, 80, 85}, HWA = {75, 80, 85}	0.499
With damping LWA = {75, 80, 85}, HWA = {77, 82, 87}	0.437

Figure 27: Ratio of policy changes to resource changes for Uniform Distribution in the range 70...90

Observations of policy changes for input data indicate an average reduction of:

- **62.33%** (with *7.55%* standard deviation) for exponential distribution,
- **16.03%** (with *2.73%* standard deviation) for normal distribution, and
- **12.45%** (with *4.82%* standard deviation) for uniform distribution.

The lower percentage of changes in current policies leads us to believe in a stable adaptation mechanism. Damping shields the application from *policy thrashing*[16]. From the above observations we can conclude that damping has profound effect — in reducing policy changes — for input data following exponential distribution. Exponential distribution of input data is similar to a resource having a consistent value of a resource attribute, but with occasional spikes in the values.

To investigate the effect of watermarks, we simulate the Adaptation Middleware to include damping for Uniform distribution of input data. For the first replication LWA is {75, 80, 85} and HWA is {77, 82, 87} and for the second replication LWA is {74, 79, 84} and HWA is {76, 81, 86}. Charts 7 & 8 in Appendix B: Results of Performance Evaluation shows the closer adaptation (with damping) to variations in resource levels — observe the time interval 0 to 40000 milliseconds — than those in Charts 5 & 6. This observation demonstrates the need to cautiously choose the watermarks.

[16] Policy thrashing refers to frequent changes in policies of an application leading to unstable behavior.

CHAPTER 6 CONCLUSIONS, SUMMARY AND FUTURE WORK

This chapter provides conclusions and summary of the research conducted. The chapter also provides some interesting improvisations that can be incorporated into the Adaptation Middleware. Further, the chapter discusses some challenges in the domain of application-level adaptation; these challenges can be used as pointers for future research in the field of application-level adaptation.

6.1 CONCLUSIONS

Our research set out with two primary objectives: to design portable application-independent middleware to support application-level adaptation and to enable applications to be notified of the status of network-wide resources.

The Adaptation Middleware provides portability that is constrained only by the portability of Java™ itself. Further, the design of Adaptation Middleware separates the platform-independent components from the platform-specific component (the Active Monitoring Service) by the standardized BSD socket abstraction that is present in all platforms. The Adaptation Middleware also separates monitoring of a resource from notification of resource-level changes — Active and Passive Monitoring Services from the FC — this separation allows for local monitoring of resources (for changes within the resources) and network-wide notification of relevant changes. Inclusion of Directory Service in the architecture allows for storage of resource levels, which can be accessed by the clients that are present in a network. The Directory Service also allows clients to obtain the levels of the *static* resources, see *Section 3.1*.

6.2 SUMMARY

Chapter 1 provides a brief introduction to the field of application-level adaptation. The chapter contains arguments as to why reservation cannot be made for all resources. Finally, the chapter concludes with a description of the objectives of our research.

Two existing systems — Odyssey and Cadmium — that provide application-level adaptation are described in Chapter 2. An architecture for adaptive applications, with no support from the system (middleware or operating system), is presented.

Chapter 3 provides the architectural-level design of the Adaptation Middleware. The chapter then classifies resources into three types: computing, communicating and application-specific. The effects of resources on each of the sub-systems are discussed. The sub-systems are classified as either platform-dependent or platform-independent. The MS is split into the Active Monitoring Service and the Passive Monitoring Service. The need for platform independent communication between the AMS and the PMS is also established. Finally, the chapter concludes by describing the collaboration among various sub-systems of the Adaptation Middleware.

Chapter 4 provides a detailed design of the Adaptation Middleware. The chapter describes the structure of the Adaptation Middleware, followed by the design of the three individual sub-systems. For each sub-system, the design of data structures and interfaces is described. Finally, the chapter concludes with a brief look at the flexibilities in the existing design.

The evaluation methods for measuring the performance of the Adaptation Middleware are described in Chapter 5. The chapter offers two perspectives in evaluating the performance of an adaptation system: Application and System. The results of evaluation from both perspectives are discussed.

The following sections discuss some of the improvisations that can be incorporated into the Adaptation Middleware and also provides some pointers for future-research on adaptation.

6.3 FUTURE WORK

This section describes some improvisations to the Adaptation Middleware and also some pointers to future research on application-level adaptation.

6.3.1 Adaptation middleware

Section 4.5 describes the flexibilities provided in the design of the Adaptation Middleware. The Adaptation Middleware also provides scope to include other flexibilities. Two

flexibilities deserve special mention: ability to introduce policy control mechanisms on-the-fly and hot-swapping of policies.

6.3.1.1 On-the-fly introduction of policy control mechanisms

The Policy Manager maintains the mapping of policy control mechanisms and the resources required by a client, by creating an object of *CNotificationObject* class. These mappings are stored in a hashtable data structure within the PM (see Section 4.4.3.4).

Existing mappings in the hashtable can be removed and new mappings can be created. A *context* string used in *IAdpStatusRegisterCallback:RegisterCallback* method-call identifies a client or a policy control mechanism. Context strings are also stored with the objects of *CNotificationObject* class. Context strings can be used to identify objects of *CNotificationObject* class to be removed. New objects of *CNotificationObject* class can then be created and mapped to the designated resources. A new policy control mechanism can now be introduced to replace any existing policy control mechanisms, in effect replacing the existing policy control mechanisms ‘on the fly’.

6.3.1.2 Hot-swapping of policies

On invoking registration of an already loaded policy-collection through the *IPolicyManager:RegisterPolicies* method, the PM calls *CleanUp* method in the proxy object using Java™ Reflection API. The *CleanUp* method in the proxy object can set the forward reference to null and invoke the garbage collector. Since the policies have no external references pointing to them, they can be garbage collected. The PM can now load the ‘new’ collection of policies in a different namespace and call *SetForwardingPointer* method of the proxy, which resets the forwarding pointers to their original policies.

The above procedure can allow existing policy-implementations to be changed or new policies to be added without halting the execution of the Adaptation Middleware. The only restriction while adding a new policy would be that new policy should be the last policy on sorting the policy-collection in alphabetical order. This is necessary to uphold the contracts made with existing policy control mechanisms, which request changes to current policy usage based on policy numbers (see *Section 4.4.3.1.3*). Further, an existing policy cannot be deleted, but special error conditions can be retorted to indicate that the policy cannot be used.

6.3.2 Application-level adaptation

This section poses some questions that can serve as pointers for future-research. Some of the questions related to adaptation are: When does a client decide to adapt? How can the client calculate the extent and cost of adaptation? Is it always best to adapt? Is it preferable for a client-application to be unaware of adaptation?

This remainder of this section gives a brief overview of the problems and/or challenges that can serve as pointers for future research in the field of application-level adaptation.

6.3.2.1 Domain-specific and domain-independent architectures

The Adaptation Middleware has an application (or domain) independent architecture. Further research is necessary to compare the performance — in terms of agility — of domain-specific and domain-independent architectures. Some questions that can be raised are: Given the plausibility of domain-independent adaptation systems, what compromises have to be made to implement specific applications such as client-server multimedia application, a server-side cache manager for mobile clients, and others? What performance-overhead does mobile clients involved in inter-process communication across a wireless link increase? Is adaptation always a good alternative?

6.3.2.2 Cooperative adaptation

From Chart 13 in Appendix B: Results of Performance Evaluation, we can observe a wide variation in the number of tuples fetched from outside a TSM while using policy control mechanism 2. This observation is unique to the policy control mechanism 2. The observed variation indicates very good co-operation among the policies of Tuple Space Managers. Given that adaptation policies are autonomous, how can we achieve co-operation among them on a more regular basis? The answer probably lies in the control of adaptation. There are more questions that can be raised: What makes the adaptation useful? How do we come up with good adaptation policies? When should a policy change occur; that is what decides the watermarks or the design of policy control mechanisms? Should policies be totally autonomous? If the policies are centrally controlled, then what flexibilities or features should an operating system (or the controlling system) possess? If the adaptation is controlled by an operating system (that is for a centrally controlled system), should the system be reflexive (monitor its own resources and inject the changes as a feedback)?

Much experimental work remains before general, or application-independent answers can be expected. We hope that this thesis has laid the groundwork for such research.

REFERENCES

[1] Aline Baggio, *Adaptable and Mobile-Aware Distributed Objects*, doctoral thesis, Project SOR, INRIA, B.P. 105, 78153 Le Chesnay Cedex, France, June 1999.

[2] Aline Baggio*, Design and Early Implementation of the Cadmium Mobile and Disconnectable Middleware Support*, INRIA Research Report 3515, October 1998.

[3] Aline Baggio, *System Support for Transparency and Network-aware Adaptation in Mobile Environments*, ACM Symposium on Applied Computing special track on Mobile Computing Systems and Applications, Atlanta, Georgia, USA, February 27th-March 1st 1998.

[4] Averill Law, W. David Kelton, *Simulation Modeling and Analysis (2 edition)*, McGraw-Hill Higher Education, (January 1, 1991); ISBN: 0070366985.

[5] Bill Venners, *Inside The Java Virtual Machine (Java Masters Series)*, Computing McGraw-Hill, (December 1997); ISBN: 0079132480.

[6] Bobby Vandalore, Raj Jain, Sonia Fahmy, Sudhir Dixit, "*AQuaFWiN: Adaptive QoS Framework for Multimedia in Wireless Networks and its Comparison with other QoS Frameworks*", LCN '99, October 17-20, Boston.

[7] Braam, P. J., *The Coda Distributed File System*, Linux Journal, #50, June 1998, pp 46-51.

[8] D. Wong, N. Paciorek, and D. Moore, *Java Based Mobile Agents*, Comm. the ACM, Vol. 42, No. 3, March 1999, pp. 92-102.

[9] D.B. Lange, and M. Oshima, *Seven Good Reasons for Mobile Agents,* Comm. the ACM, Vol. 42, No. 3, Mar. 1999, pp. 88-89.

[10] David Flanagan, *Java in a Nutshell: A Desktop Quick Reference (Java Series) (3rd Edition)*, O'Reilly & Associates, (November 1999); ISBN: 1565924878.

[11] Malcolm McIlhagga, Ann Light, and Ian Wakeman, *Towards a design methodology for adaptive applications*, The fourth annual ACM/IEEE international conference on Mobile computing and networking October 25 - 30, 1998, Dallas, TX USA, pp 133-144.

[12] *Mobile Information access – Coda and Odyssey* (http://www.cs.cmu.edu/afs/cs/project/coda/Web/coda.html) — Monday, April 30, 2001.

[13] *Monte Carlo Methods*, Computational Science Education Project (Sponsored by U.S. Department of Energy).
(http://csep1.phy.ornl.gov/mc/mc.html) — Monday, April 30, 2001.

[14] Noble, B, *Mobile Data Access,* doctoral thesis, School of Computer Science, Carnegie Mellon University, May 1998, CMU-CS-98-118.

[15] Noble, B., Price, M., Satyanarayanan, M, *A Programming Interface for Application-Aware Adaptation in Mobile Computing*. Proceedings of the Second USENIX Symposium on Mobile & Location-Independent Computing Apr. 1995, Ann Arbor, MI, pp 345-363.

[16] Noble, B., Satyanarayanan, M., Narayanan, D., Tilton, J.E., Flinn, J., Walker, K, *Agile Application-Aware Adaptation for Mobility*. Proceedings of the 16th ACM Symposium on Operating System Principles October 1997, St. Malo, France, pp 276-287.

[17] Satyanarayanan M, Noble B, Kumar P, Price M: *Application-Aware Adaption for Mobile Computing*. ACM - SIGOPS, Volume 29, Number 1, January 1995, pp 52-55.

[18] Satyanarayanan, M, *Fundamental Challenges in Mobile Computing*. Fifteenth ACM Symposium on Principles of Distributed Computing May 1996, Philadelphia, PA, pp 1–7.

[19] Satyanarayanan, M, *Mobile Information Access*. IEEE Personal Communications, Vol. 3, No. 1, February 1996.

[20] *The Cadmium Project*
(http://www-sor.inria.fr/projects/cadmium/index.html) — Monday, April 30, 2001.

APPENDIX A: REPORT ON THE UML DIAGRAMS

MONITORING SERVICE

Write API

This class provides the abstraction to communicate the resource counter values to the Passive Monitoring Service

Private Properties:

hashCounterObjects : Hashtable = initval

This attribute contains the map of resource counter names and their values.

Public Methods:

static GetWriteObject () : Write API

This method returns reference of an already existing WriteObject of creates one.

UpdateValue (strCounterName : String = default, CValue : long = default) : boolean

this method is used by the applications and Active Monitoring Service to set the new value of a resource.

WriteHashtable () : return

This method is periodically called by a thread to update the present hashtable contents in the Passive Monitoring Service.
This method implements the client-side protocol for Passive Monitoring Service.

RWLDAP

Public Methods:

CreateEnv () : boolean

Creates the heirarchical environment to store resource counters and their values.

DeleteEnv () : boolean

Deletes the heirarchical environment to store resource counters and their values, it also destroys all counters and their associated values.

AddAdpEntry (strResourceName : String = default, CValue : CCounterValue = default, strDescription : String = default) : boolean

This method creates a resource counter in the heirarchy and updates its value.

UpdateAdpEntry (strResourceName : String = default, CValue : CCounterValue = default, strDescription : String = default) : boolean

This method updates an existing resource counter entry with the new value.

ReadAdpValue (strResourceName : String = default, CValue : CCounterValue = default, timeMilliSeconds : long = default) : boolean

This method reads the value of resource counter; if it is more recent than the time specified, else it sets CValue to zero.

RemoveAdpEntry (strResourceName : String = default) : boolean

This method removes the specified resource counter from the Directory Service.

GetAdpEntries (strResourceName : String = default) : Vector

This method returns the counters (stored under the specified resource name) in a vector.

RWStatus

Private Properties:

ldap : RWLDAP = new RWLDAP()

This attribute holds reference to a RWLDAP object that can read / write values to the Directory Service.

Public Methods:

UpdateDirectoryService () : boolean

This method updates the counter values in the Directory Service thtough the use of ldap reference.

StatusChangeSocket

An object of this class is the Passive Monitoring Service. The constructor of this class creates a server-socket to listen to resource value updates from a client.

Public Methods:

listen () : void

This method listens to any new connections by the client, creating a new RWStatusUpdate object for every new client connection.

RWStatusUpdate

This method implements the Passive Monitoring Service Protocol.

Private Properties:

status : RWStatus = new RWStatus()

This attribute holds reference to the RWStatus object that parses the resource counters and correspondingly updates the Directory Service.

Public Methods:

start () : void

This method spawns a new thread that executes the 'run' method. A thread is created for every connected client.

Private Methods:

run () : void

This method implements the server-side of the protocol to communicate to the Passive Monitoring Service. This method calls the RWStatusUpdate:UpdateDirectoryService method.

FEEDBACK COMPONENT

IAdpStatusRegisterCallback

Public Methods:

RegisterCallback (strCounterName : String = default, interCallback : IAdpCallback = default, LowWaterMarks : long [] = default, HighWaterMarks : long [] = default) : long

To register the policy control mechanism for a resource counter. This method returns the stored value of the resource.

UnregisterCallback (interCallback : IAdpCallback = default, strCounterName : String = default) : long

To deregister the policy control mechanism. This method returns the latest value of the resource counter.

IUpdateCallback

This interface has a callback, that is used by the Passive Monitoring Service to communicate the map of resource counter names and their corresponding value.

Public Methods:

Callback (hashCounterObjects : Hashtable = default) : boolean

This method is used by the Passive Monitoring Service to communicate the resource counters and their values.

RegisterStatus

An object of this class is the Feedback Component.

Derived from IUpdateCallback, IAdpStatusRegisterCallback

Private Properties:

hashNotificationObjects : Hashtable

This attribute contains a map of all the resource names and notification objects (created one for every client interested in it).

hashCallback : Hashtable

This attribute contains a map of policy control mechanism and a vector of relevant resource changes (that is vector of CallbackCounterValue objects).

Private Methods:

PrepareHashCallback (strCounterName : String = default, lCounterValue : long = default) :

This method creates a map of policy control mechanisms and the resource changes that need to be passed to them.

IAdpCallback

This interface has a callback, that should implement the policy control mechanism.

Public Methods:

Callback (vecCounterValue : Vector = default) : boolean

This interface has a callback, that should implement the policy control mechanism. The callback values are returned as a vector of CallbackCounterValue.

CallbackCounterValue

This class is a collection of attributes that specify the name of the resource counter, the value of the resource, the water mark crossed by the resource.

Public Properties:

strCounterName : String

This attribute specifies the name of the resource.

lCounterValue : long

This attribute specifies the value of the resource.

lWaterMark : long

This attribute specifies the (lower or higher) water mark crossed.

bLowWaterMark : boolean

This attribute specifies if the lower or higher water mark was crossed. A value of true indicates lower water mark and a value of false indicates higher water mark.

CNotificationObject

Public Properties:

strCounterName : String

This attribute specifies the name of the resource.

lCounterValue : long

This attribute specifies the value of the resource.

interCallback : IAdpCallback

Contains reference to the policy control mechanism.

bLowWaterMark : boolean

This attribute specifies if the lower or higher water mark was crossed. A value of true indicates lower water mark and a value of false indicates higher water mark.

Private Properties:

presLowWaterMark : long

This attribute specifies the highest lower water mark for the present value of resource counter.

presHighWaterMark : boolean

This attribute specifies the lowest higher water mark for the present value of resource counter.

lowWaterMarkArray : long []

Contains a list of lower water marks.

HighWaterMarkArray : long []

Contains a list of higher water marks.

Public Methods:

FindChangeInWaterMark (lCounterValue : long = default) : boolean

This method returns true if the argument crossed a watermark (lower or higher). It also sets the data members of the object appropriately to denote the state with respect to the new value of the resource.

POLICY MANAGER

IPolicyManager

This interface is meant for registration and deregistration of policies. It is primarily meant for an application or policy developer.

Public Methods:

RegisterPolicies (strPolicyName : String = default) : boolean

This method allows for registration of policies.

UnregisterPolicies (strPolicyName : String = default) : boolean

This method allows for deregistration of policies.

IPolicyManagementUser

This interface is meant for requesting usage and change of policies. It is primarily meant for an application or policy control mechanism.

Public Methods:

UsePolicy (strContext : String = default, strPolicyName : String = default) :

RemoteStub

This method allows for clients to request usage of policy.
The method returns a reference to a proxy of a remote policy object.

SetPolicy (strContext : String = default, strPolicyName : String = default) : void

This method allows for applications of policy control mechanisms to request change of current policy.

IPolicy

This interface has no methods. All interfaces defining policies must inherit this method.

PolicyVector

This class implements a collection of policies. The class loads a policy-collection in a separate namespace through the use of custom classloaders in Java(tm). this class extends the vector data structure in Java(tm).

Public Methods:

CreateVector () : void

The name of the policy-collection is specified in the constructor of the class. This method loads the policies from the policy-collection into a vector using a custom class loader.

CleanUp () : void

This method unloads the custom class loader and causes the proxy object to release reference to the policies. Hence allowing them to be garbage collected.

PolicyManager

An object of this class is a Policy Manager.

Derived from IPolicyManager, IpolicyManagementUser

Private Properties:

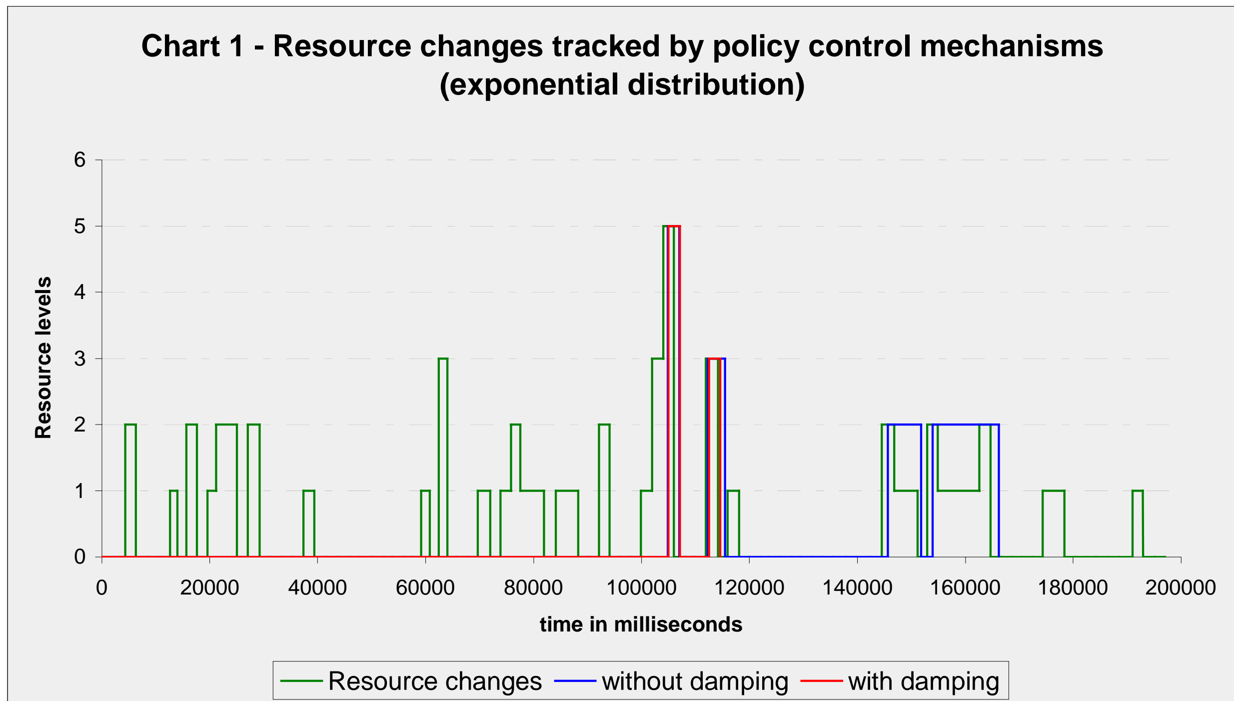
hashPolicyTable : Hashtable = null

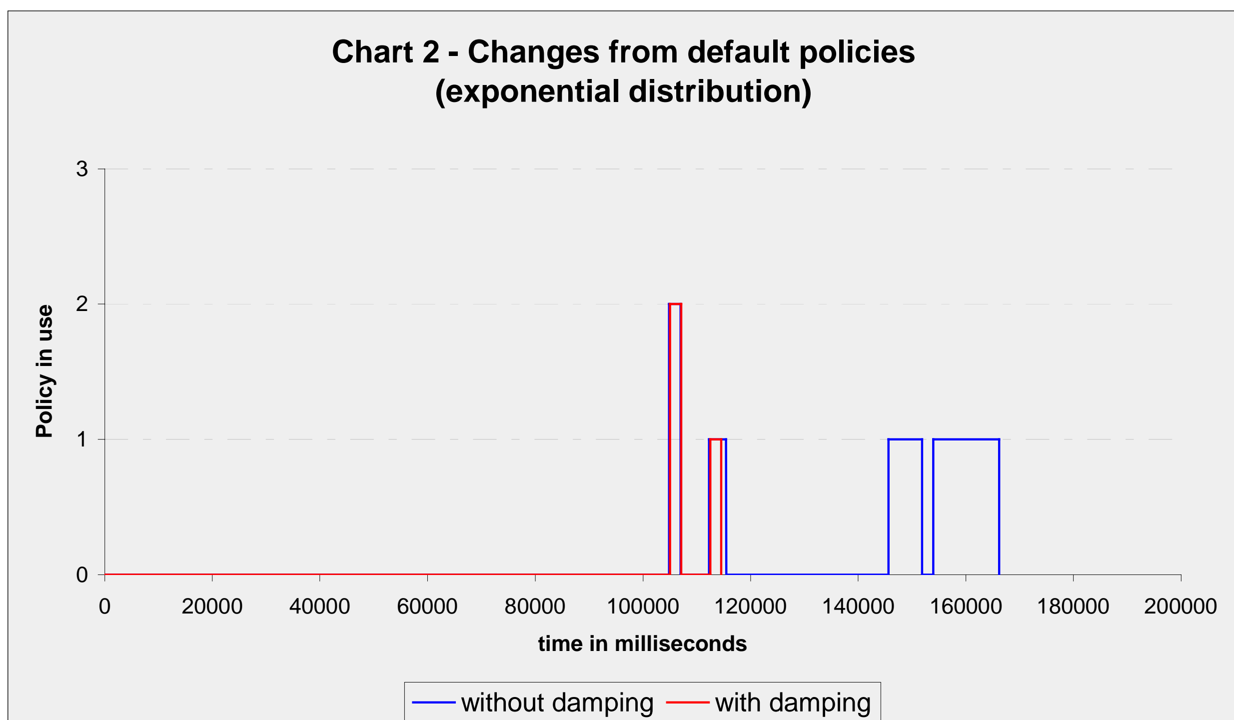
This attribute stores a map of name of policy and the PolicyVector containing the policy-collection.

hashPolicyContext : Hashtable = null

This attribute contains a map of context string and the proxy of remote policy object.

APPENDIX B: RESULTS OF PERFORMANCE EVALUATION





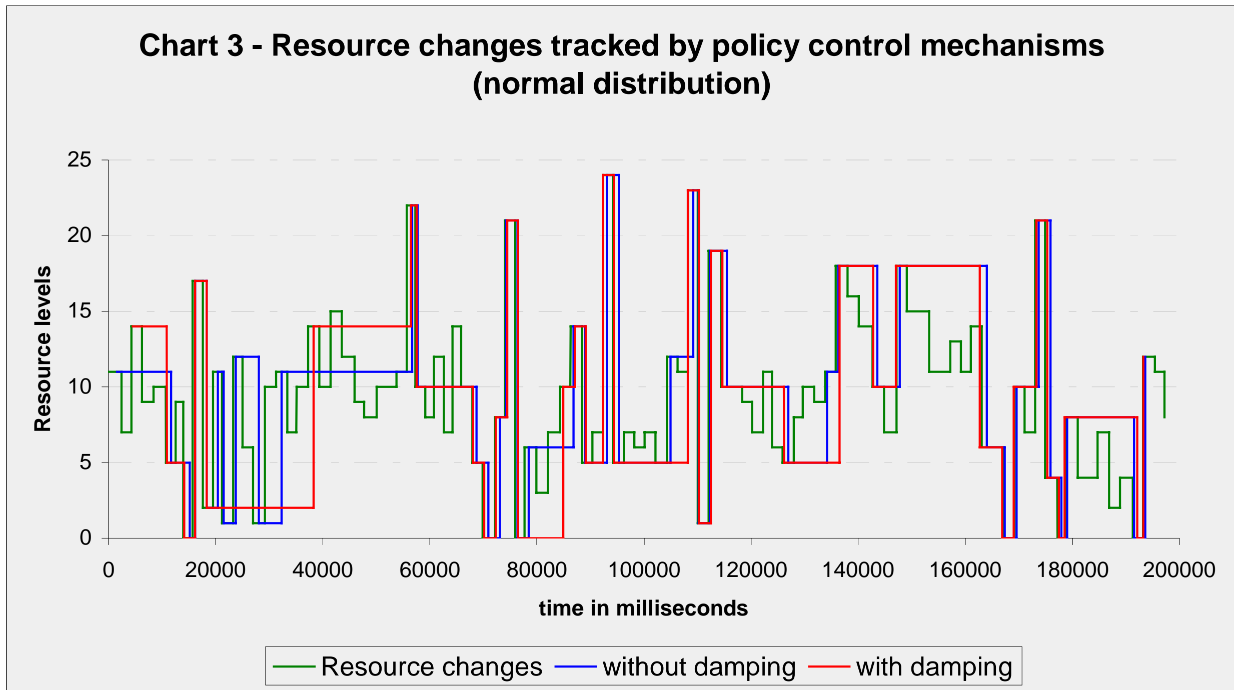


Chart 3 - Resource changes tracked by policy control mechanisms
(normal distribution)
Resource levels
0
5
10
15
20
25
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000
time in milliseconds
Resource changes
without damping
with damping

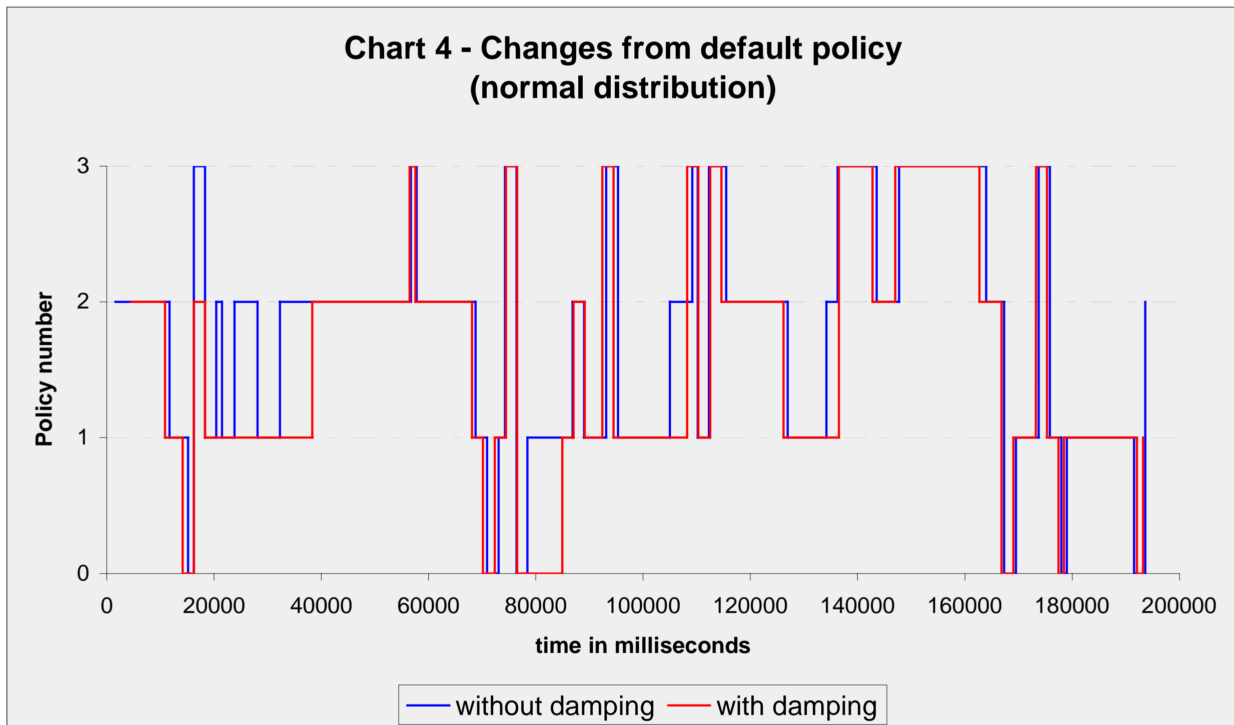


Chart 4 - Changes from default policy
(normal distribution)
Policy number
0
1
2
3
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000
time in milliseconds
without damping
with damping

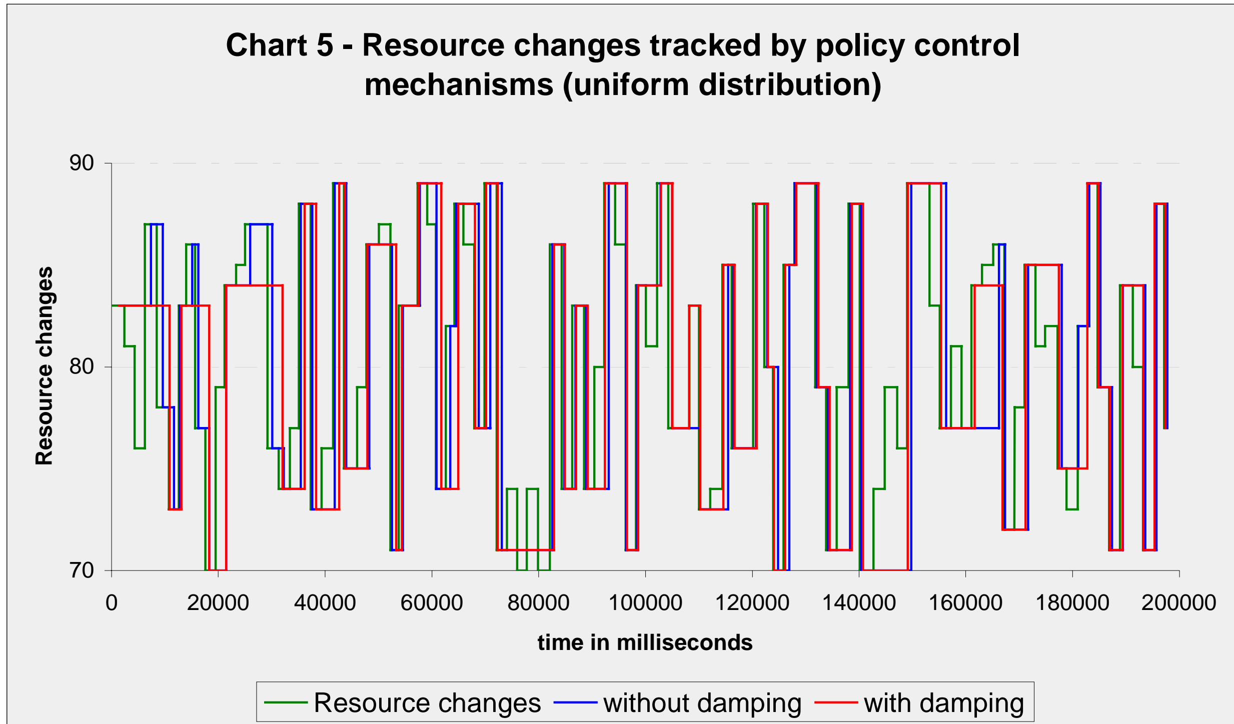


Chart 5 - Resource changes tracked by policy control mechanisms (uniform distribution)
Resource changes
90
80
70
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000
time in milliseconds
Resource changes
without damping
with damping

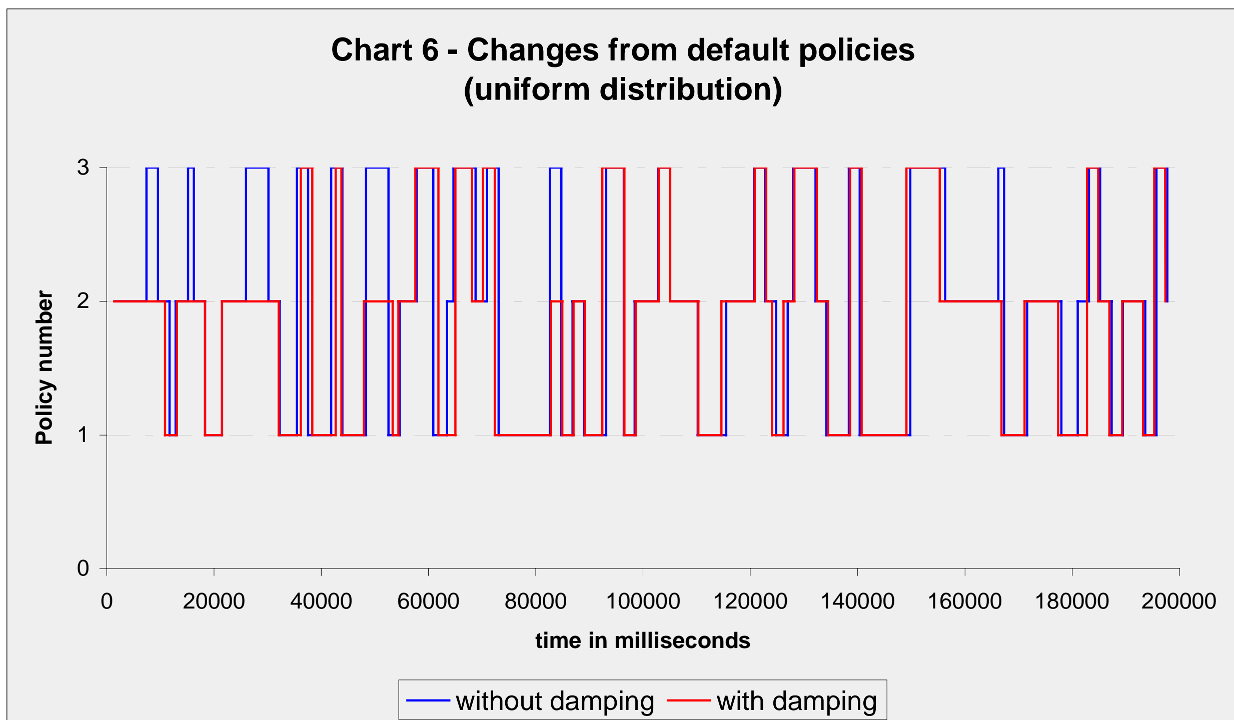


Chart 6 - Changes from default policies (uniform distribution)
Policy number
3
2
1
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000
time in milliseconds
without damping
with damping

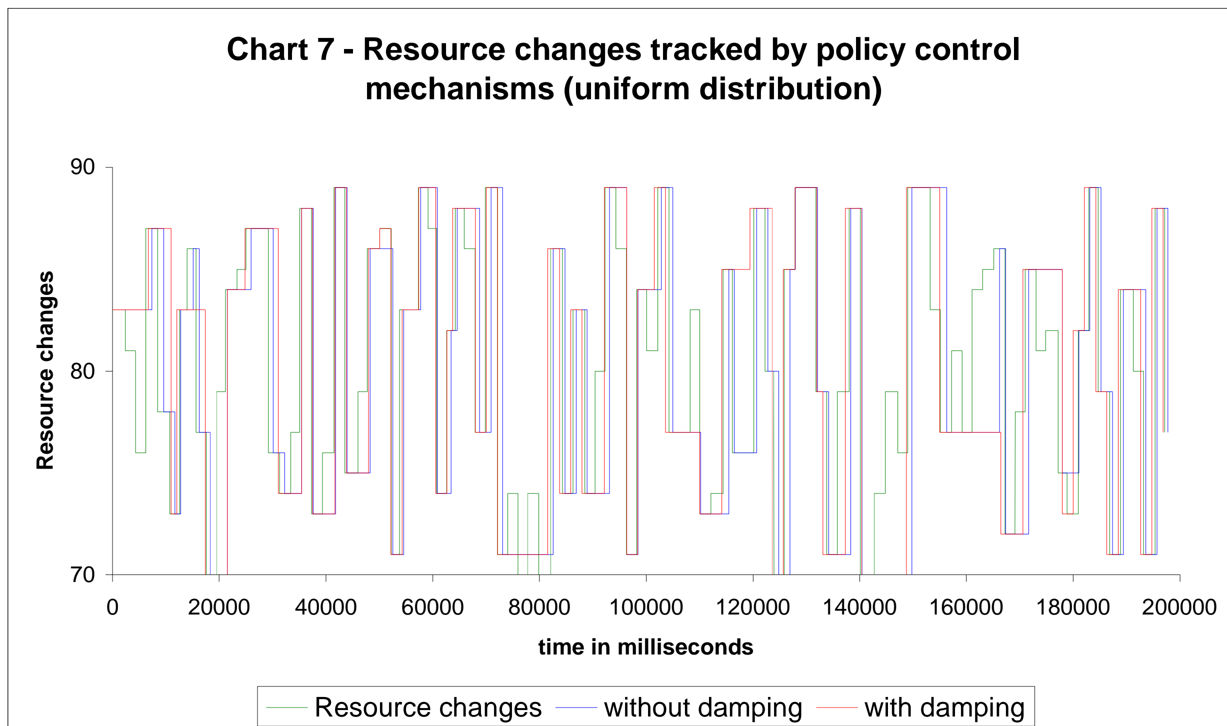


Chart 7 - Resource changes tracked by policy control mechanisms (uniform distribution)
Resource changes
90
80
70
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000
time in milliseconds
Resource changes
without damping
with damping

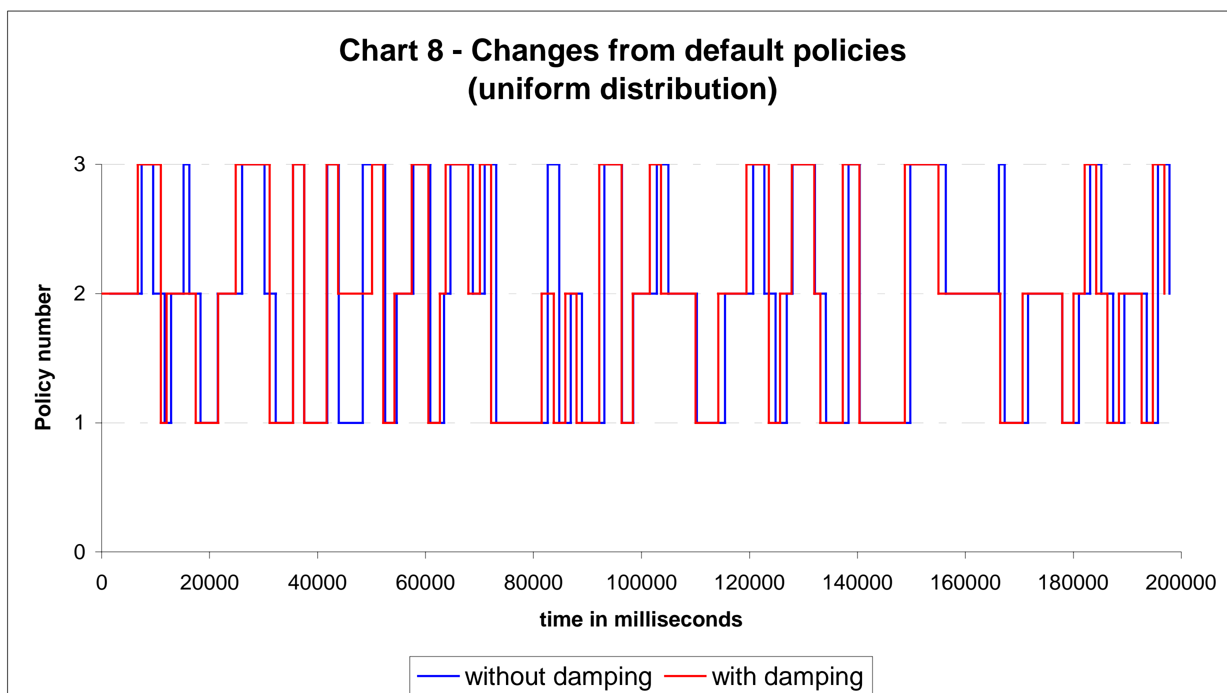


Chart 8 - Changes from default policies (uniform distribution)
Policy number
3
2
1
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000
time in milliseconds
without damping
with damping

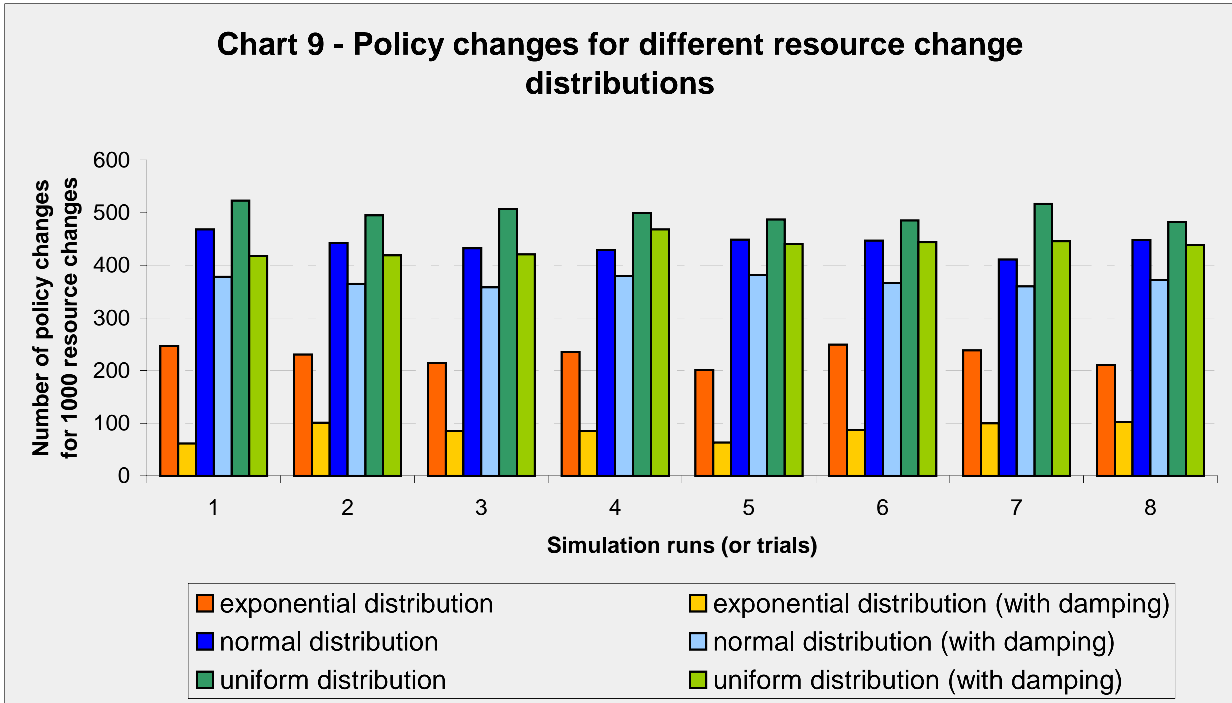


Chart 9 - Policy changes for different resource change distributions
Number of policy changes for 1000 resource changes
600
500
400
300
200
100
0
1
2
3
4
5
6
7
8
Simulation runs (or trials)
exponential distribution
exponential distribution (with damping)
normal distribution
normal distribution (with damping)
uniform distribution
uniform distribution (with damping)

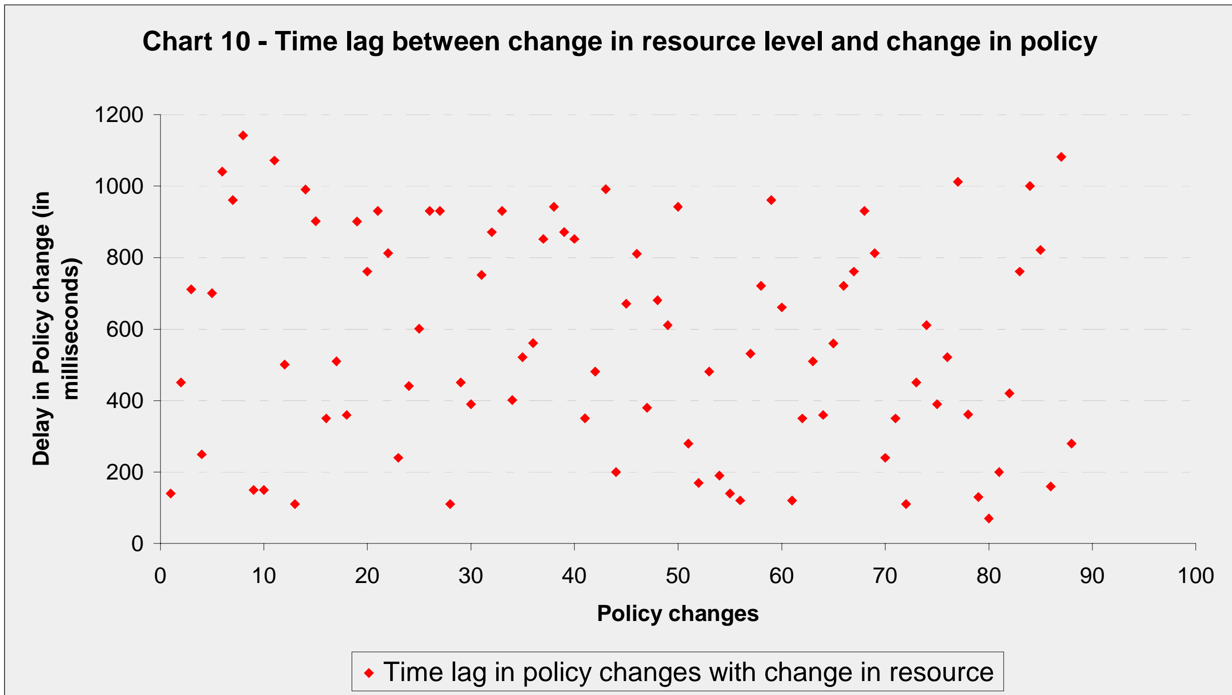


Chart 10 - Time lag between change in resource level and change in policy
Delay in Policy change (in milliseconds)
1200
1000
800
600
400
200
0
0
10
20
30
40
50
60
70
80
90
100
Policy changes
Time lag in policy changes with change in resource

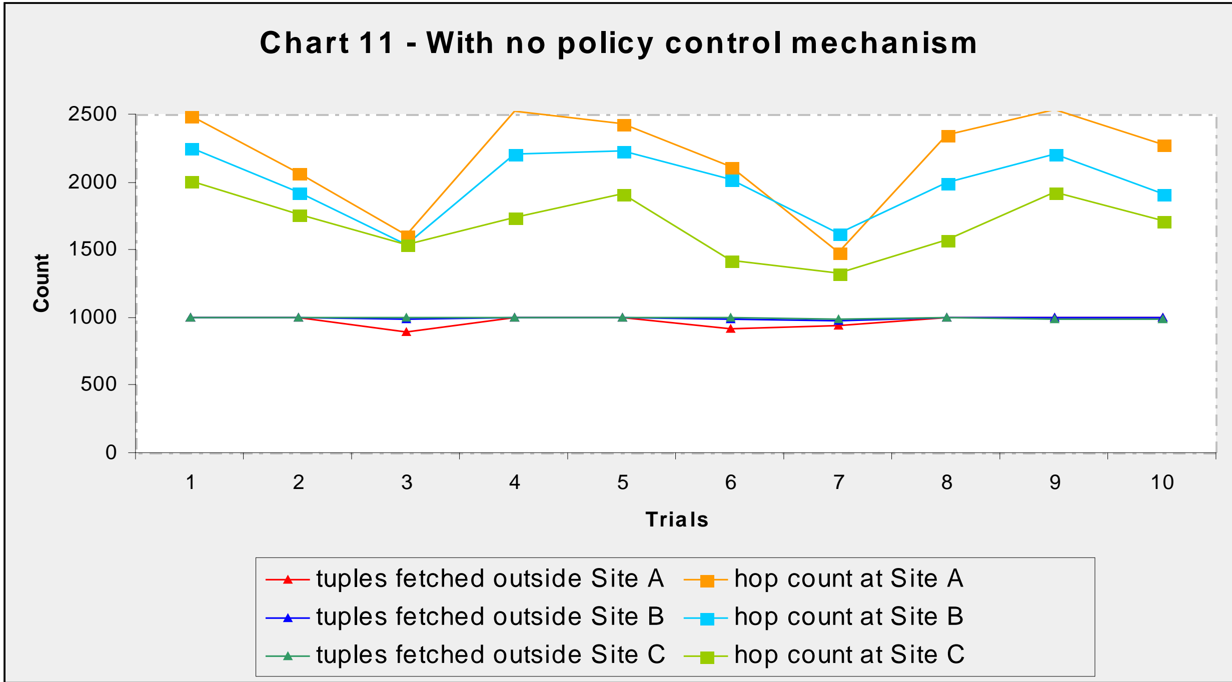


Chart 11 - With no policy control mechanism
2500
2000
1500
1000
500
0
Count
1
2
3
4
5
6
7
8
9
10
Trials
tuples fetched outside Site A
tuples fetched outside Site B
tuples fetched outside Site C
hop count at Site A
hop count at Site B
hop count at Site C

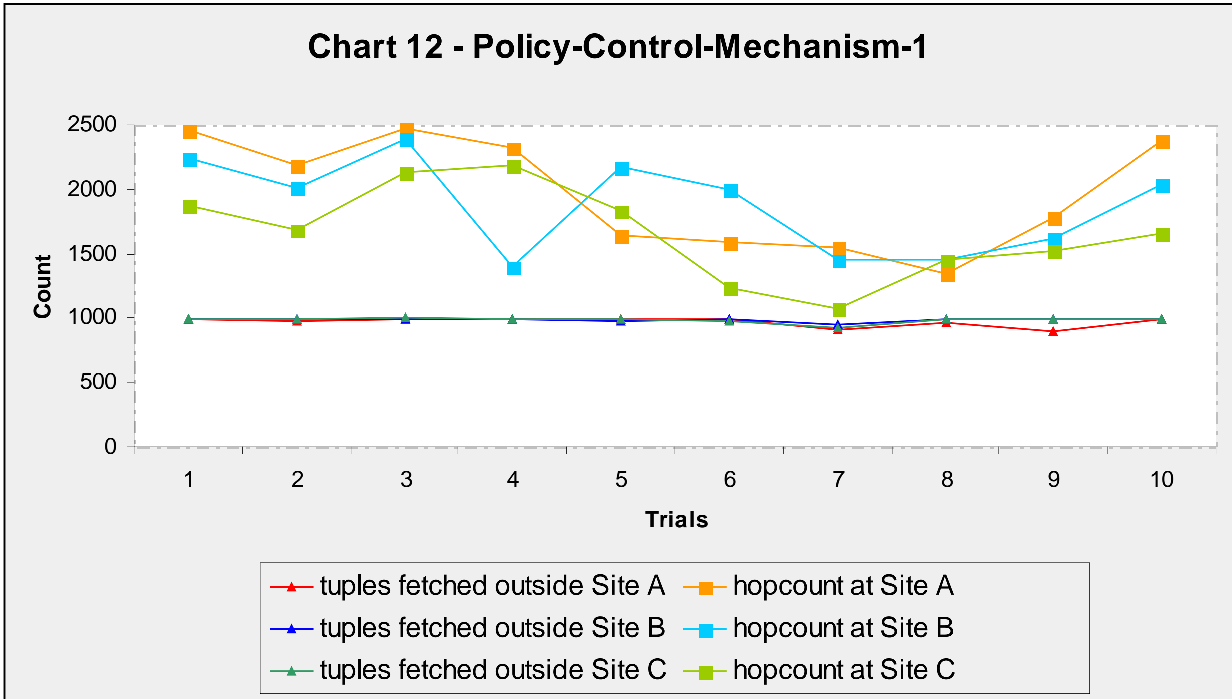


Chart 12 - Policy-Control-Mechanism-1
2500
2000
1500
1000
500
0
Count
1
2
3
4
5
6
7
8
9
10
Trials
tuples fetched outside Site A
tuples fetched outside Site B
tuples fetched outside Site C
hopcount at Site A
hopcount at Site B
hopcount at Site C

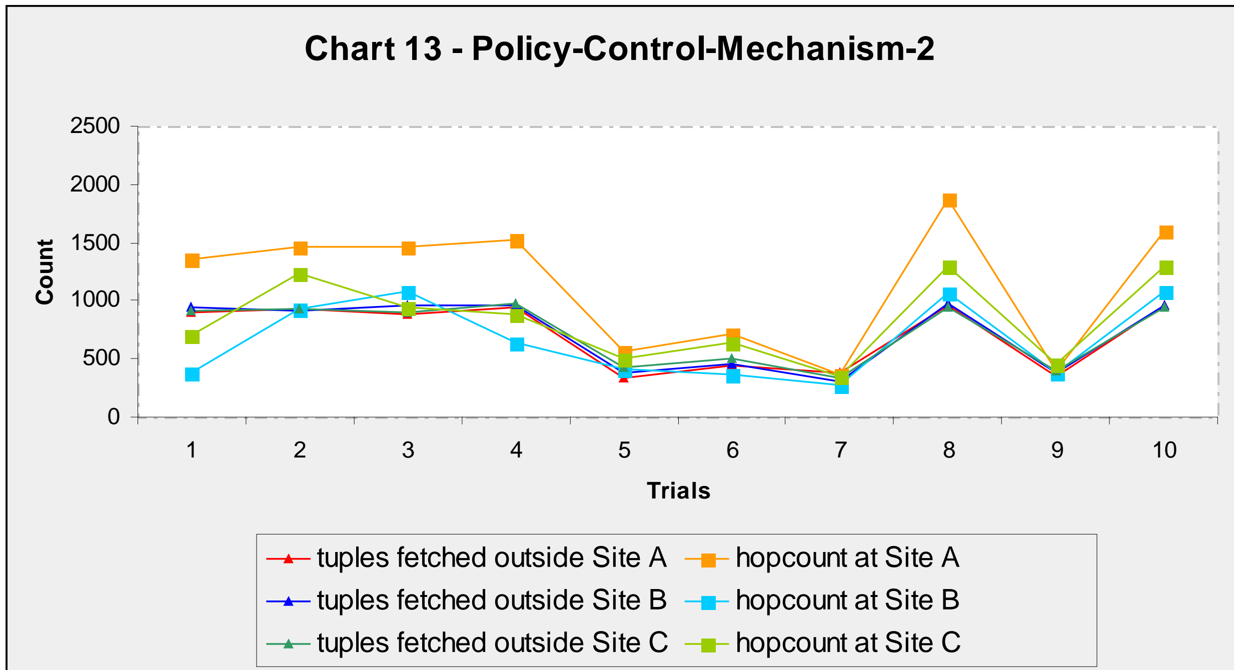
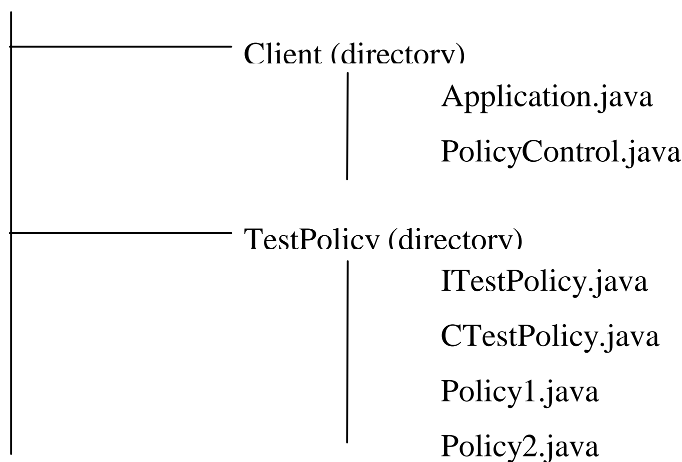


Chart 13 - Policy-Control-Mechanism-2
Count
2500
2000
1500
1000
500
0
1
2
3
4
5
6
7
8
9
10
Trials
tuples fetched outside Site A
hopcount at Site A
tuples fetched outside Site B
hopcount at Site B
tuples fetched outside Site C
hopcount at Site C

APPENDIX C: FRAMEWORK FOR AN ADAPTIVE APPLICATION

PACKAGE STRUCTURE

Note: Only implementation of Policies should be in a separate package (or directory). Application.java, PolicyControl.java and RegsiterTestPolicies.java need **not** be in the same directory. It is depicted here for the purposes of convenience only.



IPOLICY AND ITESTPOLICY INTERFACES

```
// IPolicy.java

// This interface is pre-defined.
// All policy interfaces must extend this

import java.io.*;
import java.rmi.*;

public interface IPolicy extends Remote, Serializable
{
}
```

```
// ITestPolicy.java

// Should be defined by policy (or library) developer

import java.rmi.*;

// ITestPolicy (or some other interface) is an interface for policy.
// It has no restriction on number of methods, names of methods or
// parameters and return value of the methods.

interface ITestPolicy extends IPolicy
{
  public void Policy() throws RemoteException;
}
```

CLIENT'S CODE

```
// Client.java
public class Client {
  static void main(String [] strArgs) {

    IPolicyManagementUser rmiInterPMUser;
    ITestPolicy myPolicy;

    // "client.TestPolicy" is the glue that holds the policy user,
    // policy manager and policy control mechanism together.

    String strContext = new String("client.TestPolicy");
```

```
try {

        // lookup the policy manager and request usage of policy
        rmiInterPMUser = Naming.lookup ("rmi://localhost/PolicyManager");
        myPolicy = rmiInterPMUser.UsePolicy("TestPolicy", strContext);

        while(true) {
          Thread.sleep(1000);
          myPolicy.Policy(); // No policy control mechanism in the client
        }
  }
  catch(Exception e) { e.printStackTrace(); }
 }

} // end Client
```

THE INDIRECTION - CTESTPOLICY

```
// CTestPolicy.java

// Should be implemented by policy (or library) developer.
// This class implements reference indirection.

public class CTestPolicy extends UnicastRemoteObject
      implements ITestPolicy {

  ITestPolicy fwdPolicyRef;  // forwarding reference
  int      nPolicy;

// constructor
  public CTestPolicy() throws RemoteException {      }
```

```
// SetPolicyPointer should implement support for indirection and
// also store policy the number in use
// SetPolicyPointer is not defined by any interface. Its name is fixed,
// it should accept interface to policy as first parameter, followed by
// an integer that identifies the policy (being set) for use.

  public void SetPolicyPointer(ITestPolicy policyReference,
            Integer intPolicyNumber) throws RemoteException {
    fwdPolicyRef = policyReference;
    nPolicy = intPolicyNumber.intValue();
  }

// CleanUp is needed to support hot deployment.
// CleanUp is not defined by any interface. Its signature is fixed.
// It should set current reference to null and
// call the garbage collector.
// CleanUP should also return nPolicy that identifies the current
// policy in use.

  public int CleanUp() throws RemoteException {
    fwdPtrPolicy = null;
    System.gc();
    return nPolicy;
  }

// Any call to Policy by client must be forwarded to the respective
// algorithm in use.

  public void Policy() throws RemoteException {
    fwdPtrPolicy.Policy();
  }
} // end CTestPolicy
```

ALGORITHMS FOR ADAPTATION - POLICY1, POLICY2, …

```
// Policy1.java
import java.util.*;
import java.lang.*;

// Should be implemented by policy (or library) developer
// This class implements adaptive behavior

public class Policy1 implements ITestPolicy {

// constructor
  public Policy1()       { }



// implementation of ITestPolicy. The implementation of the algorithm
// should be stateless, if hot deployment is to be supported.

  public void Policy()  {
    System.out.println("Policy 1");
  }

} // end Policy1.java



// Implementation of Policy2.java, Policy3.java and so on is as trivial
// as Policy1.java. Please note, there is no restriction on the names
// of policy classes
```

REGISTRATION OF POLICIES WITH MIDDLEWARE

```
// RegisterTestPolicies.java

import java.rmi.*;

// Should be implemented by policy (or library) developer.
// This class registers adaptation algorithms or policies.

public class RegisterTestPolicies {

  static void main(String [] strArgs) {
    IPolicyManager rmiInter;
    try {
      // lookup the policy manager and register policies
      rmiInter = Naming.lookup("rmi://localhost/PolicyManager");
      rmiInter.RegisterPolicies("TestPolicy");
    }
    catch(Exception e) { e.printStackTrace(); }
  }
} // end RegisterTestPolicies
```

REGISTRATION OF POLICY CONTROL MECHANISM

```
// PolicyControl.java

public class PolicyControl implements IAdpCallback, Serializable {

  IPolicyManagementUser rmiInter;
  String strContext;
```

```
// constructor

  PolicyControl(String context) {
    strContext = new String(context);

      // lookup PolicyManager to request SetPolicy in future.
      rmiInter = (IPolicyManagementUser)
                Naming.lookup("rmi://localhost/PolicyManager");
  }


// Callback specifies the Policy control mechanism

  public void Callback(
            VectorCallbackCounterValue vecCallbackCounterValue) {

    CallbackCounterValue tmpCallbackCounterValue;

    // we need to retrieve only the first element, as only one resource
    // was registered for this context. In case we register multiple
    // resources, then we need to loop through them to retrieve the
    // appropriate resource status.

    tmpCallbackCounterValue = (CallbackCounterValue)
                        vecCallbackCounterValue.firstElement();


    // Retrieve the water-mark crossed. In this example we are using the
    // water-mark itself to map to a policy number
    int policy = (int)tmpCallbackCounterValue.lWaterMark;
```

```
  try {
    if((tmpCallbackCounterValue.strCounterName).
                                equals("object2\\counter4")) {
      // if one of the lower water mark was crossed, then bLowWaterMark will
      // be set to true.
      if(tmpCallbackCounterValue.bLowWaterMark)
        rmiInter.SetPolicy("TestPolicy", strContext, policy);
      else
        rmiInter.SetPolicy("TestPolicy", strContext, policy+1);
    }
  }
  catch(Exception e) { e.printStackTrace(); }

} //end Callback



static void main(String [] strArgs) {
  try {
    String strContext = new String("client.TestPolicy");
    String strResource = new String("object2\\counter4");

    PolicyControl PC = new PolicyControl(strContext);

    IAdpStatusRegisterCallback registerCallback;

    // define lower and higher watermarks for the resource. Callback is
    // called (notification is received) when the resource level falls
    // below a lLowWaterMark or when the resource level rises above a
    // lHighWaterMark.

    long [] lLowWaterMark = {5, 10};
    long [] lHighWaterMark = {7, 12};
```

```
    // lookup NotificationManager, unregister any callbacks previously
    // registered w.r.t. strContext and register the new callback w.r.t.
    // strContext.

    registerCallback = (IAdpStatusRegisterCallback)
            Naming.lookup("rmi://localhost/NotificationManager");

    registerCallback.UnregisterCallback(strContext, strResource);

    registerCallback.RegisterCallback(
            PC, strContext, strResource,
            lLowWaterMark, lHighWaterMark);
   }
   catch(Exception e) { e.printStackTrace(); }
  }
} // end PolicyControl
```

Vita

Deepak Rao

Deepak Rao was born on 16th July 1976, in Bangalore, India. He completed his Bachelor of Engineering in 1998, from Sri Jayachamarajendra College of Engineering, affiliated to the University of Mysore, Mysore, India.

He was employed as a software engineer with SIEMENS Communication Software Ltd., Bangalore, India, for a period of 15 months. During this period, his responsibilities included the design, implementation, testing and documentation of telecom software.

He joined Virginia Tech in Spring 2000 to pursue his graduate studies in Computer Science. He graduated in Spring 2001. He is currently working with Hewlett Packard in their Manageability Solutions Lab.