# Quality of Service Resource Management

by

Michael J. Katchabaw

Graduate Program in Computer Science

Submitted in partial fulfilment
of the requirements for the degree of
Doctor of Philosophy

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario
June, 2002

THE UNIVERSITY OF WESTERN ONTARIO
FACULTY OF GRADUATE STUDIES

CERTIFICATE OF EXAMINATION

| Chief Advisor | Examining Board |
| --- | --- |
| ____________________ | ____________________ |
| Advisory Committee | |
| ____________________ | ____________________ |
| ____________________ | ____________________ |
| ____________________ | ____________________ |

The thesis by
Michael J. Katchabaw
entitled

QUALITY OF SERVICE RESOURCE MANAGEMENT

is accepted in partial fulfilment of the
requirements for the degree of
Doctor of Philosophy

| ____________________ | ____________________ |
| --- | --- |
| Date | Chair of Examining Board |

# Abstract

Today's computing environments are growing both in size and complexity at an astounding rate. At the same time, the applications used in these environments are becoming more numerous and more complicated and are being used in more mission critical roles in the enterprise. Consequently, users' demands for performance, reliability, and availability are increasing rapidly. To meet these needs and satisfy user requirements, a high level of quality of service must be delivered to the user. Since neither applications nor the computing environments they are deployed in are designed to do so naturally, quality of service must be managed.

At the heart of quality of service management lies resource management. If we had an infinite supply of all computing resources, we could satisfy all user requirements simultaneously; unfortunately, this is not the world we live in. Given the limited and often scarce supply of computing resources, effective resource management is therefore necessary to provide quality of service to the user. While this has been widely recognized, and work has been done to address this problem, there are many issues and problems that remain unresolved.

In this work, we present a quality of service resource management solution that overcomes the obstacles imposed by the problem. In doing so, we make several key contributions to this area of research. The approach to quality of service management taken is new and innovative; one that is dynamic, end-to-end, application-level, adjusts the computing environment to fit application needs, and is application, environment, and resource independent. A theoretical examination of the quality of service resource management problem found that most realistic resource management scenarios are NP-complete problems. With this finding in mind, several resource management algorithms were developed to improve quality of service delivery. These algorithms were integrated into a general quality of service management architecture, which was in turn developed into a working prototype management system. With this prototype system, extensive quality of service management experiments were conducted and new light was shed upon the issues surrounding quality of service resource management.

**Keywords:** Quality of service management, resource management, distributed systems management, performance management

# Acknowledgements

I would like to begin by thanking my supervisors for their invaluable encouragement, insight, and assistance in this work. While juggling his many hats, Michael Bauer was still always able to find the time for meetings and discussions whenever I needed them. Hanan Lutfiyya was also there whenever she was needed, and her time and efforts in the revision process are greatly appreciated. (Hanan is also thanked for typing her suggestions, instead of writing them out by hand, which I am certain saved countless hours of deciphering in the end!)

I am also very grateful to the members of the examining board for their recommendations and suggestions. Their feedback was quite valuable to the final revisions and editing of this work.

I would also like to acknowledge the support of the IBM Centre for Advanced Studies and thank the many people at CAS for the opportunities I was given. Discussions with other CAS fellowship students provided useful guidance and ideas in early stages of this work. I would also like to thank NSERC for its financial support over the years.

Many fellow graduate students at the University of Western Ontario deserve mention as well. Both Stephen Howard and Andrew Marshall provided valuable guidance and experience as PhD students. Andrew, in providing a LaTeX thesis template, undoubtedly saved me many hours of frustration adjusting margins, spacing, fonts, and so on. I would also like to thank Gary Molenkamp—our collaborations enriched the quality of both our works, and made my experience easier and more enjoyable.

Special mention is also deserved by David Wiseman for his insight and advice on many of the technical aspects of this work. David is also thanked for not justly punishing me for the abuse I levelled at the machines and networks he was trying to maintain and administer at the same time.

In no way can I begin to thank my family for the support I have had over the years. My wife Celia provided constant inspiration and encouragement throughout the years, and our son George provided his own special inspiration when he entered our lives not long ago.

My parents and brother also provided great support throughout the process, as did Celia's parents and her brother and sister.

Lastly, and most importantly, I would like to thank God for the opportunities and support I was given along the way. This work would not have been possible otherwise.

# Table of Contents

# List of Figures

# List of Tables

# List of Theoretical Proofs

# List of Algorithms

# Chapter 1

# Introduction

The concept of quality is innately a human notion. If a person is genuinely satisfied with something, or that thing has met expectations, then it can be said that it has quality acceptable to that person. If the person is disappointed and expectations are not met, then quality is substandard. On the other hand, if expectations are exceeded, if that is at all possible, then quality must be quite high indeed. Because quality is such a fundamental human notion, it becomes difficult to deal with quality as it applies to computing systems and the applications executing within them.

The quality of service delivered by an application can best be defined as the level of satisfaction a user derives from the use of the application—whether it meets, falls short of, or exceeds the user's expectations [201]. User satisfaction is impacted by factors that include performance, reliability, and availability. Since it is difficult to deal with this satisfaction directly, the best that can be done typically is to ensure that application behaviour meets user-provided constraints, which are referred to as quality of service requirements in this work. Meeting quality of service requirements does not necessarily guarantee user satisfaction, but should, at least, provide a reasonable approximation.

Today's typical computing environment is widely distributed, consisting of hundreds or thousands of heterogeneous computing nodes, and is growing in both size and diversity at a rapid pace. As users move to these new environments and the enabling technologies become more readily available [59], the applications being used are becoming more widely distributed, more complex and are more likely to involve the exchange of data which is time-sensitive. These types of applications, which include video-on-demand, distance education, tele-medicine, tele-conferencing, and electronic commerce, co-exist with more traditional applications that include transaction processing, data processing, and software development. This type of environment has the following characteristics:

- The quality of service requirements are considered to be soft. In other words, applications with these requirements are still considered functionally correct if the quality of service requirement is not satisfied. This is opposed to hard quality of service requirements, where applications (for example, flight control systems, chemical process control systems, patient-monitoring systems) must meet the requirement to avoid disastrous consequences.

- Many distributed applications are dynamic in nature. For example, the quality of service requirements may vary for different users (and different sessions) of the same application, for the same user of the application at different times, or even during a single session of the application. Also, during a session, the quality of service requirements may change. For example, in a multicasting application it is not unusual to have new clients join a multicasting session.

Increasingly, these environments are playing mission critical roles in the enterprise, and satisfying their quality of service requirements is essential. For example, it is estimated that more than \$4.35 billion in electronic commerce sales in the United States is lost each year due to slow page loading times (considered greater than just 8 seconds on average [215]) and other quality of service problems [214]. Indeed, users are demanding levels of service comparable to those on traditional centralized systems, where quality of service problems are smaller, better contained, more readily identified, and easier to resolve. Consequently, meeting the quality of service requirements of users is becoming a considerable challenge for today's administrators.

Given an infinite supply of all computing resources (including CPU[1] cycles, memory, disk bandwidth, network bandwidth, and so on), we would be able to satisfy the quality of service requirements of all users on all applications. Since this will never be the case, providing as many users as possible with expected quality of service entails effective management of the limited supply of resources that are actually available. In this work, this is referred to as quality of service resource management, or just quality of service management.

The problem examined in this work is to take application processes, with quality of service requirements, executing in a computing environment and ensure that these requirements are met, without undue restrictions. In this work, this problem is referred to as the quality of service resource management problem. To address this problem, we must ensure that the applications receive the resources necessary to meet their requirements. Determining the resources required in this process is therefore the crux of the problem.

[1]A list of commonly used acronyms can be found in Appendix B.

Quality of service management techniques, such as resource reservation and admission control techniques, can be used to provide guarantees of quality of service requirements, but these techniques usually base the resource reservation on worst-case needs, which leads to inefficient resource utilization [19]. In addition, these approaches often require that the developer or user have intricate knowledge of application resource needs (for example, the number of CPU cycles needed). Other quality of service management techniques focus on having applications either adapt their behaviour based on reduced resource availability (for example, changing video resolution), or make adjustments to their resource demand in the system. Generally, making adjustments to resource demand entails adjustments to the behaviour of the service consuming the resources (for example, dropped frames from a video stream will cause a video server to back off and consume fewer resources). Thus, monitored resource usage is used to determine adaptations. It should also be noted that much of the existing work focusses on approaches that manage only a single resource (for example, focus is placed on network management or CPU management to satisfy quality of service requirements, but the management of combinations of resources is not considered). In Chapter 2, we describe the shortcomings of the current research in further detail.

Quality of service management is difficult for several reasons:

- The relationship between resource allocations and quality of service delivered is unknown, or at the very least, too costly to determine [60, 207]. In general, we simply cannot determine in advance the resources required to meet a quality of service requirement for an application carrying out an arbitrary task in a computing environment without restrictions.

- The relationship between resource allocations and quality of service delivered changes over time. This is due to user requirements changing over time, as well as the resource needs of the application changing as the tasks executed by the application change [44].

- All of a computing environment's resources must be managed [33]. Otherwise, problems with unmanaged resources could adversely affect quality of service, even if most resources needed by an application are being effectively managed.

- Application users and developers cannot be burdened with making management decisions. These decisions determine when something must be done to improve quality of service, what actions are required to do so, how these actions should be carried out, and who must do so. Often, people with expert knowledge such as developers and administrators do not have the ability to make such decisions to effectively provide quality of service to application users [60, 207]. Since the majority of application users do not even have this level of expertise, they should not be responsible for such management decisions either [44].

- Providing existing applications and computing platforms quality of service resource management facilities can be difficult as they were not developed with this in mind to begin with. Since these applications and platforms will be used for years to come, this support is still necessary.

- Application and platform developers are already under severe financial and time-to-market pressures, leaving them little time to add support for quality of service resource management to their work.

In this work, we provide a solution that overcomes the above obstacles to delivering quality of service. In doing so, we make the following contributions to this field:

- We provide a new and innovative approach to quality of service resource management; one that is dynamic, end-to-end, application-level, adjusts the computing environment to fit application needs, and is application, environment, and resource independent.

- We demonstrate that most realistic quality of service resource management scenarios are NP-complete problems.

- We design several resource management algorithms with a variety of characteristics to improve quality of service delivery.

- We develop a working prototype quality of service management system incorporating these characteristics.

- We introduce techniques to allow administrative goals to guide resource management decisions. These techniques are used, for example, to provide several methods of handling overload situations, when resource demand completely outstrips supply.

- Using this prototype system and a quality of service management testbed also developed as part of this work, we conduct a variety of experiments that both demonstrate the usefulness and benefits of our approach, and shed new light on the problem of quality of service resource management in general.

The remaining chapters of this work are organized as follows. Chapter 2 presents the background behind this work, including an overview of quality of service management, and a detailed discussion and evaluation of related work in this area. Chapter 3 delves into the formalization of this problem, presenting a model and notation used in this work, and deriving several useful theoretical results on the nature of our problem and its solutions. Chapter 4 presents the architectural design of our solution to provide quality of service to application users, keeping in mind these theoretical results. Chapter 5 examines several resource management algorithms to be housed in this architectural framework, providing an analysis and discussion of each. Chapter 6 examines a prototype quality of service management system that implements our architecture and algorithms. Chapter 7 details our experience in using the prototype to date, including experiments that analyse its performance under a wide variety of operating conditions. Finally, Chapter 8 concludes this work with a summary of our results, and a discussion of directions for future work in this area.

# Chapter 2

# Background

In this chapter, we discuss the process of managing quality of service and provide background and references to related work in this area.

## 2.1 An Overview of Quality of Service Management

In this section, we examine the various aspects of quality of service management.

### 2.1.1 Approaches to Quality of Service Management

There are two main approaches to quality of service management—static and dynamic. Both approaches have their advantages and disadvantages, as discussed below.

#### 2.1.1.1 Static Quality of Service Management

Static quality of service management is static in that all management activity is completed at application start-up; after that, no changes are made to the application or environment to affect quality of service. A single resource allocation is made to each application when it is started, and this allocation stays with the application until it is terminated. If this allocation is made correctly, delivered quality of service will meet requirements throughout the life-time of the application.

The advantages of this approach are: it is relatively simple, and it does not impose run-time overhead where the application competes with its own management for computing resources. The disadvantages of this approach are that it is exceedingly difficult in general to determine the resources to meet quality of service requirements of applications in computing environments without placing undue restrictions, and that it is not likely that a single resource allocation will suffice throughout the life-time of an application. Consequently, applications often end up with too many or too few resources, resulting in resource wastage or dissatisfied users. Furthermore, this approach cannot take into account changes in user needs or environment conditions that can affect quality of service.

#### 2.1.1.2 Dynamic Quality of Service Management

Dynamic quality of service management is an on-going process; management occurs throughout the life-time of each application. As an application executes, the quality of service management system adjusts application behaviour or its environment to ensure that quality of service requirements are being satisfied. Resource allocations made to applications are subject to change as needed to maintain levels of quality of service.

This approach is advantageous as there are no resource over-allocations or under-allocations, so users are kept satisfied without wasting resources in the process. This approach can also cope with changes and take these into consideration when supporting service provisioning. The disadvantages of this approach are that it is relatively complex and it incurs run-time overhead because management requires resources of its own to maintain acceptable quality of service. This, in turn, can impact quality of service, particularly when resources are scarce.

### 2.1.2 Stages of Quality of Service Management

There are many stages to quality of service management; some are common to both static and dynamic approaches, and others are different. The dynamic approach has more stages than the static approach to management in order to support run-time adjustments. The various stages are depicted in Figure 2.1 and are discussed below.

#### 2.1.2.1 Specification

The first step to quality of service management is to specify the quality of service requirements that must be met by the application. Specifications can apply on a network-wide, system-wide, per-application, or per-user basis. These specifications can also be made at a variety of different levels, as shown in Table 2.1. Each level is described in detail below:

**User:** At the user level, quality of service specifications are some kind of expression of the user's requirements on quality of service. Such a specification would be perceptual or

**Figure 2.1** Stages of Quality of Service Management

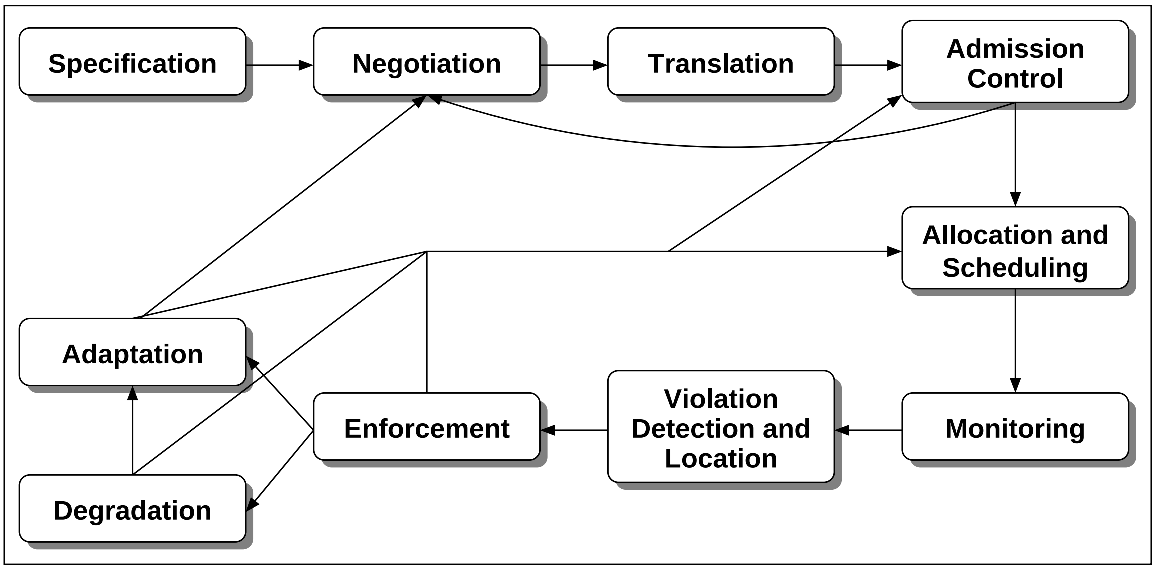



**Table 2.1** Quality of Service Specification Levels

| Level | Type of Specification |
|---|---|
| User | Expectations |
| Application | Metrics |
| Resource | Requirements |

psychological in nature, and difficult to capture accurately and formally. For example, a user may rank expected quality on a scale, with quality ratings ranging from imperceptible quality impairment at one end of the scale to very annoying quality impairment at the other end [80].

**Application:** At the application level, specifications of quality of service requirements are made in terms of application metrics. These metrics are quantifiable application characteristics—criteria to compare and measure performance or some other element of application behaviour [83]. Measurements can be extracted from the application and compared to given specifications to determine if sufficient quality is being provided to the end user. For example, a video application's quality can be measured in terms of its performance, measured in delivered video frames per second.

**Resource:** At the resource level, quality of service specifications are made in terms of resource allocations required to meet quality levels. These can be host resources such as CPU cycles, memory, and disk bandwidth, or network resources such as bandwidth, data loss, and so on. For example, an application may be given a specification allocating it 50% of a system's CPU cycles, 32 megabytes of in-core memory, 25% of the disk system's bandwidth and 5 megabits per second of network bandwidth.

Specification is the first step to both static and dynamic approaches to quality of service management, although the two approaches will usually use different specification mechanisms. Static approaches often use resource level specifications because inadequate translation mechanisms (discussed below) could lead to poor resource allocations over the life-time of an application. Resource level specifications alleviate the need for translation but places the burden on users, developers, and administrators. Dynamic quality of service approaches can make use of user or application level specifications, and make adjustments to resource allocations as necessary throughout the life-time of an application. These approaches also occasionally make use of resource level specifications.

#### 2.1.2.2 Negotiation

Quality of service negotiation is used to submit a quality of service specification from the previous stage to the quality of service management system. At the same time, negotiation ensures that specifications made in the previous stage are acceptable to the system and network and that the services exist to support the specifications. If applicable, prices and costs for the use of services and resources are also haggled over and set. As a result, negotiation is often a two-way process. It is intended that this be the only access point where the user need communicate with the quality of service management system.

At times throughout the life of an application, renegotiations are required if user needs change (and a new quality of service specification is made) or if a previously agreed upon specification can no longer be supported. For example, renegotiations would be necessary in the following scenarios:

- The user of an application decides that he or she can make do with a lower level of quality to receive a discount on service cost, and changes the quality specification accordingly.
- A failure of a network link results in a shortage of network resources, meaning that previously accepted application specifications can no longer be met.

Both static and dynamic approaches to quality of service management require negotiation to establish quality of service specifications for applications. Renegotiations are used by both approaches, but are typically rarely allowed or restricted in static approaches where run-time adjustments to the application and environment are not permitted.

#### 2.1.2.3 Translation

Quality of service translation is the process of mapping quality of service specifications into the resource allocations and parameters needed to provide the levels of service required by the specifications. This includes:

- CPU cycles
- In core memory and paging space
- Disk bandwidth and cache space
- Network mean and peak bandwidth
- Network packet loss and delay
- *etc.*

Depending on the level of quality of service specifications provided, there may be more than one mapping required. For example, user level specifications may be mapped to application level specifications and then mapped to resource level specifications, as opposed to using a direct mapping. If resource level specifications are used to begin with, little or no translation is required.

Unfortunately, quality of service translation tends to be an incredibly arduous process as there are no well-understood relationships between delivered quality of service and the wide variety of factors that can influence it. These factors include computing environment configuration (*e.g.* the types and quantities of resources available), operating system configuration (*e.g.* services installed, kernel parameters), application characteristics (*e.g.* timing constraints, processing behaviour, data consumption or creation rate), application workload (*e.g.* timing of input, nature of data input), environment workload (*e.g.* number of user processes, number of network connections), and so on.

All of the above factors affecting quality of service are inputs to the mapping process, with the result being the resources needed to satisfy quality of service requirements under those conditions. Since even application developers and administrators do not have the knowledge or experience to do this mapping, developing the mapping tables or functions to support this would require an exceedingly large number of experiments to be run on every application under every possible configuration, which is practically impossible. Even if there was sufficient time to carry out these experiments, the resulting mapping tables or functions would be so incredibly large and complex, that they would impose huge overhead and would be rendered practically useless. As a result, general purpose quality of service translation algorithms do not exist. This means that any translation only works well in a very restricted set of circumstances, producing inaccurate results in others.

Nevertheless, quality of service translation is incredibly important to static approaches to quality of service management. If specifications are mapped incorrectly to poor resource allocations, applications are locked into either wasting resources (if they are allocated too many resources), or falling below requirements (if they are not allocated enough). Since these allocations last the life-time of the application, there is little room for error in static approaches. Static approaches to quality of service management would favour a long and complex, yet accurate, translation process to ensure good results. Since these mapping algorithms do not generally exist, most static approaches use resource level specifications and bypass the translation process entirely. Unfortunately, this shifts the burden from the developers of the quality of service management solution to the shoulders of application users, developers, and administrators who are, in reality, no better suited to do the job.

Dynamic approaches to quality of service management, on the other hand, are more flexible when it comes to quality of service translation. A good initial translation of the quality of service specification may result in fewer resource adjustments through the life of an application; a poor translation can still be acceptable, although requiring more adjustments. As a result, dynamic quality of service approaches can also live with simpler, less accurate translation mechanisms than static approaches. It is also possible for dynamic quality of service approaches to use no translation mechanism at all, relying solely on resource adjustments to meet quality specifications at any level (unlike static approaches, which would require them at the resource level in such cases). In such cases, one would view the translation and resource allocation stages of quality of service management as being very tightly coupled.

#### 2.1.2.4 Admission Control

The quality of service admission control process is used to prevent the computing environment from being saturated with applications and unable to meet the quality of service specifications for any of them. In this process, translated application resource needs are compared to available resources in the environment:

- If sufficient resources are present in the environment, the application is admitted and allowed to execute. "Sufficient" resources could mean exactly the amount of resources remaining in the environment, fewer resources than what actually remains (allowing for additional resources to be made available in special cases or emergencies), or more resources than what actually remains (allowing for resources to be over-booked). The exact meaning of “sufficient” is up to administrators to define.

- If sufficient resources are not available, the application is rejected. At this point, renegotiation is necessary before the application can be admitted with specific quality of service requirements. Alternatively, the application can still enter the environment and receive best-effort service.

Static quality of service approaches often use some form of admission control to protect the integrity of their resource allocations. Otherwise, too many applications could enter the environment and impact the quality of service of previously admitted applications—if the approach is static, it could not make adjustments to repair the problem after it has occurred. Dynamic quality of service approaches use admission control as well to prevent resource saturation, although it is not quite as integral as in static approaches. In some dynamic approaches, admission control is not used at all, such as when no translation is used and resource allocations are set entirely by adjustments (in such cases, there are no initial allocations upon which admission checks can be made).

#### 2.1.2.5 Allocation and Scheduling

The allocation and scheduling stage is central to the management of quality of service. In this stage, resources are allocated to applications to meet the level of quality of service required by their specifications. Allocations can be made using a variety of different algorithms, depending on the resources in question.

In static quality of service management approaches, this is the final stage of management. After resources are allocated to the application, the application executes on its own, presumably meeting its quality of service specifications. In dynamic approaches to quality of service management, this stage is visited repeatedly as adjustments are made to resource allocations to bring quality of service into line with requirements. In these cases, resource allocations and adjustments must be made quickly, with as little overhead as possible.

#### 2.1.2.6 Monitoring

Monitoring is used by dynamic quality of service management approaches to collect data on executing applications and their operating environments for quality of service management purposes. The more information that is available to the management process, the better the management decisions that can be made. Coupled with these better decisions, unfortunately, is a higher cost in resource usage to collect, store, and process this data.

Monitoring can by request-driven, in which a component is queried explicitly to monitor it, or event-driven, in which a component asynchronously reports its status at regular intervals, or when certain conditions have been met. For example, the frame rate of a video application can by determined by asking the application for it, or by having the application report its frame rate after every ten frames of video. Monitoring can also be direct, by observing the component of interest, or indirect, by observing the effects the component has on other components. For example, the resource usage of an application can be determined directly by asking the application, or indirectly by asking the operating system on the host on which the application is executing. The type of monitoring used depends on the components being monitoring, the resources available to the monitoring process, and the information required for monitoring.

This monitoring includes tracking the following:

- The level of quality of service being delivered by applications to their users.
- The resource consumption of applications.
- The behaviour of users and their applications.
- The behaviour of systems and network components in the operating environment.

This information is used by later stages of quality of service management to determine if quality of service specifications are being met and to help determine what to do if they are not. This information is also used to optimize resource allocations and consumption, and to tailor the configuration of applications and environment components where appropriate.

#### 2.1.2.7 Violation Detection and Location

A violation occurs whenever an executing application does not satisfy its quality of service requirements or whenever a user or application abuses negotiated service terms. Since these indicate a problem for either the application or its operating environment, steps need to be taken to solve the problem.

To be able to deal with these situations, they must first be detected. This can be done by analysing data collected through monitoring. Detecting violations of quality of service requirements is usually a simple matter of comparing the quality being delivered by the application to its quality of service specification, if this specification is an application level specification. If its original specification was user level, then delivered quality is compared to the application level translation of its specification. If its original specification was at the resource level, then its resource consumption is checked against its specification instead. Mismatches in any of these comparisons result in a detected quality of service violation. To detect violations from abuses of service terms, more complex behavioural analysis is required over time.

Once violations are detected, the sources of the problems must be identified so that appropriate corrective actions can be developed. For example, if a distributed client-server application performs below requirements, the cause of this problem could be at the client or server systems, or in the network. One, or possibly more, of these components need to allocate additional resources to the application to increase its performance. Which component(s) and which resource(s) need to be determined so that the problem can be effectively solved as quickly as possible without wasting resources needlessly. To do this location, data collected through the monitoring stage must be analysed and correlated accordingly.

#### 2.1.2.8 Enforcement

Once a violation has been detected, and the source of the problem has been located, quality of service enforcement determines the sequence of actions necessary to correct the problem. The sequence needed may be simple, involving a single action, or very complex, involving several actions taken over time.

Corrective actions can include the following:

- Changing resource allocations to the application encountering the violations of quality of service requirements.
- Changing resource allocations to other applications to free up resources for the application with a violation of quality of service requirements.
- Changing application behaviour.
- Renegotiating service terms.

The corrective actions taken depend on environment conditions, the application, its user, and the nature of the problem that caused the violation in the first place. Once these corrective actions have been applied, the quality of service of the application will move into line with requirements. If not, further data collection, analysis, and actions are required by repeating the monitoring, violation detection and location, and enforcement stages of management.

Unfortunately, it might not be possible for management to allocate resources so that quality of service requirements may be satisfied. This is particularly the case when little or no admission control is done, and too many applications have saturated the environment, and there are simply not enough resources to meet demand. These situations are discussed below.

#### 2.1.2.9 Adaptation

In cases when there are not enough resources to meet an application's quality of service requirements, the application needs to somehow deal with the situation. This requires some kind of graceful adaptation on the part of the application.

Possible adaptations can include the following:

- Accept fewer resources and deliver quality of service below original requirements.
- Accept fewer resources and make the best of the situation by changing its behaviour to compensate.
- Attempt to renegotiate service to achieve the quality that is wanted.
- Cancel its quality specification and proceed with best effort service.
- Terminate and attempt execution later when the desired level of quality can be provided.

Naturally, if an application is forced to adapt, it will likely receive a discounted cost for its service provisioning, if one was incurred in the first place.

The adaptation taken depends on the application, its user's needs, and the exact situation. Some applications may retain most of their value even if they cannot satisfy their quality of service requirements; others may have little or no value if requirements cannot be met. The decision on how to adapt can be made directly by the application, its user, or the quality of service management solution in place, on behalf of the application.

#### 2.1.2.10 Degradation

In this stage, there are no longer any available network or system resources for allocation, and there are constant violations of quality of service requirements from all applications—the demand for resources has outstripped supply. Furthermore, the remaining adaptations or renegotiations that can be made produce other undesirable results.

In this stage, the options are limited and will likely produce a great deal of short term discomfort, hopefully to produce a better situation in the future. These options include:

- Instituting admission control immediately to prevent additional applications from entering the environment.
- Cancelling quality of service specifications for some or all applications. This not only frees up resources from those applications to allow others to complete, but also reduces management workload from monitoring those applications, freeing additional resources.
- Suspending or even terminating executing applications, selected according to their relative priority. This priority is determined by a number of things including the type of application, its value, the user executing the application, and a variety of other factors.
- Further adjusting the quality of service management process to reduce overhead and make resources available to executing applications. For example, this can be done by reducing the amount of data collected during monitoring; fewer resources would be required to both collect and process this data, making these resources available to applications.

The best solution to this problem would both get the environment out of a degraded mode of operation as soon as possible, and also prevent the environment from falling into this situation again for a long time. Such a solution would depend on the environment, the application, and the sequence of steps and problems that resulted in the degradation.

### 2.1.3 Levels of Quality of Service Management

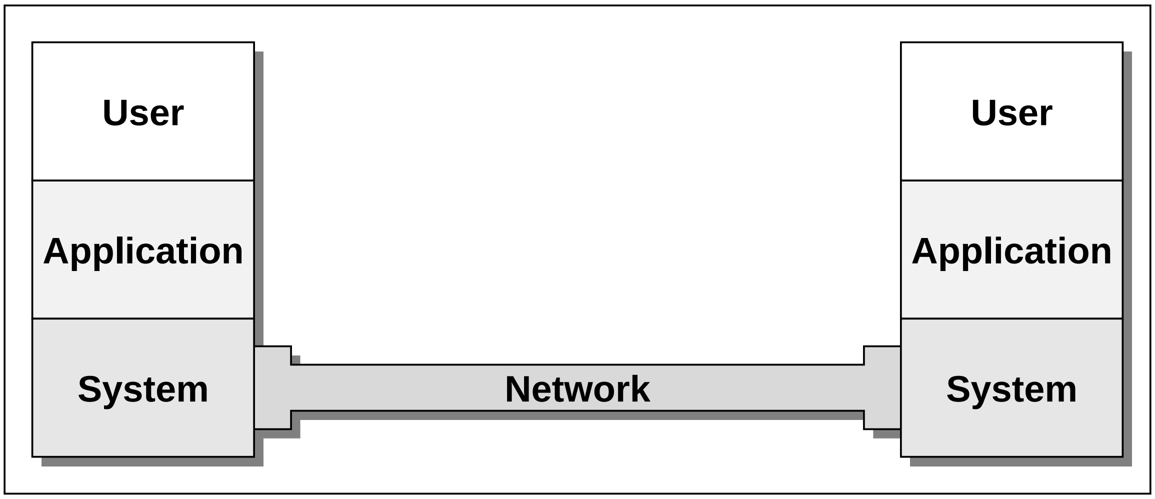
The stages of quality of service management presented above can focus at one of several levels, as shown in Figure 2.2. In this section, we describe the various levels.

#### 2.1.3.1 Network Level Quality of Service Management

At this level, quality of service management is aimed at providing sufficient network resources and data traffic characteristics to meet the needs of applications communicating across the involved network links and devices. This includes things such as bandwidth, delay, jitter (variation in delay), and data loss. When research into quality of service management began, this was the focal point of research, given that networks were typically the bottlenecks and the cause of quality problems in distributed applications.

Unfortunately, focussing on this level alone misses a considerable portion of the potential quality of service problems that can occur. Not all difficulties in providing quality of service

**Figure 2.2** Levels of Quality of Service Management




are confined to the network, especially when one considers the improvements in network speed and reliability, and the maturation of network research. The end systems that lie on the edge of networks can also be the source of quality problems, with lack of resources and performance bottlenecks, some of which shifted there as network bandwidth improved. Without considering these end systems in quality of service management, it is incredibly difficult to provide quality assurances to users, let alone guarantees.

The quality of service specifications used here are almost always at the resource level.

#### 2.1.3.2 System Level Quality of Service Management

At this level, quality of service management is directed at providing sufficient levels of all resources—both end system resources (such as CPU cycles, in-core memory, disk bandwidth, and so on) and network resources. The goal is to provide an "end-to-end" solution where all resources that a given application needs on all hosts and network devices are managed to provide acceptable quality of service.

In recent years, research in this area has started to appear, and is beginning to mature. Much of this work has been adapted from more traditional work in real-time systems, processor scheduling, memory management, and disk and I/O optimization.

Like network quality of service management, the quality of service specifications used here are almost always at the resource level.

#### 2.1.3.3 Application Level Quality of Service Management

At this level, quality of service management is still directed at providing an end-to-end quality solution, as in system level quality of service management. The difference is that in application level management, quality of service specifications are application level specifications. The management system effectively hides the translation process from those making quality specifications.

Work in this area is still in relative infancy, though promising progress is now being made.

#### 2.1.3.4 User Level Quality of Service Management

At this level, quality of service management is still directed at providing an end-to-end quality solution, as in the previous two levels. The different is that in user level management, quality of service specifications are user level specifications—user perception and behaviour are incorporated into the management process. The management system effectively hides the translation process from those making quality specifications.

Work in this area is the least mature of all of the levels, though, once again, promising steps forward are starting to appear. Currently, this work is more often done by psychologists and perception or behavioural specialists than computer scientists, although researchers in human-computer interaction are beginning to bridge the gap.

## 2.2 Related Work

The leading solution to quality of service problems today is to simply add additional resources to systems in an attempt to make the problems go away. This includes faster processors, more memory, higher bandwidth network connections, and so on. Unfortunately, this tends to only delay the inevitable resource shortage, and can be quite costly over the long term. Furthermore, there is usually downtime and service disruption while these upgrades are made. Without proper quality of service management, studies have shown that approximately 20% of resource capacity is wasted, and fails to satisfy quality of service requirements in 90% of cases [51].

Because of recent economic hardships in the world of information technology, however, it would appear that simply upgrading or adding computing resources as a solution to quality of service problems may cease being such an attractive solution due to decreasing or more constrained budgets. Indeed, people are now being urged to do more with the resources they have, or perhaps with even fewer resources [64]. Consequently, to fully address quality of service problems, research into quality of service management is needed. In this section, we organize and examine some of the research work to date in this area and assess its effectiveness and suitability to task.

### 2.2.1 Resource Management Algorithms and Policies

Over the years, a number of approaches for managing computing resources have been developed, varying in complexity, flexibility, robustness, efficiency, and overhead. In this section, we list many of the more classic and commonly-referenced algorithms that have formed the basis for further research into quality of service resource management. Following sections will delve more deeply into the modern approaches and algorithms.

For the CPU, classic algorithms include first-come first-serve, shortest job first, round robin, and priority queues, as discussed in [175], as well as additional real-time algorithms such as rate monotonic [116], earliest deadline first [35], proportional share [179] and others. For memory, algorithms include first-in-first-out selection [55], least recently used selection [36], the working set approach [42], clock selection [48], WSClock [26], and several others, including newer approaches such as the access-graph algorithm [52]. For disk access, algorithms include first-come first-serve, shortest-seek-time-first, SCAN, priority SCAN, C-SCAN, LOOK, C-LOOK, and others, as discussed in [175, 208], and newer algorithms such as YFQ [23]. For network resources, algorithms include first-come first-serve, fair queueing, weighted fair queueing, virtual clock, earliest deadline first, delay earliest-due-date, jitter earliest-due-date, stop-and-go, hierarchical round-robin, leave-in-time, RCSP, PGPS, and others, as discussed in [210], as well as newer algorithms designed for quality of service resource management, such as SCED [168] and REQS [159].

All of the above algorithms have their own strengths and weaknesses, and are suited to some tasks more than others. Combining these approaches, or supporting several of them at once, would provide very powerful resource management capabilities to support quality of service.

It is important to note, however, that algorithmic solutions on their own are not enough for quality of service management. Users and developers should not interface with, or be exposed to, these algorithms and approaches directly. Instead, these algorithms used at the core of quality of service solutions for support, with other mechanisms in place to work with applications. Otherwise, as mentioned earlier, users and developers may be overwhelmed by what they have to work with. Many researchers in the area have recognized this, with many new algorithmic approaches developed in the context of broader architectures and solutions, as will be seen in the sections below.

### 2.2.2 Commercial Systems

Resource management in most commercial operating systems is based heavily on the algorithms and policies listed above. Some operating systems, however, have provided additional support to improve resource management capabilities.

#### 2.2.2.1 Solaris

Solaris provides several mechanisms that give additional support for resource management [125]. To assist in CPU scheduling, Solaris provides several scheduling classes, as well as the ability to change the behaviour of each class. One of these classes supports real-time scheduling [182], which can be used for extending additional scheduling mechanisms, as was done in [33, 87]. To provide additional memory management support to programmers to reduce page faulting, Solaris also provides a memory advisement facility [183]. Programmers provide intended memory access patterns for address ranges to the kernel, which in turn attempts to optimize paging according to this advice. Both of these mechanisms, however, tend to require specific a-priori knowledge, and could potentially cause detrimental system behaviour if used improperly or incorrectly.

Newer versions of Solaris also support management of network resources through the Solaris Bandwidth Manager [186]. This manager allows bandwidth to be allocated to classes of traffic, traffic marking and prioritization, and network optimization to make complete use of available bandwidth. To do this, the Solaris Bandwidth Manager supports RSVP, differentiated services, and other mechanisms. Unfortunately, only resource level specifications are supported, and making these specifications can be cumbersome. Currently, only Java applications can dynamically configure quality of service specifications through Java beans, unless a directory service is used to store this information [185]. In this case, applications written in other languages such as C can indirectly manipulate these configurations through a directory service API. The reason for these restrictions is that the configuration of the Solaris Bandwidth Manager is assumed to be static and typically set by administrators either using configuration files or a supplied graphical administration tool. This is intentional, to keep resource management decisions in the hands of administrators, rather than users and developers, to retain control over the computing environment. While this might be a safe practice, it is somewhat restrictive and presents a barrier to automated or dynamic quality of service solutions. Third party support for network resource management through RSVP and differentiated services is starting to mature as well, however, possibly filling in the gaps.

#### 2.2.2.2 Windows 9x/NT/2000/ME/XP

Windows NT [162] provides multiple scheduling classes and real-time scheduling support that can be extended much in the same way as Solaris [115]. As was the case with Solaris, however, this mechanism must be used very carefully by users and developers or else system performance and operation can be disrupted severely. Windows NT is also interesting as it can capture some user activity to change scheduling and resource allocation, such as adjusting application priorities and time quanta slightly as applications are foregrounded and backgrounded. Memory management in Windows NT attempts to approximate the working set approach, without many of the overhead costs associated with this approach [95]. Programs start with a default working set size, which grows as memory demand increases. Periodically, the working set manager steals pages from each process. If a page fault occurs on a stolen page in short order, it is returned to the process. Otherwise, the working set size for the process is reduced and the page is freed for allocation elsewhere. While this approach is relatively efficient, it does induce unnecessary page faults, and is not necessarily as optimal as the traditional working set approach, as unneeded pages may be retained longer than necessary before they are stolen.

Windows 2000 is the successor of the Windows NT operating system, and as such, contains the resource management features (and drawbacks) of Windows NT listed above. Windows 2000 additionally supports the Generic Quality of Service (GQoS) API for developing quality of service enabled applications [131] (Windows 98 supports a portion of GQoS; presumably Windows Millennium Edition has at least the same support). The purpose of this API is to allow applications to easily invoke network quality of service functionality through RSVP, differentiated services, and potentially other mechanisms, such as ATM signalling. While this API provides access to these mechanisms, it requires resource level specifications, requiring users or application developers to know the resource needs of their applications, which tends to be problematic. Resource capacity checking and admission control is also minimal in this approach, allowing users to quickly soak up all available network bandwidth, which can seriously impact the performance of other applications. Furthermore, this API only supports network resources, and does not provide a unified mechanism for managing all of an application's resources—the other mechanisms described above must be used instead.

Windows XP is the latest incarnation of the Microsoft Windows product line. Currently, it is still difficult to find much on the inner workings of Windows XP, in both its home and professional editions, since they were released only in late 2001. While it is generally believed that Windows XP is largely a re-release of Windows 2000, with a cosmetic make-over and additional features bundled in, it will be interesting to see what quality of service features are supported. One would imagine that it retains most of the features of its predecessors, with possibly support for new technology as well.

It will also be interesting to see how research from the Millennium project at Microsoft Research [20] is integrated into the Windows product line. In this project, Microsoft Research is investigating new directions in operating system design and implementation. Among the various elements being examined are introspection and self-tuning to improve resource usage and application behaviour. However, it is too early to determine how and when these features will be made available for general use.

#### 2.2.2.3 Linux

A real-time version of the Linux operating system is available [11]. This provides very interesting scheduling capabilities, with similar benefits and potential hazards as Solaris and Windows NT. Furthermore, as reported in [49], many Linux variants adopt a page stealing approach in virtual memory management resembling the approach taken in Windows NT, again with similar benefits and problems. Linux also supports network resource management through RSVP, differentiated services, and other mechanisms, such as ATM signalling. Once again, this is done through resource level specifications. Clean APIs to simplify network resource management are currently under development.

Since Linux is completely open source, and has large community support, it is easier to develop quality of service extensions to the system. Solaris is becoming more open, with several academic and industrial options available to get access to operating system source code. The Windows product line, on the other hand, is generally closed, so it is more difficult for the average researcher to work on quality of service extensions to these operating systems.

#### 2.2.2.4 FreeBSD/OpenBSD

Quality of service development on the FreeBSD [192] and OpenBSD [82] operating systems is somewhat slower than their open source cousin, Linux. A real time POSIX feature set is currently being added to OpenBSD, to provide better CPU management. Both use variations of the LRU paging algorithm, and support additional memory management features such as the memory advisement feature also available in Solaris. For network resource management support, FreeBSD has RSVP support, while differentiated services support is very sparse.

FreeBSD and OpenBSD are open source variations of the 4.4 BSD Unix operating system. As such, developing quality of service extensions to the operating system is easier than

Solaris and Windows operating systems. Unfortunately, FreeBSD and OpenBSD do not have the same developer base or popularity as Linux does, so quality of service additions are slower in coming and not as mature.

### 2.2.3 Research Approaches and Systems

Contributing to the above commercial work, there is a great deal of non-commercial research in the area of quality of service management, including a wide variety of approaches, frameworks, architectures, and systems. In this section, we present some of this work and discuss its suitability to task.

#### 2.2.3.1 Architectures and Frameworks

There are several different approaches to quality of service management. This has led to the development of several different architectures and frameworks by a variety of researchers in the field.

Among the different approaches, there are a few "classic" approaches that have been around for several years. These include the Heidelberg QoS Model [41], the Extended Integrated Reference Model (XRM) [105], the OMEGA Architecture [141] which led the QoS Broker model [140], the Quality of Service Architecture (QoS-A) [25], the OSI QoS Framework [79], the Tenet Architecture [54], and so on, as discussed in [7]. Most of these approaches are rooted in network quality of service, but many could be applied to end-to-end quality of service management. Most of these approaches are relatively static, although some support a form of renegotiation of their quality of service terms, which are primarily resource-level quality of service specifications. It is interesting to note that while these approaches laid much of the foundation for quality of service management research, and are commonly cited in the literature, they are no longer used in practice. Many of the concepts they introduced, however, do carry on in quality of service management research today.

The work in [13] is interesting in that it proposes a quality of service management scheme that dynamically adjusts resource allocations to meet performance requirements of the applications. The intention is to adjust the operating system to the application, rather than the other way around, which could sacrifice some application functionality to maintain quality of service. In this approach, the applications themselves make most of their own resource management decisions, instead of the operating system or other services. While this may get better results for individual applications, it places additional burden on application developers to develop this logic, and can be dangerous since it relies upon all applications and users to behave in a fair and equitable manner when resources are being allocated.

Unfortunately, the implementation and validation of this approach has been self-described as ad hoc and incomplete; further refinements of this work have not been done.

The work in [145] presents a Conductor/Performer model for providing quality of service translation and coordination services in real-time Mach. The goal of this work is to allow abstract user level expressions of quality of service needs, with the system translating these needs into lower level resource requirements. This is done through table mechanisms and two translation processes that map from user to application level specifications and then application to resource level specifications. Unfortunately, experimentation involved only a single application on a single platform with few quality of service possibilities. Even with this limited scenario, the mapping tables contained dozens of entries—one can imagine the number and size of the tables required for this to have more general applicability.

The Configuration and Resource Allocation (CoRA) [155] approach developed as part of the Dynamic Configuration of Protocols (DaCaPo) project is another interesting approach to resource management. In this approach, applications and the interprocess communication stack is decomposed into modules. Based on user needs, CoRA chooses and configures the necessary modules, and optimizes them to meet these needs through allocating sufficient resources. Unfortunately, for this approach to work, the resource usage of each application and system module must be determined in advance, which would be a time consuming and complex procedure. Furthermore, the quality of service specifications used tend to be at the resource level, making this approach more difficult for end-users.

The User Services Assistant framework [104] is a dynamic approach to quality of service management. In this framework, the environment adapts as much as possible to application needs, unlike most adaptive approaches, which have the application adapt to environment restrictions. The user is central to this process, as it is up to the user the decide when quality is unacceptable, and what to do to correct the problem, with the assistance of feedback from the system. While this ensures that the user gets what he or she wants, this degree of user involvement places a great deal of burden on the user and forces them to be conscious of quality problems in their applications, which is something that quality of service management should attempt to avoid altogether.

The Cooperative Multimedia Environment (CME) approach [5] provides services that allow users to specify quality of service requirements, map these requirements into resource allocations, and makes allocations based on these requirements. This mapping is done in two stages, from user level to application level, and from application level to resource level. The user level to application level mapping is based on experimental studies of users, translating from a simple scale from one to five to different application-specific quality of service metrics. The application level to resource level mapping is also based on experimentation, to determine the resource requirements for each possible application level specification. Unfortunately, this again leads to incredibly large mapping tables, particularly for the application level to resource level mapping. When one considers multiple applications under a wider variety of computing environments, this approach is no longer feasible.

The Matchmaking resource management paradigm [158] was designed to allow for resource management in a distributed environment with distributively owned computing resources. It essentially provides a trading service, allowing resources to be made available to other users and applications through a sophisticated look-up and querying system. While this can provide good matches to user needs, mapping and allocation services need to be put into place.

The QOS-Ticket Model [98] uses tokens called QOS-Tickets to encapsulate application resource reservations and consumption and to mediate quality of service activities. When an application starts, it registers its quality of service needs (referred to as factors in this work) with a quality of service manager; in return, the application receives a ticket from the manager representing the resources to which the application is entitled. This ticket is then used by the application to adapt its own behaviour based on the resources it has been granted. In [97], the notion of Q-Threads is introduced—a new quality of service thread mechanism that makes use of the QOS-Ticket Model. Unfortunately, the quality of service factors that must be provided to generate tickets are resource level specifications, requiring too much knowledge from end users to use. Furthermore, this approach requires applications to adapt to their environment, which can produce undesirable effects.

The work in [32] proposes a dynamic quality of service management framework that eliminates the need for complex mapping and a-priori knowledge of resource requirements. This approaches uses a feedback loop to adapt resources allocated to applications, with the application user in control of decisions to be made. The user has a slider control that can be adjusted up or down for each application, to increase or decrease quality of service respectively. When the user signals that quality should change, resource allocations are adjusted up or down accordingly. Not only does this allow users to have control over their applications, they explicitly give indications of their preferences in case resource shortages mean that some applications may not meet quality objectives. While eliminating the difficulties surrounding mapping, an additional burden is placed on the user, as users are now responsible for managing their own applications, as was the case with [104], as described above.

The work in [138] introduces a quality of service resource management model called QualMan. This is a follow-up to the OMEGA project [141], making use of the lessons learned in that research. In this approach, quality of service resource management is made available as a loadable middleware on general purpose workstations. For flexibility, users can specify acceptable regions of quality of service for delivery, allowing the individual resource controllers to provide a range of service to the application. While application level quality of service requirements are identified as important, the work focusses instead on resource level specifications to avoid complex mapping issues, placing additional burden on application developers and users.

In [147], a dynamic quality of service architecture based on service contracts is introduced. Service contracts provide a mechanism for specifying quality of service requirements and negotiation. Using mapping tables, these service contracts are mapped into resource requirements used for allocations. A variety of novel graphical abstractions allow users to examine their service contracts and how they are broken down and mapped into lower level requirements. Unfortunately, as identified above, mapping table approaches do not scale well when other applications and environments are considered, because of the large amount of empirical data that must be collected, analysed, stored, and searched through to support the mapping process.

The Adaptive Quality of service Architecture (AQUA) [101] proposes a resource management architecture in which applications and the operating system cooperate to dynamically adapt to variations in resource requirements and availability. Every application is linked with resource-specific quality of service managers so that when a quality of service specification is provided by the application, the manager requests the appropriate resources from resource controllers in the system. When resource levels are insufficient, or when additional resources become available, the linked managers can invoke application-supplied adaptation functions to handle the situation. While the quality of service managers are aimed at simplifying the tasks of management for application developers and users, they require resource level specifications, meaning that these developers or users must supply the resource requirements for their applications, which is often a difficult and complex task.

The Quality Objects project (QuO) [213] is aimed at providing a framework for providing quality of service to applications composed of objects (CORBA objects, in particular). This project contains a number of other projects, including one named AQuA, not to be confused with AQUA above. QuO's emphasis is on specification [117, 148], measuring, controlling, and adapting to changes in operating conditions [169, 199]. While the framework appears to be very comprehensive, it is unclear how much of the various services and facilities have been implemented. Resource management facilities are currently limited to a higher level interface to RSVP, without support for other resources (indeed, [199] reports that the response to CPU saturation would be for QuO to "get out of the way" to free resources, and nothing else). Consequently, resource-level specifications are still required. QuO appears to support a variety of different application and system adaptations, but it is never specified which adaptations are provided with QuO and it rather seems that it is up to developers to provide them instead. While a prototype of QuO is available, there is little information available on results from using QuO, and little mention of the overhead in using all of the QuO services to support quality of service. While QuO appears to be a very promising framework, much of it still needs to be flushed out, implemented, and experimented with to determine just how effective it is.

The Distributed Resource Management Architecture (DRMA) [46, 47], provides an adaptive, hierarchical approach to quality of service management. Their aim is to provide a two-tiered “loose” resource management reservation and admission system. Approximations are used in admission control, so only statistical guarantees of quality of service are made. It is assumed that resource allocations are static, unless application resource requirements change, in which case adaptations are made to maintain quality of service. Adaptations can take the form of resource control, reconfiguration, or change of service. While the need for adaptations are recognized, this architecture assumes that most applications know their resource requirements a priori, which is why static resource allocations are used. This, however, is not a good assumption, as there are many factors that cannot be foreseen at implementation time, such as the environment the application is executed in, user perceptions of quality, and so on. Furthermore, while the DRMA approach contains interesting ideas, it has not yet been implemented.

The SWiFT software feedback toolkit [66] provides a framework for constructing, analysing, and visualizing systems that make use of software feedback. SWiFT has evolved over several years, with some of its roots going as far back as the feedback techniques used in the Synthesis kernel [124]. Currently, it is being used to help build applications using feedback to adjust their performance and resource consumption. In the future, plans are to use SWiFT in developing operating system schedulers in Linux. As SWiFT only provides a framework and tools, it is unfortunately up to other developers to actually build applications and systems using it. One example of this is in [178], where SWiFT is used to build a CPU allocation mechanism that uses feedback to adjust application proportion and period. Unfortunately, these adjustments are based solely on measurable system level progress, primarily buffer or queue levels, which can limit the types of applications manageable through this mechanism. More importantly however, this approach does not consider the needs of the end user, and cannot handle user or application level quality metrics as measures of progress. Without the ability to specify quality of service requirements, this allocation mechanism is not suitable for quality of service management. However, it does demonstrate that SWiFT presents an interesting avenue for work in the future.

The work in [113] introduces a Task Control Model for quality of service adaptations in the Agilos middleware control architecture [111], which was later refined in [114] and [112]. The purpose of this model is to provide a mechanism whereby the stability and equilibrium of applications can be analysed and maintained, while providing fairness to other applications in the system, in light of fluctuations in available computing resources. The general approach is to adapt applications to their environment for these purposes, as opposed to adapting the environment to meet the needs of the applications. Critical quality of service parameters are maintained, while non-critical parameters are traded off by a variety of adaptations. These adaptations usually take the form of changing request or delivery rates in applications. Unfortunately, much of the control theory developed and presented is applicable only to application adaptations, as opposed to environment adaptations. Since the application is adapting to meet environment constraints, it is unclear exactly what kind of quality is being delivered to users, as often application functionality must change in the process, and the application user may no longer be receiving what was expected. Instead, wherever possible, the environment should adapt to meet application needs first; the application should only adapt when this is not possible.

Another application of control theory to achieving service objectives can be found in [149]. In this approach, historical measurements of an application's performance are used to generate transfer functions and control system models. These models are then used to construct a controller capable of tuning performance. In this work, the target application was Lotus Notes, with the objective being to maintain the length of the server request queue, by adjusting the number of users allowed to access the server concurrently. Both the models and the resulting controller are analysed and evaluated. While this approach is currently focussed on tuning a single application parameter to meet service objectives, it can be extended with great effort to tune multiple parameters or even resource allocations and other environment variables instead. Unfortunately, this approach requires considerable experimental measurements to construct the transfer functions in the control models off-line. These models would need to be built for each application being used, and tuned for each environment configuration and application workload. This would require a large number of models, and considerable experimentation to generate them. Nevertheless, this approach is promising, particularly when historical measurements have already been collected.

Other work in this area examines entirely new approaches to resource management. Two good examples of this are the */reserv* file system [24] and resource containers [9]. The */reserv* file system provides a uniform interface for organizing and interacting with a system's resources. Resource containers tie resources to independent activities as opposed to processes, allowing for improved accounting and fine grained control of resource scheduling and allocation. While these approaches contain excellent ideas for improving resource management, application users and developers are once again burdened with the details of resource management, as they were in many of the cases above.

While not directly aimed at quality of service, the work in [163] is quite interesting as it focusses on manipulating applications, as opposed to the operating system, to address performance problems caused by resource issues. Instrumentation in this work measures application behaviour, and uses this information to rewrite application binaries to optimize code layout and improve memory-related performance. In theory, code layout could be optimized to improve CPU-related performance, and so on. Unfortunately, this optimization must be done off-line, after application termination, and cannot be used to improve the performance of executing applications dynamically.

Along similar lines is work done in [27], where idle resources are used to speculatively execute an application to analyse and predict its future I/O accesses. This information could then be pre-fetched, for improved performance. In theory, this work could be extended so that in addition to predicting future I/O accesses, future CPU, network, and memory resource needs could be predicted and preallocated to maintain stable levels of good quality of service.

#### 2.2.3.2 Goal Oriented Resource Management

Goal oriented resource management is an adaptive approach to resource management that combines areas such as operations research and stochastic control with resource scheduling and management [144]. The philosophy behind this approach, as stated in [144], is that users should be able to specify "*what* the performance expectation for their work is, *not how* to achieve this expectation". In this approach, only performance goals for applications are specified, and the underlying system adjusts resource allocations until the goals have been satisfied. Consequently, this approach to resource management has the potential for giving insight into application-level quality of service management. Interesting proposals in the area of goal oriented resource management have been presented in [22, 60, 144]. Unfortunately, much of this work is largely theoretic and algorithmic. Proof of concept, if any, has largely been in the form of simulation studies, with few general prototype implementations.

The work in [122, 123] is a relatively new continuation of the goal oriented resource management work. The general objective of this work is to develop models and algorithms to support goal oriented resource management in a database management system and to create tools based on the models and algorithms that can be used by administrators to dynamically reconfigure the system. The aim in doing this is to provide "self-managed databases" [212]. Prototype work is currently being developed to tune buffer pools and other resources in IBM's DB/2 Universal Database (UDB). While this work is preliminary and directed only at databases, it appears to be quite promising.

#### 2.2.3.3 Economics-Based Approaches

Several approaches to quality of service management are based upon economic models and principles. Generally, economics-based approaches make use of free market principles such as supply and demand curves, some form of utility model, or both.

The earliest work in [53, 119, 181] was based entirely on supply and demand curves, in which resources were the commodities, and user applications or application requests were the consumers in the market. Depending on the currency available to each competing application or application request, prices to access resources fluctuate. This, in turn, determines the amount of resources allocated to each, and the quality of service delivered as a result. This work has been revisited with more sophisticated markets, as in [102, 204]. Unfortunately, with supply and demand curves alone, it is difficult to capture quality of service requirements; a quality specification in this case would be limited to only how much money an application would be willing to spend to access a particular resource. While this work was quite promising, much of it remained largely theoretic and algorithmic, with proof of concept limited mainly to simulation studies and no general prototypes available, as researchers moved on to either more advanced economic approaches, or entirely different approaches to quality of service management.

Utility functions (also known as benefit functions) are used in economic models to express the benefit a consumer derives from having various amounts of available commodities. The goal in this kind of a system is to maximize the global utility, which can be derived from an aggregation of individual utility functions. Since the concept of utility closely corresponds to quality of service under a wide variety of resource allocations, several researchers make use of utility functions in their work, attempting to maximize the global utility, or quality, delivered to end users.

The work in [29, 99] is a good example of the use of utility concepts, providing an adaptive and dynamic solution to allocating all of a system's resources for quality of service, with a Linux prototype developed. [121] extends this previous work to apply to network resources as well, in the development of QoSNet, a small experimental network under construction to test this work. [28] is another example of this principle, using similar multi-dimensional functions to capture quality of service needs, without a prototype implementation, however. The resource management work in the Amaranth project called Q-RAM (Quality of service based Resource Allocation Model) [107, 157], detailed in [106], also takes a utility-based approach to maximize benefit realized by multiple applications with multiple quality characteristics consuming multiple resources. The work in [17, 150] takes similar utility-based approaches to allocating resources in telecommunication systems. Unfortunately, all of this work assumes the existence of known functions or mappings to derive both utility functions and resource requirements for an application from some kind of quality profile. Since these functions or mappings do not exist, and it can be incredibly difficult for a user to develop a quality profile for an application before they have had the opportunity to use it, there are serious practical limitations to these approaches. Further, these approaches typically use overly-simplistic techniques for computing global utility, which is important to determine accurately, as the goal of these approaches is to maximize this utility value. Lastly, proof of concept in this work tends to be largely theoretic or simulation-based, with very few general purpose implementations.

As work in this area continues, models are becoming increasingly sophisticated. The work in [160], for example, takes into consideration supply and demand curves, utility functions, and price sensitivity. In addition, resources can be "purchased" or "traded", with two levels of resource providers—retailers and wholesalers—as well as other resource consumers. Unfortunately, this work requires users to provide profiles containing utility functions to the system, which can be quite cumbersome. (This is similar in difficulty to providing quality of service specification to resource mapping functions, provided above, as users would need to know the benefit provided by each possible resource allocation.) Once again, validation is provided only through simulation, without any prototypes available.

#### 2.2.3.4 Perceptual and Behavioural Quality of Service

Interesting and innovating work has also been carried out towards quality of service management through the use of perceptual quality metrics, which are based upon how humans perceive quality in their applications. Further work also examines how user behaviour can be used as feed-back into the quality of service management process.

Much of the literature in this area makes use of a five-level scale, with quality ratings ranging from imperceptible quality impairment (Level 5) down to very annoying quality impairment (Level 1) [12, 80, 198]. With these approaches, users can easily select the level of quality desired at a very high level of abstraction, perhaps using slide bars, radio buttons, or some other graphical mechanism. In work such as [4, 58], studies have been carried out to determine how human users would map these quality levels into application-level quality metrics for a variety of applications. For example, in subjective tests detailed in [4], users of video applications rated quality provided by the applications at different frame rates and resolutions, using the five-level scale. These test results could be used to derive mappings to convert from quality levels to target frame rate ranges, with slight adjustments as necessary on an individual basis. New work along similar lines is also beginning in [44]. Unfortunately, the work in this area also attempts to map these derived application-level metrics to resource requirements directly. This, however, encounters the difficulties mentioned above with quality of service mapping, as trying to determine the resource requirements for an application under all possible hardware and software configurations, workloads, and resource demands is extremely difficult, if not intractable.

Using user behaviour as a feedback for quality of service management is a relatively new concept. This issue was raised in [4], but unfortunately no design or implementation work was ever completed. [147] also examines this issue, providing an environment that fed into the quality of service management process when users interacted with it. However, this was only done in the context of their own 3D landscape environment, and not generally available production environments. The SCUBA (Scalable ConsensUs-based Bandwidth Allocation) protocol [6] is aimed at allowing multimedia sources to account for receiver interest in adjusting transmission rates. While this work is interesting, it is geared towards only a particular class of applications, and does not adjust resources allocated or consumed; instead only modifying application behaviour. Additional research into multimedia applications such as [126, 127, 188, 203] and others has yielded adaptive applications capable of responding to user feedback (for example, by not transmitting audio streams when audio is muted by the user). Unfortunately, much of this work is domain-specific and limited in scope, quite often only to reducing work done in response to user feedback, without deallocating and descheduling resources, which can lead to further problems.

Microsoft Research is studying similar issues with the design of its next generation operating systems and applications based on its new ".NET" strategy, aiming for "attention sensitive user interfaces". The Lumiere project [74], for instance, uses cameras and microphones, in addition to other user feedback, to determine the goals and needs of users. Currently, this data is being used primarily for alerting and communicating with the user [75]; however, it could very easily be applied for adjusting resource allocations to user applications. Other work in [73] aims to use idle time when users are not fully using resources for precomputation and analysis, which could be used in making more informed resource management decisions in the future.

Along similar lines is work in analysing patterns of user actions for a variety of purposes, including adaptive scheduling. In theory, by analysing what a user does, and comparing that to a known history of the particular user or users in general, one can predict what a user is about to do, and provide resources in advance. Unfortunately, research in this area has yet to reach this point. The work in [39, 135, 100] examines extracting behavioural patterns from user command sequences and other information in Unix environments, but only use these patterns to preemptively launch new commands, without special resource considerations. [61, 62] examines how to adapt scheduling based on user activity, but only in the context of particular industrial settings, and not in terms of general computing resource management.

#### 2.2.3.5 Microkernel and Extensible System Approaches

Many microkernel operating systems, such as Mach [2], Chorus [165], and V++ [31] provide support for changeable resource management services outside of the kernel for memory management, and so on [1, 69, 128]. This allows for great flexibility in the kinds of quality of service support that can be made available. Some microkernel operating systems, such as Mach, have real-time support as well [194]. While the enhanced resource management capabilities present in this work is impressive, there also tends to be heavy burdens on application users and developers to port applications and make use of these facilities. Unfortunately, it is often the case that resource level specifications are required to make use of improved resource management services available in these environments, which can lead to problems as described earlier in this section.

Research effort has also been directed towards providing extensible systems where services, including resource management services, can be changed dynamically and/or tuned to particular applications [16, 50, 110]. While this can offer even great flexibility than microkernel approaches, this places a large burden on application developers which may now also have to develop the operating system services required to support quality of service in their applications. In some cases, this may be desirable, but in others, it may be too cumbersome to deal with. Also, since these environments are not mainstream computing environments, as is the case withe most microkernel operating systems, application porting is necessary to take advantage of additional operating system support for quality of service.

Another interesting research system was the Rialto operating system [85] from Microsoft Research. This system showed great promise for quality of service management, with its real-time support and enhanced resource management capabilities. Unfortunately, Rialto is no longer being researched as a separate entity; however, some of its concepts may eventually make it into future versions of the Windows operating system.

#### 2.2.3.6 CPU Management Research

A variety of novel approaches to CPU scheduling and allocation have been developed, including processor capacity reserves [108, 129], rate-based execution [84], rate-controlled scheduling [209], hierarchical scheduling [68], proportional share resource allocation [179], scheduling graphs and time constraint satisfaction [86], biased virtual finishing times [143], lottery scheduling [154], stride scheduling [202], and real-time upcalls [67]. Most of the above approaches provide significant benefits to quality of service management over traditional approaches to scheduling, and many have been implemented, either in microkernel environments such as real-time Mach, or in open-source operating systems such as Linux and FreeBSD. The work described in [33, 87, 115, 173] is also significant, providing flexible techniques to introduce improved CPU scheduling schemes into existing commercial systems such as Solaris and Windows NT, without kernel modifications. Instead, these systems work at the user level and employ their own scheduling system that works in addition to the system scheduler, rather than replacing it altogether. In doing so, arbitrary scheduling mechanisms can be used, using any of the approaches listed in this section above, or any of the CPU scheduling algorithms given in Section 2.2.1.

Unfortunately, the above approaches require environment quality specifications in terms of resource needs, typically in terms of proportions, execution periods, execution rates, deadlines, and so on. This can make these systems more difficult to use for application users and developers, as these resource needs may not necessarily be known in advance. Furthermore, most of these approaches are also static in nature, making firm resource allocations for the life-time of the application. Those that are not completely static allow resource specifications to be a range of acceptable values [33, 108], giving some flexibility to the scheduler, but not much. Other approaches such as [84, 209] also permit a certain amount of adaptation, though this is somewhat limited.

#### 2.2.3.7 Memory Management Research

As noted in [34], relatively little research effort has been recently directed towards memory management to support quality of service. This is unfortunate given that applications that are time sensitive, for instance, can suffer greatly under memory shortages and page faulting. The penalty incurred for a page fault is in the range of 700,000–6,000,000 clock cycles (on the order of 10 milliseconds) [153]. For time sensitive applications, a few page faults could cause a noticeable degradation in performance. Distributed multimedia programs, such as video conferencing, video-on-demand, and so on, are good examples of these applications. If a video application is expected to deliver video at a rate of 30 frames per second (or 33 microseconds per frame), quality of service requirements could be violated when only a few page faults are encountered [34].

The work in [34] is an interesting approach, whereby processes needing guaranteed access to memory can share locked memory from a memory server. Programs can then store data into this shared memory, guaranteed that it will never be paged out and will always be available for accessing. Unfortunately, this approach requires a-priori knowledge of what data needs to remain in memory, putting additional burden on the programmer. Furthermore, code and stack segments for processes remain pageable, so memory problems can still occur.

A different approach is taken in [49]. In this work, processes can specify minimum and maximum memory bounds, providing guaranteed levels of memory access. Like the approach above, this requires the programmer to have knowledge of a program's memory needs in advance, which tends to be difficult to predict under a wide variety of workloads. Furthermore, while each process can have a guaranteed amount of memory available, the operating system still decides which pages are kept in this memory and which are not, which could lead to unexpected and undesirable results.

#### 2.2.3.8 Network Management Research

Because networks are traditionally the source of bottlenecks in computing environment, most early work in quality of service management focussed on improving the management of network resources. This led to many of the concepts used throughout quality of service management such as admission control, resource reservations, load shaping, and so on.

Several network protocols have been developed to assist in delivering quality of service. RSVP [211] provides an excellent resource reservation mechanism to allocate network resources on a per flow basis; doing so with some scalability issues. MPLS (Multi-Protocol Label Switching) [164] is another mechanism by which flows of packets are labelled to support traffic forwarding and resource reservation decisions. DiffServ [18] provides mechanisms for prioritizing network traffic easily and linking network resource management behaviour with data packets. RTP [171] provides a real-time communications protocol with end-to-end network transport functions suitable for applications transmitting real-time data, such as audio, video or simulation data. It does not address resource reservation and does not guarantee quality of service for real-time services, but rather relies on underlying services to do so. ST-II [195] is designed to provide the delay and data-rate characteristics necessary to support multimedia applications. Although intended to replace the Internet protocol (IP), more focus has been given to the above protocols to meet these needs. IPv6 [40] is designed to have more built-in support for quality of service through the inclusion of flow labels and traffic classes. It is likely that some form of reservation and differentiation services will be required to make use of these extensions.

At a higher level, interfaces and libraries are required for users to interface with these network quality of service solutions. On Windows platforms, the Winsock API [130] is used to do so. This API interacts with provided services to make use of GQoS, RSVP, and so on, as discussed in Section 2.2.2.2. This can be done in a dynamic fashion, allowing resource allocations to change as necessary throughout the life-time of an application. On other platforms, there is no standard networks API that supports quality of service, although researchers are beginning to propose either extensions of the standard BSD socket API [109] or new APIs altogether, such as QoSockets [56].

At a lower level, work in ATM [156] has made it possible to make resource allocations for channels spanning an entire network. Recent advances in standardization at the ATM Forum will allow these allocations to be dynamic throughout the life-time of a connection, as opposed to the relatively static mechanisms widely in place now [43, 146, 176, 197, 206]. Work in a real-time Ethernet protocol [200] is also promising, but is likely not to be accepted in the near future due to the growing acceptance of Gigabit Ethernet and, soon, 10 Gigabit Ethernet. DOCSIS [189] is aimed at providing improved quality of service management over cable modem services, which are growing in popularity to provide high-speed Internet connections in homes and small businesses.

In addition to these protocols, new approaches to managing network resources have also emerged. These include HeiTS [41], Tenet [54], QoS-A [25], the QoS Broker [140], and others. It would appear, however, that these relatively proprietary approaches are being left behind in favour of the more standardized protocols given above.

Another interesting addition to quality of service management from network management research comes in the form of proactive management, whereby performance problems or faults are predicted and dealt with before they actually occur. The work in [70, 72, 193] is quite promising in this area.

#### 2.2.3.9 Disk and I/O Management Research

Despite recent advances in technology, disk I/O operations are still approximately $10^5$ to $10^6$ times slower than main memory accesses [65]. This means that disk operations are quite expensive in terms of performance, and can have a significant impact on the quality of service of an application. As noted in [65], if disk performance does not keep pace with CPU performance, disk I/O operations become a large bottleneck in high-performance computations. Unfortunately, there has been relatively little research effort into providing improved disk and I/O management for quality of service management. To date, improved quality of service has hinged primarily upon faster disks, higher performance I/O buses, more intelligent caching, and so on. This will have to change in the near future.

In light of this, there has been some work conducted into optimizing the disk and I/O systems of operating systems as a whole, to provide better overall performance to user applications. This work tends to either propose new scheduling techniques and approaches to improve performance [30, 172, 205] or reuse existing techniques and approaches while minimizing data copying operations to improve performance [38, 151, 191]. While this work provides substantial performance gains over more traditional approaches, it is not sufficient for quality of service management on its own, as there is no way to make specific resource allocations for specific application processes.

[136] notes the importance of per application management of disk and I/O resources, necessary for quality of service management. While relatively little has been done towards this, the work to date is quite impressive. In [49], support is provided to set I/O priorities for individual applications to give them preferential access to disks. While this does not give absolute resource allocations, it is definitely a step in the right direction. The work in [23] allows weights to be attached with I/O requests and uses the relatively new YFQ algorithm to schedule resources. The Cello disk scheduling framework [174] defines class-independent and class-specific disk and I/O schedulers, which allow service to be provided to meet application requirements. While several class-specific schedulers can be provided for use, optimal results can only be achieved by further customizing these schedulers to particular applications or by developing new ones, which can be a complex process. Disk shuffling, proposed in [166], is an interesting approach where disk layout is adapted at run-time so that data needed by time sensitive applications is arranged for optimal access times. While this can lead to performance improvements, it is difficult to use this approach in anything more than a priority scheme, where applications with higher priorities receive better arrangements than those with lower priorities.

## 2.3 Discussion

What approach should a solution to the problem of quality of service resource management take? What characteristics should a solution have? What kind of support should a solution provide? How close does the current work in the area come to providing this, and what needs to be done? In this section, we sort through these questions, using the previous sections as a reference, and use this as basis for defining requirements for this work.

### 2.3.1 A Static or Dynamic Approach?

Static approaches to quality of service management are ideally suited for applications with *hard* quality of service requirements, such as flight control systems, chemical process control systems, and patient-monitoring systems. In these applications, quality of service requirements must be met to avoid disastrous consequences. In these cases, applications are usually analysed with excruciating detail, and resources are over-provisioned to ensure that these disasters are avoided. Logistically speaking, this is not as bad as it could be, since the applications are few in number, and there often constraints on the operation of these applications (for example, the exact hardware configuration is known in advance, and cast in stone). Despite this, doing so is still an incredibly expensive prospect, but the money is well spent when one considers the consequences otherwise.

Most applications, however, do not have these hard quality of service requirements. Instead, they have *soft* quality of service requirements, where applications are still of value when requirements are missed on occasion. This includes multimedia applications, office productivity applications, development tools, and so on. In these cases, as noted in [8, 13, 45, 159], it is inherently difficult to characterise these applications and their needs. Indeed, when one considers the sheer volume of applications in this category, in addition to time and financial constraints, it is not feasible to conduct extensive analysis of the applications or over-provision resources. Since these applications are often general purpose, for general purpose environments, the number of environment configurations possible makes this analysis intractable. As a result, we cannot determine in advance the resource allocations required by these applications to meet arbitrary quality requirements.

Another factor to consider is that applications and their environments in this case are also dynamic. Users often change the quality requirements for their applications, meaning that different resource levels are required throughout the life of an application. Depending on what an application is currently doing, different resource levels are also often needed, even if user requirements do not change. As applications enter the environment, resource availability changes. All of these issues make for a constantly changing target for resource management.

When all of this is considered, static approaches do not work well in general purpose environments for general purpose applications. Even if the single initial resource allocation is correct at the time, which is very unlikely considering all of the factors that this depends upon, it is even less likely that this allocation will be suitable throughout the life-time of a dynamic application. As a result, in these situations, static approaches either tend to deliver insufficient resources to applications, causing them to meet quality of service requirements, or too many resources, resulting in costly resource wastage and a reduced number of users and applications that can be supported. This sentiment is echoed in [13, 159].

Since we are interested in general purpose applications in general purpose environments, a dynamic approach to quality of service management must be taken. In doing so, we do not need to know the relationship between resources allocated and quality of service delivered as a result. Instead, dynamic adaptations will ensure that the appropriate resource allocation is found at run-time.

Some researchers already identify dynamic quality of service management as an important issue, and reflect this in their work, including [13, 32, 104, 113] and others; as such, their approaches may be good starting points for a solution. Future research planned for operating systems development appears to be heading in this direction as well [20]. A dynamic approach, however, leads to other issues, as discussed in later sections of this chapter.

### 2.3.2 At Which Level of Management?

The possible management levels, as described earlier in this chapter, are network, system, application, and user. The choice depends on the coverage needed for the solution, as well as types of quality of service specifications to handle.

As far as coverage is concerned, an end-to-end quality of service solution is necessary, as noted in [139]. Not only must we consider the network, but also the hosts that applications execute on. Resources on all of these elements must be effectively managed, or else shortage of these resources could lead to quality problems. For example, even the best network bandwidth management solution will not provide adequate quality of service for an application when it is starving for CPU cycles on its local host. Without enough cycles, data cannot be produced, transmitted, and received quickly enough to make use of the bandwidth available. Consequently, a network level approach to management is not sufficient. A system level approach or higher must be used.

Which of the remaining approaches to use depends on the type of quality of service specifications the solution should handle. Ideally, specifications should state what user expectations exist, not how to achieve them [44, 144]. This is the case for several reasons:

- Users are the ones making quality of service specifications. Users often have difficulty in operating their applications, let alone trying to tune the application or its operating environment to meet their quality needs. As a result, it is better to leave users to specify what their requirements are (which on its own can pose challenges to some users), and let the underlying quality of service management solution be concerned with how to meet those requirements.

- Typically, the causes of quality of service related problems can only be diagnosed at run-time by checking the current state of an application and its operating environment. Only then can an action plan be developed to correct the problem and bring quality of service back into line with requirements. Since quality of service specifications are usually made when an application begins and only slightly adjusted throughout the life-time of the application, one cannot anticipate and provide solutions for all of the potential problems that may or may not occur in the future.

- As mentioned earlier in this chapter, there is an exceedingly large number of variables influencing quality of service, and no known relationship exists between these variables and the level of quality delivered by an application. To repeat, these variables include computing environment configuration, operating system configuration, application characteristics, application workload, environment workload, and so on. Consequently, it is exceedingly difficult, if not impossible, to determine the resource needs for an arbitrary application under all of these conditions, ever for application developers and system administrators.

User and application level quality of service specifications deal with what user requirements exist, since they can describe the desired behaviour of the application. Resource level specifications, on the other hand describe what an application needs, not what an application does; thereby dealing with how to achieve specifications instead of what those requirements are. As a result, user and application level specifications are preferable over resource level specifications whenever possible. Consequently, an application or user level approach to quality of service management should be taken.

Ultimately, a user level approach is most desirable. However, such an approach depends on many psychological, perceptual, and sociological factors that are outside the scope of this research. A user level approach is also heavily dependent upon application level techniques, which, at this point, do not exist. As a result, in this work, an application level approach is taken, with some user level extensions provided. The work in [4] provides interesting ideas of use to this work for user level quality of service management, but takes a static approach which ultimately limits its applicability. Dynamic application level approaches such as [13, 32, 104, 113] may be more applicable, but they too ultimately have their limitations, as will be shown below.

### 2.3.3 How Should Adaptations be Made?

A dynamic approach to quality of service management inherently requires some form of adaptation. As noted in [19], either the computing environment must adapt to meet the needs of applications executing in it, or the applications must adapt to meet the constraints imposed by their environment.

Adjusting an application to meet environment constraints usually requires sacrificing application characteristics not constrained by quality of service requirements in favour of those that are. While this may meet quality of service specifications in the strictest sense, this is not in the true spirit of quality of service management—satisfying the end user. This sentiment is echoed in [13]. For example, if a video application cannot display its video at the target frame rate set by its quality of service specification, the application could reduce its resolution, colour depth, or picture quality to meet its target. While meeting the specified requirements, the user is not likely to get what was expected. This approach, nevertheless, is taken in [113].

Adjusting the computing environment to meet application needs avoids these problems. The environment adapts resource allocations, schedules and other parameters to meet the quality of service constraints of applications as best as possible. This makes better use of computing resources and avoids sacrificing unconstrained application characteristics when doing so is unnecessary. It is important to point out, however, that the computing environment may not be able to adapt to meet the quality of service requirements of all applications at all times, particularly during periods of extreme overloading. In such cases, it might be necessary to have applications make additional adjustments on their own, but only in these exceptional circumstances.

It is important to make sure that the adaptations and adjustments occur in an automated fashion, with as little user interaction as possible, as recognized in [20]. Constantly requiring the user to determine if and when an adjustment is necessary or what adjustments should be made would place an enormous burden on the user, who likely does not have the background, training, or ability to make these decisions. For example, requiring the user to select when quality is no longer acceptable throughout the life-time of an application, or to select the additional resources an application needs is not acceptable. This, however, is the approach taken in work such as [32, 104]. In extreme cases, however, it might be necessary to consult the user on certain adaptation decisions. For example, if the computing environment can no longer adapt to meet quality of service requirements for a video application, the user could be asked whether it would be preferable to proceed at a lower resolution, reduced colour depth, or reduced image quality.

Finally, it also important to ensure that adjustment decisions are not solely in the hands of individual applications, as was done in [13]. While developers could build applications to automatically adapt their own behaviour, or perhaps even make changes in their own resource allocations, this approach is generally unacceptable. First, this places an enormous burden on application developers, as their applications must be considerably more complex to handle their own management. Second, individual applications are concerned with only meeting their own needs, without regard for other applications or broader goals of the computing environment. Allowing individual applications to have final control over how the computing environment is used can potentially result in chaos.

In this work, adaptations will first be made by the computing environment as much as possible, and adaptations will only be made by applications when the computing environment can no longer do so itself. These adaptations will be automated, with user consultation only when absolutely necessary, if at all. Finally, these adaptations will be decided upon and carried out by services with system-wide, or perhaps environment-wide, vision and responsibilities.

### 2.3.4 Other Considerations and Requirements

In addition to the above requirements, the following must also be considered.

#### 2.3.4.1 Speed and Efficiency

Adaptations and adjustments made by the solutions must be done quickly. Since several adjustments may be required to bring quality into line with requirements, each adjustment must be made quickly to avoid long periods when requirements go unsatisfied. Furthermore, due to the dynamic nature of applications and their environments, adjustments are made towards potentially moving targets—these adjustments must be nimble to keep up with change. In fact, as noted in [21, 81, 86, 124, 137], faster adaptations are often favourable over more correct adaptations that require more time or other resources.

In addition to time-wise efficiency, a solution must also be efficient in terms of resource consumption, and produce little overhead, except in rare circumstances [77]. The more resources that management uses, the fewer that are available for user applications.

#### 2.3.4.2 Flexibility, Applicability, and Portability

The solution developed must be flexible and applicable to a wide variety of situations. Many approaches taken in the related work in this area, however, such as [113, 149], are tied very much to particular applications or particular environments, and lack this.

Flexibility includes the following:

- Supporting the management of all computing resources.
- Supporting the management of a wide variety of applications.
- Supporting a wide variety of existing commercial computing environments.

These factors, in turn, imply that the solution must support a number of underlying approaches to resource management, scheduling, and allocation. To do so, flexibility is a key requirement.

#### 2.3.4.3 Usability

As mentioned earlier, it is important that a quality of service management solution be usable to both application users and developers. Neither users nor developers should need to be quality of service experts to make use of such a solution; rather, they should be shielded from both the solution and its internal complexities [19]. If not, it will likely be given little serious consideration.

To users, the solution must be transparent and have very little impact on the way they use their applications. It should place little burden on them, and not be difficult to use.

Developers also need a transparent solution. It should require very little deviation or additions to the tasks they must already carry out, and should integrate seamlessly into their applications. A clean, consistent, and easy to use application programming interface is necessary. Again, there should be as little burden as possible placed on developers.

### 2.3.5 Summary

From an evaluation of related work in this area, we have been able to identify the desired characteristics of a solution to the problem of quality of service resource management undertaken in this work. The approach taken must be dynamic, user or application level, based on environment adjustments, independent of the applications and computing environment involved, and should not place a large burden on application users, developers or administrators.

This work is unique, as no other related work in this area takes this approach. A quick review of this related work verifies this:

- Resource management algorithms and policies discussed in Section 2.2.1 represent only part of an overall quality of service management solution; users and developers should be protected from dealing with the algorithms directly as they tend to address resource management issues at a lower level. Further many of these approaches treat every process the same, making it difficult to meet individual quality needs of each process.

- Needless to say, commercial approaches to quality of service management discussed in Section 2.2.2 tend to be very platform dependent, and not applicable to a variety of computing environments. These approaches also tend to focus on each resource in isolation, and do not provide a unifying resource management solution. Furthermore, most are too low-level and require that the user or developer be aware of the underlying resource management.

- Goal oriented resource management, discussed in Section 2.2.3.2, provides interesting principles for quality of service management, but originally did little to develop them into a working approach to quality of service management. New work in goal oriented resource management addresses this, but is focussed primarily on database management systems, and is not directly applicable to other applications.

- Economics-based approaches, discussed in Section 2.2.3.3, tend to make the assumption that the relationship between resources allocated and quality of service delivered is well known in advance. This, however, is not a good assumption in practice, and one that we have taken steps to avoid.

- Perceptual or behavioural approaches to quality of service management, discussed in Section 2.2.3.4, provide some insight into handling user-level quality of service issues. These approaches, however, assume that functions or mapping tables exist to map application-level quality of service requirements to resource requirements. This again, however, is a bad assumption in practice, and one that we avoid.

- Microkernel and extensible system approaches, discussed in Section 2.2.3.5, often focus on resource-level quality of service management, placing a great deal of burden on application users and developers. These approaches tend to place further burdens on developers in requiring them to code resource management software to support quality of service for their applications.

- Research into management of individual resources, discussed in Sections 2.2.3.6, 2.2.3.7, 2.2.3.8, and 2.2.3.9, provide good approaches to managing CPU, memory, network, and disk resources individually. For quality of service management, however, these approaches need to be combined under a single framework that supports the management of all computing resources.

- After eliminating these sections of related work, the only category remaining is for quality of service architectures and frameworks, which is the category into which this work falls. Many approaches to quality of service management in this category, however, are static. Of the few approaches that are dynamic, many are not at the appropriate level of management. Of those that are at the right level, none make their adjustments or adaptations in the same as we propose, in which environment adaptations are made first, with application adaptations made only when necessary, and in which adaptation decisions are made by the quality of service management system, not by the user or applications. Further, most approaches make simplifying assumptions that ultimately limit their applicability, or place unacceptable burdens on those that either use or implement applications or their execution environments.

After these considerations are made, there is no related work taking the same approach to quality of service management as this work.

Consequently, the approach taken in this work is unique. While building upon concepts present in other work, this work delivers an innovative and insightful solution to quality of service resource management that addresses the many difficult unsolved problems left by previous work in this area.

# Chapter 3

# Formal Problem Definition and Analysis

In this chapter, we delve into the formal theoretical aspects of the problem of quality of service resource management. In doing so, we accomplish two goals. The first goal is to provide formalisms useful in characterizing the problems faced in this work, and designing algorithms later in the work. The second is to investigate, analyse, and discuss the computational complexity of these problems, which have ramifications on the directions this work takes.

## 3.1 Modeling the Problem

In this section, we develop a set of definitions to be used in describing the problem of quality of service resource management. With this in hand, we can begin to formally analyse this problem.

### 3.1.1 Assumptions

The assumptions used in this work are discussed below.

- User applications, distributed or not, can be adequately modeled by their individual component processes. This makes sense since each component process may have its own quality needs, and it is ultimately these processes that are the entities that can be allocated resources by the computing environment. While communicating processes within an application may depend upon one another, dealing with the required dependency-discovery and resource synchronization issues exceed the scope of this work. Consequently, application processes are treated independently.

- Computing environment resources are all treated in a similar fashion. It is assumed that they are allocated in discrete units, and not in a continuous fashion. This is reasonable since this is the way most resources are managed in reality. We also assume that each resource has a finite capacity, and that it is possible to be allocated none of a resource, or perhaps even all of it. Lastly, we assume that resources are not added while the computing environment is active; if this is done, we assume that the problem can be modeled appropriately as a sequence of problems, each with a constant supply of resources.

- We assume that the quality of service delivered by a process can be characterized by some kind of function. Such a function would output the quality delivered by a process under the input resource allocation according to some quality of service metric. As mentioned in the previous chapter, a quality of service metric is a measured value of an attribute that represents some element of application behaviour.

  We assume that each process has such a function, but that we do not necessarily know what this function is in advance. Consequently, satisfying quality of service requirements will entail making resource allocations and comparing against the output of this function to determine if we are meeting objectives, and to help determine what to do if we are not. We further assume, for simplicity, that each process has one such quality of service function. In reality, processes could have several, but resolving conflicts arising from attempts to satisfy multiple such functions at the same time is beyond the scope of this work.

  Lastly, we assume for simplicity that these functions can be computed in polynomial time. If not, variations of the quality of service resource management problem found to be NP-complete would be NP-hard instead.

- We assume that there is one quality of service requirement to satisfy per process. Once again, this is for simplicity to avoid conflicting requirements that cannot be satisfied at the same time.

These assumptions lead to requiring a variety of notation elements. A summary of these elements is provided in Table 3.1 for the convenience of the reader. This notation is described formally in the sections below.

**Table 3.1** Summary of Notation

| Notation | Description |
|---|---|
| $p_i$ | Denotes process $i$ |
| $r_i$ | Denotes resource $i$ |
| $r_i$.granularity | Denotes the integral allocation amount of resource $i$ |
| $r_i$.capacity | Denotes the total capacity of resource $i$ |
| $r_i$.levels | Denotes the number of levels of allocation of resource $i$ |
| $a_i$ | Denotes the resource allocation to process $i$ |
| $a_{i,j}$ | Denotes the allocation of resource $j$ to process $i$ |
| $q_i$ | Denotes the quality of service function of process $i$ |
| $e_i$ | Denotes the expectation function of process $i$ |

### 3.1.2 Processes

Each host in a distributed computing environment has a finite set of processes at any time that execute and consume its resources. When we consider all processes across all hosts, we come up with PROCESS, the set of all distributed system processes. We can refer to these processes individually as $p_1$, $p_2$, $p_3$, ..., $p_m$.

### 3.1.3 Resources

Each host and network device in a distributed computing environment has a set of resources that are provided to application processes executing in the environment. When we consider all of the resources across all hosts and network devices, we have RESOURCE, the set of all distributed system resources. We can refer to these resources individually as $r_1$, $r_2$, ..., $r_m$.

Each resource has the following attributes:

- granularity – The smallest integral amount in which the resource can be allocated.
- capacity – The total amount of the resource available for allocation, which is an integral multiple of its granularity. The capacity of a resource is finite.
- levels – The number of levels of allocation of the resource, which can be computed as $\frac{\text{capacity}}{\text{granularity}}$. Since capacity is a finite integral multiple of granularity, levels will also be a finite integer.

We denote the capacity of resource $r_i$, where $1 \leq i \leq |\text{RESOURCE}|$ to be simply $r_i$.capacity. Likewise, we refer to the granularity of allocation of resource $r_i$ to be $r_i$.granularity and the number of levels of allocation of $r_i$ to be $r_i$.levels.

### 3.1.4 Allocations

Given RESOURCE, we define an allocation of RESOURCE to simply be an integer tuple, with each element of the tuple indicating the amount of each corresponding resource being allocated. We denote the set of all possible resource allocations as ALLOCATION.

We denote the resource allocation to process $p_i \in$ PROCESS as $a_i = <a_{i,1}, a_{i,2}, \ldots, a_{i,m}>$, where $m$ is the size of RESOURCE. $a_{i,j}$ denotes the allocation of resource $r_j \in$ RESOURCE to process $p_i$. To ensure each resource allocation is within the capacity of the given resource, each such $a_{i,j} \in \mathbb{Z}_{r_j.\text{levels}+1}$.

### 3.1.5 Quality of Service Functions

Quality of service characteristic functions map resource allocations to a particular quality of service metric. Without loss of generality, we assume that all metrics have real values. We let QUALITYFUNCTION denote the set of such quality of service functions. As noted in Section 3.1.1, while we assume that such functions exist for each process, we also assume that we do not know these functions in advance.

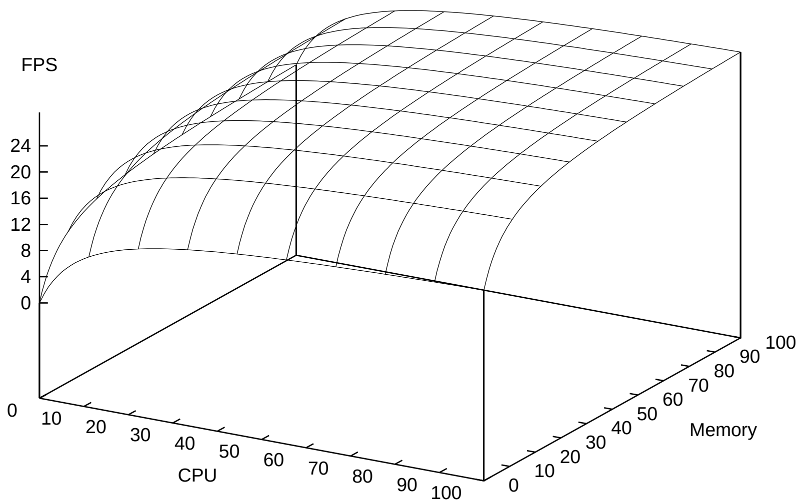
We refer to the quality of service characteristic function of a given process $p_i \in$ PROCESS as $q_i$. We can define $q_i$ as $q_i$: $\mathbb{Z}_{r_1.\text{levels}+1} \times \mathbb{Z}_{r_2.\text{levels}+1} \times \ldots \times \mathbb{Z}_{r_m.\text{levels}+1} \rightarrow \mathbb{R}$, with $m$ being the size of RESOURCE.

For example, suppose we have a video application process $p_i \in$ PROCESS under a fixed workload. This application is allocated two available resources, CPU and memory, with both resources having 100 levels of allocation. In this scenario, the video application would have a quality of service function that mapped allocations of these two resources to a quality of service metric. In this case, video frame rate, measurable in frames per second (FPS), would be an appropriate metric. One possible quality of service function $q_i \in$ QUALITYFUNCTION in this scenario is shown in Figure 3.1. It is important to note that this quality of service function is given only as an illustration—we still assume that we would know what this function is in advance.

### 3.1.6 Expectations

To capture user needs and to regulate application behaviour in the computing environment, quality of service requirements must also be formalized. Expectations are used to specify the behaviour required from application processes, relative to the quality of service characteristic functions defined for the processes.

**Figure 3.1** Example Quality of Service Function




Let EXPECTATION be the set of quality of service expectation functions. For each process $p_i \in$ PROCESS, the quality of service expectation function from EXPECTATION that applies to $p_i$ is denoted as $e_i$.

Each expectation $e_i$ is a function that defines the conditions under which the requirement has been met or violated. These conditions include quality of service metric values, as well as the values of attributes of the various objects in the computing environment. The expectation function $e_i$ can be defined formally as $e_i$: $\mathbb{R} \rightarrow \{-1, 0, 1\}$. In other words, $e_i$ takes a quality of service metric value as a parameter and returns 0 if the expectation $e_i$ is satisfied, -1 if the metric is below requirements, and 1 if the metric is above requirements.

Continuing the example from the previous section, suppose we want to specify that the video application process $p_i \in$ PROCESS display video at a frame rate of 27 frames per second, plus or minus 3 frames per second, giving an acceptable range between 24 and 30 frames per second inclusive. An expectation, $e_i$, capturing this requirement would be defined as follows:

$$e_i(m) : \begin{cases} -1 & \text{if } (m < 24.0); \\ 0 & \text{if } ((m \geq 24.0) \wedge (m \leq 30.0)); \\ 1 & \text{if } (m > 30.0). \end{cases}$$

Consequently, to determine if a given resource allocation $a$ meets this expectation, we would verify whether $e_i(q_i(a))$ is zero or not.

Please note that other expectations can be described using this mechanism, such as availability expectations, security expectations, and so on. However, for this work, we are primarily concerned with those quality expectations focussing on performance.

## 3.2 Problem Definition

We can formulate the quality of service resource management problem using the notation discussed in the previous section as follows:

### Given:

1. $\{ p_1, p_2, \ldots, p_n\}$ such that $p_i \in$ Process
2. $\{ e_1, e_2, \ldots, e_n\}$ such that $e_i \in$ Expectation
3. Resource

### Find:

$\{ a_1, a_2, \ldots, a_n\}$ such that $a_i \in$ Allocation $\wedge\ e_i(q_i(a_i)) = 0$

We cannot assume that any $q_i$ is known in advance. The only piece of information available to help solve this problem, at this point, is that each quality of service characteristic function $q_i$ must meet the following constraints:

1. $((q_i(< x_1, x_2, \ldots, x_m >) > q_i(< y_1, y_2, \ldots, y_m >)) \rightarrow \exists\ (1 \leq a \leq m) \mid x_a > y_a))$

   This states that if one resource allocation $x$ provides higher quality than another allocation $y$, that more of at least one resource was allocated in $x$ than in $y$.

2. $(\exists\ (1 \leq a \leq m) \mid (x_a > y_a \wedge \forall\ (1 \leq b \leq m, a \neq b), x_b \geq y_b) \rightarrow$
   $(q_i(< x_1, x_2, \ldots, x_m >) \geq q_i(< y_1, y_2, \ldots, y_m >))$

   This states that if at least one resource was allocated higher in an allocation $x$ compared to an allocation $y$, and that no resources were allocated less in $x$ than $y$, then the quality derived when $x$ is allocated must be at least as high or higher than the quality derived when $y$ is allocated instead.

Together, these ordering constraints dictate that as resources are allocated to a process, quality does not decrease and that as resources are taken away from a process, quality does not increase. This, of course, assumes that the quality delivered by one process is not affected by resource allocations made to other processes; as we will see later in this work, this can be accomplished using the appropriate resource allocation mechanisms.

## 3.3 Problem Analysis

With the formal problem definition and notation from the previous sections, we can begin analysing this problem. Before we begin, however, we need to devise a strategy to do so.

### 3.3.1 Analysis Strategy

As was mentioned earlier, the problem being solved is an intrinsically difficult one. Indeed, there are several factors that affect the solution, as well as other considerations that need to be taken into account. All of these together make directly tackling the problem in its entirety all at once an unfavourable approach.

Instead, a better approach is to examine a restricted version of the problem with constraints that make it easier to solve than the general problem given in Section 3.2. These constraints can then be relaxed one at a time, until the general problem is finally addressed. This leads to the following plan of attack:

1. The simplest and most restricted version of the problem would be to solve it when there is only a single resource, with only one application process executing and consuming the resource. We would further assume that application and user needs are static. While this is presents a highly unrealistic scenario, it allows us to begin with a very simple situation.

2. Next, we take step 1, remove the single resource constraint, and examine the problem when multiple resources must be considered.

3. Next, we take step 2, remove the single process constraint, and examine the problem with multiple application processes being managed.

4. Finally, we take step 3, remove the static constraints, and examine the problem under a dynamic environment.

In the next several sections, this strategy for analysis is carried out.

### 3.3.2 Single Process, Single Resource With Static Conditions

In this, the simplest version of the problem, we have a single process consuming only a single resource. The resource needs of the application and environment conditions are constant. Placing these constraints on the formal problem statement from Section 3.2, we have the problem:

**Given:**

1. $p_i \in$ PROCESS
2. $e_i \in$ EXPECTATION
3. RESOURCE$= \{r\}$

**Find:**

$a_i \in$ ALLOCATION $\wedge\ e_i(q_i(a_i)) = 0$

**Theorem 1** *The single process, single resource quality of service resource management problem under static conditions is in P.*

**Proof:** This problem amounts to searching through all possible resource allocations to the single process to locate one that results in acceptable quality of service. Since the resource being allocated, $r$, has a finite number of possible levels of allocation, $r$.levels, there are $r$.levels allocations to search through and the size of the search space input to this problem is $n = r$.levels. A brute force solution trying every possible resource allocation in its search will determine the resources required to deliver acceptable quality in at most $r$.levels allocations. Consequently, this simple approach takes O($n$) time. Thus, this problem is in P. □

### 3.3.3 Single Process, Multiple Resources With Static Conditions

While the only constraint that has been relaxed in this version of the problem is the number of resources to allocate, this problem is significantly more complex. With these constraints on the formal problem statement from Section 3.2, we have the problem:

**Given:**

1. $p_i \in$ PROCESS

2. $e_i \in$ Expectation
3. Resource

**Find:**

$a_i \in$ Allocation $\wedge$ $e_i(q_i(a_i)) = 0$

**Theorem 2** *The single process, multiple resource quality of service resource management problem under static conditions is in NP.*

**Proof:** To prove this, we must demonstrate that a nondeterministic algorithm exists to solve this problem in polynomial time. Suppose we begin with the minimum resource allocation possible, $< 0, 0, \ldots, 0 >$. At this point, we choose which resource whose allocation should be incremented by 1 to form our next allocation—this decision is the nondeterministic choice in our nondeterministic algorithm [120]. We repeatedly make this nondeterministic choice, always incrementing a single resource at a time, until we either find our target allocation satisfying quality requirements, or until we hit the maximum resource allocation possible, $< l, l, \ldots, l >$. Either way, the algorithm terminates. To reach the maximum allocation $< l, l, \ldots, l >$ starting from $< 0, 0, \ldots, 0 >$ would require $n \times l$ increment operations. Consequently, this nondeterministic algorithm could find its solution in polynomial time, and thus this problem is in NP. □

**Theorem 3** *The single process, multiple resource quality of service resource management problem under static conditions is NP-complete.*

**Proof:** Having established that this problem is in NP, we now must demonstrate that the problem is NP-hard. Finding $a_i \in$ Allocation such that $e_i(q_i(a_i)) = 0$ amounts to finding a $t$ such that $q_i(a_i) = t$ and $e_i(t) = 0$. With $n$ resources in Resource, this is equivalent to stating $q_i(a_{i,1}, a_{i,2}, \ldots, a_{i,n}) = t$, where each $a_{i,j}$ is a component of the resource allocation $a_i$. Consequently, a solution to this equation needs to be derived. If each $a_{i,j} \in \mathbb{R}$, doing so would be trivial. However, since each $a_{i,j}$ is restricted to be in $\mathbb{Z}_{l+1}$, the problem is much more complicated.

To show that solving such an equation is NP-complete, we prove that doing so is NP-complete when the quality of service function $q_i$ is a simple linear combination. Since the problem is NP-complete for that class of functions, it must be NP-complete in general; otherwise, the general solution could be used instead to solve the restricted problem, and the restricted problem would not be NP-complete.

To examine the problem with simple linear combinations as quality of service characteristic functions, we begin by examining the subset sum problem [37], which was proven to be NP-complete through a reduction from the vertex cover problem in [37] and through a reduction from integer linear programming in [170]. In this problem, we are given a set $S \subset \mathbb{N}$ and a target $t \in \mathbb{N}$, and we must determine whether there is a subset $S' \subseteq S$ whose elements sum to $t$.

This problem can be mapped into our problem of deriving a solution to the quality of service equation given earlier. As noted in [103] and elsewhere, the subset sum problem is, in fact, equivalent to determining if a solution exists to a very simple linear equation where we are given $n$ constants $s_i \in S$ and $t$ as above, and must solve:

$$\sum_{i=1}^{n} b_i s_i = t \qquad b_i \in \{0, 1\}$$

If a solution exists to this equation, then the subset $S'$ must also exist, and can be determined as $S' = \{s_i \mid b_i = 1\}$. In other words, we use the $b_i$ values as booleans to indicate membership in the subset summed to the given target. Thus, we have shown that given $n$ constants $c_1, c_2, \ldots, c_n$ and constant $q$, solving the following linear equation is NP-complete:

$$c_1 x_1 + c_2 x_2 + \ldots c_n x_n = q \qquad q \in \mathbb{N}, c_i \in \mathbb{N}, x_i \in \{0, 1\}$$

Given the above NP-complete linear equation problem, [170] demonstrates that this problem maps into a more general form that is also NP-complete, where we are given $n$ constants $c_1, c_2, \ldots, c_n$ and constant $q$, and must solve:

$$c_1 x_1 + c_2 x_2 + \ldots c_n x_n = q \qquad q \in \mathbb{Q}, c_i \in \mathbb{Q}, x_i \in \mathbb{N}.$$

Using the technique in Corollary 18.1b in [170], we can take this problem with non-negative integer variables, and map it into a problem in a $l$-ary number system where we are given $m$ constants $d_1, d_2, \ldots, d_m$, and constant $t$, and must solve the equation:

$$d_1 y_1 + d_2 y_2 + \ldots d_m y_m = t \qquad t \in \mathbb{Q}, d_i \in \mathbb{Q}, y_i \in \mathbb{Z}_{l+1}$$

This problem, then, is NP-complete.

This last equation, however, is a simple linear combination that is a valid quality of service function. If we defined $q_i$ as:

$$q_i(a_{i,1}, a_{i,2}, \ldots, a_{i,m}) = d_1 a_{i,1} + d_2 a_{i,2} + \ldots d_m a_{i,m}$$

then attempting to solve:

$$q_i(a_{i,1}, a_{i,2}, \ldots, a_{i,m}) = t$$

is equivalent to our previous NP-complete problem. Consequently, our mapping is complete, and solving $q_i$ for a given value $t$ is NP-hard when $q_i$ is a simple linear combination. Deriving a solution to the more general problem when $q_i$ is not necessarily a linear combination must also be NP-hard. Otherwise, the more general solution could be used to find a solution to the case where $q_i$ is a simple linear function, rendering it not NP-hard as well. Since we have just shown that an NP-complete problem can be easily reduced to the general resource management problem with a single process and multiple resources under static conditions, and Theorem 2 showed that this problem is in NP, we conclude that this problem must be NP-complete. □

### 3.3.4 Multiple Processes, Single Resource With Static Conditions

Given the above extension to the problem, supporting the allocation of multiple resources, is NP-complete, one would expect that supporting multiple processes is also an NP-complete problem. It turns out, however, that this version of the problem has polynomial time solutions. With the constraints from this variation on the formal problem statement from Section 3.2, we are left with the problem:

**Given:**

1. $\{ p_1, p_2, \ldots, p_n\}$ such that $p_i \in$ PROCESS
2. $\{ e_1, e_2, \ldots, e_n\}$ such that $e_i \in$ EXPECTATION
3. RESOURCE$= \{r\}$

**Find:**

$\{ a_1, a_2, \ldots, a_n\}$ such that $a_i \in$ ALLOCATION $\wedge\ e_i(q_i(a_i)) = 0$

**Theorem 4** *The multiple process, single resource quality of service resource management problem under static conditions is in P.*

**Proof:** In this situation, we would have to satisfy the following quality conditions for the $n$ processes being managed: $q_1(a_1) = t_1$, $q_2(a_2) = t_2$, ..., $q_n(a_n) = t_n$. Letting $l$ be the number of levels of resource allocation, $r$.levels, this problem amounts to finding a distribution of the $l$ units of input resource $r$ to the $n$ input processes such that the quality requirements of each are met. Since the number of possible distributions can be large, we must take a clever approach to limit the number of resource allocations required.

So, we methodically examine each of the $n$ processes in sequence, and find the level of allocation of the single available resource $r$ required to satisfy each process's quality constraints in turn. To do this, we start with the smallest possible allocation, and increment it by the allocation granularity for the resource, $r$.granularity, until we either satisfy quality, or exhaust the supply of the resource (in which case a solution does not exist). Once quality is satisfied, we move on to the next process and repeat. This approach will find the smallest solution to each quality equation and will find a solution to all equations, if one exists. If a solution does not exist, this approach will recognize this upon exhausting the supply of the resource. This approach is also quite efficient, finding a solution satisfying all processes in at most $l$ steps, as we are making at most $l$ allocations along the way to all of the processes. (If we reach $l$ allocations, we have exhausted the supply of the resource, and terminate regardless.) Consequently, this simple algorithm can solve this problem in O($l$) steps, and this problem is in P. □

### 3.3.5 Multiple Processes, Multiple Resources With Static Conditions

With multiple resources allocated to multiple processes, we are now dealing with a very realistic version of the problem. In essence, this problem is what most computing environments face, except that resources are managed individually, and usually in an ad hoc manner. Coordinating resource management to satisfy quality of service needs is significantly more complex.

With the constraints from this variation on the formal problem statement from Section 3.2, we have the original problem form again:

**Given:**

1. $\{ p_1, p_2, \ldots, p_n\}$ such that $p_i \in$ Process
2. $\{ e_1, e_2, \ldots, e_n\}$ such that $e_i \in$ Expectation
3. Resource

**Find:**

$\{ a_1, a_2, \ldots, a_n\}$ such that $a_i \in$ Allocation $\wedge\ e_i(q_i(a_i)) = 0$

**Theorem 5** *The multiple process, multiple resource quality of service resource management problem under static conditions is in NP.*

**Proof:** Once again, to prove this, we must demonstrate that a nondeterministic algorithm exists to solve this problem in polynomial time. To do so, we call upon the nondeterministic algorithm shown in Theorem 2 to be in NP. This algorithm will determine the resource allocation required to satisfy the expectations of a single process when there are multiple resources to allocate.

To use this algorithm to handle multiple processes, we apply the following procedure. We take the first process $p_1$ from our set of processes, and use the nondeterministic algorithm from Theorem 2 to find the allocation $a_1$ in polynomial time. We then subtract $a_1$ from the maximum allocation $< l, l, \ldots, l >$ to produce the maximum allocation to be used when applying the nondeterministic algorithm to the next process. We then repeat this process for $p_2, p_3, \ldots p_n$, each time reducing the maximum allocation available for the next process by the allocation used to satisfy the current process under examination.

Carrying out this procedure, we will find the resource allocations used to satisfy each process in $n$ invocations of the nondeterministic algorithm. If the nondeterministic algorithm cannot satisfy expectation $e_i$ for process $p_i$ because the maximum allocation possible at that point is too low, then we can conclude that it is not possible to satisfy all processes at once. Since each invocation of the nondeterministic algorithm requires polynomial time, and the approach taken here requires at most $n$ invocations of this algorithm, this procedure can also be completed nondeterministically in polynomial time. Consequently, this instance of the quality of service resource management problem is also in NP. □

**Theorem 6** *The multiple process, multiple resource quality of service resource management problem under static conditions is NP-complete.*

**Proof:** In this problem, we have a series of quality of service equations that we need to solve, each with multiple bounded integral variables. With $n$ processes and $m$ resources to allocate, we must solve:

$$\begin{aligned} q_1(a_1) &= t_1 \\ q_2(a_2) &= t_2 \\ &\cdots \\ q_n(a_n) &= t_n \end{aligned}$$

This is equivalent to the following:

$$\begin{aligned}
q_1(a_{1,1}, a_{1,2}, \ldots, a_{1,m}) &= t_1 \\
q_2(a_{2,1}, a_{2,2}, \ldots, a_{2,m}) &= t_2 \\
&\ldots \\
q_n(a_{n,1}, a_{n,2}, \ldots, a_{n,m}) &= t_n
\end{aligned}$$

In this problem, we must also ensure that the total amount of each resource allocated is less than the capacity of that resource. In other words, we need to ensure for each resource $r_j$:

$$\sum_{i=1}^{n} a_{i,j} \leq r_j.\text{levels}$$

In essence, we must solve $n$ instances of the already proven NP-complete problem from Section 3.3.3. In fact, this current problem is a generalization of that previous problem, where $n = 1$. The restriction that the sum of each resource allocation is less than each resource's capacity is easily met by the single process problem, since each allocation is between 0 and $r_j$.levels by definition of $q_i$. Consequently, we can map our earlier NP-complete problem into this problem quite easily, making this instance of the problem NP-hard. Since the previous theorem proved that this problem is in NP, it must be NP-complete.

□

It is important to note at this point that the distribution of resources across hosts does not impact the NP-completeness of the problem. Instead, it is the behaviour of the processes consuming these resources that can affect NP-completeness. Suppose, for example, that our $m$ resources were spread across $m$ hosts, with one resource per host. If our $n$ processes did not depend upon one another, and operated in isolation, we would have a collection of single resource problems with either single or multiple processes in each. In such cases, Theorem 1 and Theorem 4 would apply, and we would have a collection of at most $m$ problems that are in P; consequently, the larger problem would be in P as well.

If we were to suppose, however, that our $n$ processes did depend upon one another (for example, because they communicate with one another), the situation would be different. In this case, the quality of service delivered by one process is directly affected by the resource allocations given to the other processes. If insufficient resources are allocated to a process on another host, it could have a detrimental impact on the quality delivered by another process on a different host that depends on it. Likewise, too many resources allocated on one host could cause other processes on other hosts to outperform quality targets. Because the quality delivered by a process depends on the resources allocated to the other processes that it depends upon, all of these resources must be managed in a consistent and coordinated fashion, even when they are allocated to processes spread across multiple hosts. Consequently, our multiple $m$ resources must be managed as if they were all on a single host, and we have a situation equivalent to multiple processes using multiple resources. In such a case, Theorem 6 would apply, and the problem would be NP-complete.

## 3.4 Quality of Service Resource Management Under Dynamic Conditions

Most realistic computing environments are dynamic. User requirements, application resource needs, and the computing environment itself all change over time. Unfortunately, this dynamic nature can make ensuring quality of service quite difficult. In this section, we examine how the dynamic nature of computing environments affects the quality of service resource management problem.

### 3.4.1 Dynamic User Requirements

A change in user requirements is a problem that is relatively easy to deal with. The life time of an application can be broken into several periods, each with constant user requirements. Each of these periods can be thought of as a separate problem that is equivalent to one of the variations of the quality of service management problem given earlier in this chapter. Since these changes are driven by the user, this is a reasonable way of treating the problem since they are naturally aware of the change and can expect some period of readjustment to their new needs.

### 3.4.2 Dynamic Application Resource Needs

As mentioned in earlier chapters, it is rare for any application to consume a constant amount of resources throughout its life time. As a result, the amount of resources required to maintain a constant level of quality of service delivery changes over time—coping with this kind of change can be incredibly difficult.

If application needs change arbitrarily, it is possible for a solution resource allocation to never be determined. Every time a solution is about to be found, needs may change, and the problem needs to be solved again from scratch since the target being sought has changed as well. If this were to happen indefinitely, a solution might never be found, and quality of service is never met. On the other hand, if reasonable bounds are placed on the dynamic nature of the system, we can divide the problem into a sequence of several sub-problems, each with static conditions. Each such sub-problem would inherit the characteristics of one of the possible problem scenarios from the previous section, depending on the number of processes and resources involved. If multiple resources were involved, each sub-problem would be NP-complete. If only a single resource was involved, however, each sub-problem would be in P.

## 3.5 Summary of Results

In this chapter, we have examined many of the theoretical underpinnings of the quality of service resource management problem. We developed notation for working with this kind of problem, and provided a formal definition of this problem. An analysis of this problem yielded several interesting results:

- Most realistic quality of service management scenarios are NP-complete problems. Consequently, most likely algorithmic solutions do not exist that can efficiently solve these problems, and alternative techniques are necessary to assist in the process.
- It is wise to eliminate resources, hosts, and processes from quality of service management decisions as quickly as possible to reduce the computational needs of these decisions and improve their efficiency.
- The dynamic nature of modern computing environments makes quality of service resource management particularly difficult. Techniques are required to help overcome these difficulties.

Because of the complex nature of the quality of service resource management problem as proven in this chapter, there is no point in seeking an efficient algorithmic solution to directly solve this problem; one does not exist, unless P=NP. Instead, our findings indicate that we must investigate heuristic approaches to delivering acceptable quality of service. In the next chapter, a quality of service management architecture is introduced as a framework for doing just this. In the following chapter, several heuristic resource management algorithms are studied to be housed within this framework.

# Chapter 4

# Architectural Design

In the previous chapter, we showed that most realistic variations of the quality of service resource management problem are NP-complete. Since a direct algorithm solution to our problem is not feasible, we turn to heuristic techniques instead. With this in mind, in this chapter, we provide a management architecture for resource allocation to satisfy quality of service requirements that meets the needs identified in Section 2.3.

## 4.1 General Strategy

When one considers the dynamic approach to quality of service management discussed in Chapter 2 and shown in Figure 2.1, the basic management procedure is to iteratively allocate resources to application processes, measure the change in delivered quality as a result, and use these differences as feedback in determining adjustments. This is repeated until quality of service requirements are satisfied. An iterative approach to quality of service management makes sense, since attempting to move delivered quality to requirements in a single step is too expensive computationally, and moving forward in iterations allows management to cope with changes resulting from the dynamic nature of the problem.

The management system designed must have the following features:

- Specifying quality of service requirements.
- Collecting quality measurements and other information from applications.
- Determining whether quality of service requirements are met or not.
- Deciding which adjustments are needed to satisfy quality of service requirements.

- Making necessary adjustments to resource allocations or application behaviour.
- Determining what to do if resource demand exceeds resource supply.

Each of these management activities are realised as one or more components in our architecture. These components then work with another to provide our quality of service management strategy.

There are many general approaches to management that this architecture could take, each with their own strengths and weaknesses. In this work, a policy based approach to management is used, as this approach provides the best fit with our needs and our general management strategy. This is discussed in further detail in the next section.

## 4.2 Policy Based Management

Policy based management is an approach to management in which the requirements used to govern management decisions are logically separated from the underlying mechanisms used to actually carry out management activities. In this approach, the formalism used to capture these requirements is policies. In the literature, policies are generally defined as if-then rules: if the appropriate conditions are met, then a corresponding set of actions are carried out as a result [177]. In this work, we will adopt this definition of a policy as well.

A policy based approach to management is only natural since, as pointed out in [76], virtually all management relies on policies of some sort; whether formal or informal, written, or not. Formalizing the usage of policies makes sense to avoid the potential inconsistencies and problems in the ad hoc application of informal policies, as well as to automate the management process and address the complexity of the problem (as per the previous chapter). From an architectural point of view, the use of policies decouples the logic of what management is to accomplish from the mechanisms used to achieve these accomplishments. This separation allows a policy based management system to be more flexible and dynamic, since its policies are no longer hard-coded.

### 4.2.1 Types of Policies

In this work, we make use of two different kinds of policies to guide management decisions:

**Expectation Policies:** Expectation policies are used to state requirements of attributes of application behaviour; if the requirements are not being satisfied, the corresponding actions are taken. For example, an expectation policy for a video application could specify a condition that if the frame rate of delivered video falls below 24 frames per second, actions would be taken to notify the quality of service management system of the problem. These policies can be specified by application users, developers or administrators.

**Enforcement Policies:** Enforcement policies are used to state how to achieve quality of service requirements. For example, an enforcement policy for the same video application as above could state a condition that if several applications are executing at once with at least one application not satisfying its quality of service requirements, and there are insufficient resources to satisfy the requirements of all applications at the same time, then actions be taken to ensure that applications started first be favoured over those applications that started later. These policies are usually specified by system administrators of some kind, perhaps with input from application users and developers.

By supporting both kinds of policies, we can easily control the behaviour of both the applications being managed, as well as the quality of service management process itself.

### 4.2.2 Stages of Policy Based Management

As noted in [180], there are five stages to policy based management, once policies have been specified. These are as follows:

**Monitoring:** In this stage, information is collected from both the computing environment and the applications executing within it. This information includes application quality of service metrics, resource utilization information, and so on. For example, if we are interested in maintaining the quality of service requirements of a client-server video streaming application, we would be interested in monitoring the client video frame rate, client video jitter, server transmission rate, client and server resource utilization, and perhaps other information. This information is then used in the latter stages of policy based management.

**Detection:** Using information collected in the monitoring stage, expectation policies are checked to determine if the quality of service management system should be notified of a violation of quality of service requirements. For example, if the client video frame rate from the previous example falls below 24 frames per second, the quality of service requirement for the video frame rate policy for the application has been violated. Reports of such detected violations must be passed on and processed further to ensure that acceptable behaviour is restored.

**Location:** After violations of quality of service requirements have been detected, they need to be analysed and diagnosed to determine the cause and location of the root problem that induced the violation. This includes determining which computing environment components are at fault, if the application is distributed across several hosts in a network, as well as which resources on those components are causing the problems. For example, if the video client from the previous example is not delivering video at the expected rate, the problem could be caused by server congestion, saturation in the network, or insufficient resources on the client end-system. Locating the cause of the problem is a crucial step in determining what actions need to be taken.

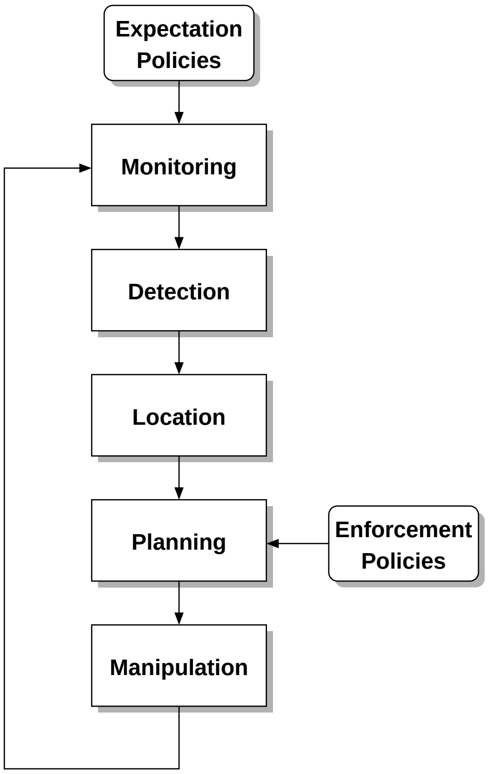
**Planning:** Based on location results, monitored information, and enforcement policies, the planning stage determines the set of corrective actions necessary to ensure that applications are once again compliant with their expectation policies. For example, if the end-system that the video client from the previous example is executing on proves to be the source of its poor performance, and further analysis determines that the client lacks CPU resources, planning will determine a new resource allocation (or possibly series of resource adjustments) to satisfy quality of service requirements. If there were insufficient resources for this planned allocation because other applications were consuming resources, the conditions for the example enforcement policy from Section 4.2.1 would be met. The planning stage, adhering to this policy, would determine which applications were started after the above video client, and steal away sufficient CPU resources.

Whenever possible, corrective actions will only entail adjusting resources to the application experiencing problems to restore quality of service to policy requirements. However, when demand for resources by applications outstrips supply, more drastic actions might be necessary, as dictated by the enforcement policies in place.

**Manipulation:** Finally, once the corrective actions necessary have been determined, the actions must be executed. This is done through changing resource allocations, updating application configuration, or communicating with application users to change the way they are using their applications. For example, the video client in the previous example will receive additional CPU cycles, as dictated by its action plan.

Policy based management is a cyclic process, with these steps repeated indefinitely, to continually manage an environment and its applications. This process is shown in Figure 4.1. Depending on the resource management algorithm in use, more or less emphasis may be placed on each stage of policy based management.

**Figure 4.1** Stages of Policy Based Management



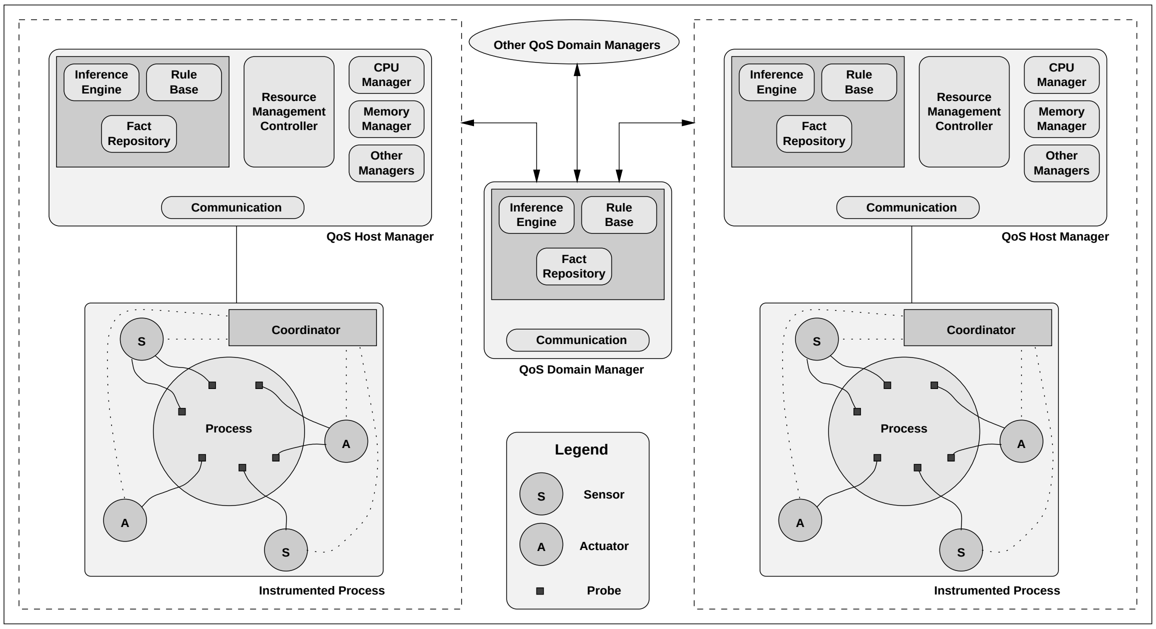

The stages of policy based management are very similar to the stages of dynamic quality of service management discussed in Section 2.1.2 and shown in Figure 2.1. Further, user quality of service requirements map nicely into expectation policies, and the dynamic nature of policy based management allows these requirements to change over time, as they often do. Consequently, policy based management appears to be a good fit to the quality of service resource management problem and addresses many of the issues raised as requirements in Section 2.3.

## 4.3 Architecture

Based on the policy based quality of service resource management process discussed in the previous section, in this section, we discuss a software architecture developed to support this process. This architecture is shown in Figure 4.2, and is discussed in detail in the sections below.

After examining the details of this architecture, we present several scenarios that describe how the various components of the architecture function and interact with each other.

**Figure 4.2** Quality of Service Resource Management Architecture




### 4.3.1 Application Instrumentation

Application instrumentation allows for optimal monitoring and control of application behaviour. This is crucial for the monitoring, violation detection, and manipulation stages of policy based management. Consequently, there needs to be a mechanism in place to facilitate management on a consistent and uniform basis. In this section, we examine the mechanism used, which is application instrumentation.

#### 4.3.1.1 Instrumented Process

An instrumented process is an application process with instrumentation code embedded at strategic locations to facilitate management. It is this instrumentation code that enables the monitoring and manipulation of the application process by the quality of service resource management system.

While some monitoring can be made by observing external application behaviour and rudimentary manipulations can be achieved through operating system interactions, work in this area has found that these approaches are limiting in both accuracy and flexibility in the kinds of metrics and control available. Since these limitations are unacceptable for the requirements of quality of service management as discussed in Section 2.3, instrumentation is the best option.

#### 4.3.1.2 Coordinator

The coordinator facilitates interactions between an instrumented process and the quality of service resource management system. Its role includes message routing, managing other instrumentation components, and handling initialization and termination activities within the process. All knowledge of the management system is confined to this component, effectively hiding it from the remaining instrumentation components.

The coordinator also oversees the policies associated with an application process. Upon initialization, the coordinator retrieves the set of expectation policies that apply to its process and uses these policies. Values of each attribute that is part of an expectation policy are assumed to be collected and maintained by sensors. Based on reports from sensors, the coordinator determines which quality of service requirements have been violated and forwards a notification to the appropriate management process in the quality of service resource management system (which should be specified in the action part of the corresponding expectation policy). Likewise, the coordinator can invoke actuators, components used to encapsulate functions that exert control over an instrumented process to change its operation or behaviour, to carry out management tasks as directed by the management system. As such, the coordinator plays a key role in both violation detection and, occasionally, manipulation stages of policy based quality of service management.

To continue the client-server video streaming example from Section 4.2, suppose we have an expectation policy dictating that the client should display streamed video to the user at a rate of 24 to 28 frames per second, and that if this requirement is not met, a notification of this violation is forwarded to the management process on the same host as the application process. The coordinator for the client would retrieve this policy on initialization and ensure that instrumentation components (sensors, in this case) are prepared to support this policy. When measured data indicates that the required conditions for the policy have been violated, the coordinator notifies the local quality of service resource management system process of the situation so that it may resolve the problem.

#### 4.3.1.3 Sensors

A sensor encapsulates an attribute characterizing some element of application behaviour, measured according to some application metric. Sensors get their input data from probes inserted at strategic points in the application code or by reading other sensors. A sensor probe is an interface between process code and sensors. Sensors provide their information to the coordinator in the form of periodic reports, alarm reports (when exceptional or critical circumstances arise), or in response to explicit requests. Sensors are dynamic components—they can be enabled or disabled, reporting intervals can be adjusted, thresholds can be modified, and sensor processing algorithms can be changed, all at run-time while the application process is executing.

Sensors play key roles in the monitoring and violation detection stages of policy based quality of service management; the information monitored from sensors is also critical to location and planning stages.

#### 4.3.1.4 Actuators

Actuators are used to encapsulate functions that can exert control over an instrumented process to change its operation or behaviour. Actuators, like sensors, function through the use of probes or other actuators, and possess similar dynamic properties.

Actuators are useful to support quality of service negotiation, adaptation, service differentiation, and a variety of other quality of service functions. As such, actuators are very useful to the manipulation stage of quality of service management.

Continuing our example from the previous sections, actuators may be put in place and initialized by the instrumentation coordinator to adjust video parameters in response to adaptation requests from the quality of service resource management system. For example, if insufficient resources exist to deliver quality above 24 frames per second, actuators may be used to reduce spatial resolution or choose a less computationally intense dithering method to free resources and bolster the delivered frame rate.

#### 4.3.1.5 Probes

A probe is an interface between process code and instrumentation sensors and actuators. Typically, probes take the form of method invocations or function calls to facilitate interactions with sensors and actuators. Probes allow sensors to collect attribute information and allow actuators to exert control over process behaviour. Each probe is specific to a particular sensor or actuator. Probes are the only instrumentation component that require modifications of the original process source code—all other components need only reside in the same address space as the process, and do not require such modifications.

Continuing the example from the previous sections, we would require probes for all of our sensors and actuators. The frame rate sensor would require a probe in the code stream processing video frames to update the current frame rate in the sensor. The adaptation actuators adjusting video or processing characteristics would need probes to manipulate the appropriate data structures in the process to change the process's behaviour upon request.

**Figure 4.3** Instrumentation Pseudo-code Example

**Input:** Video application process $p_i \in$ PROCESS
**Input:** $e_i \in$ EXPECTATION
**Output:** Stream of video frames $f_1, f_2, \ldots, f_n$

1: Perform initialization for $p$
2: Initialize coordinator $c$
3: $c$ retrieves $e_i$
4: $c$ initializes frame rate sensor $s$ with threshold from $e_i$
5: **while** $p_i$ not done **do**
6:    Retrieve next video frame $f_j$
7:    Decode and display $f_j$
8:    Execute $s \rightarrow probe$
9: **end while**

#### 4.3.1.6 Instrumentation Example

To illustrate the interactions of the various instrumentation components, consider Figure 4.3, which contains pseudo-code for the instrumented video application process used in examples throughout this chapter. This simple application initializes itself (step 1), and continuously retrieves frames of video from a source, decodes the frames, and displays them until the application is completed (steps 5–7 and 9).

Instrumentation has been added in Figure 4.3 to provide a coordinator $c$ and a frame rate sensor $s$ to measure the application's performance. After initializing the coordinator in step 2, the coordinator in step 3 retrieves all of the expectation policies that apply to the application, in this case only $e_i$, which is the same frame rate expectation policy used in earlier examples. The coordinator then in step 4 initializes the frame rate sensor so that the violation of its threshold indicates that quality of service requirements from $e_i$ were not met. After this, the only additional instrumentation required is to insert a probe for the frame rate sensor in step 8 to measure the frame rate delivered by the application; if measurements taken through this probe violate the frame rate sensor's threshold, the requirements of $e_i$ are not met, and the coordinator issues a notification to the local quality of service management process.

### 4.3.2 Quality of Service Managers

In this section, we examine the internals of the quality of service managers, detailing the components used to support the location, planning, and manipulation stages of policy based management. The architecture used for the quality of service managers is extended from the quality of service violation detection and location architecture used in [132], which was in turn based on the instrumentation infrastructure discussed above. From the architecture in [132], we use the rule based inferencing engine for location and planning phases of policy based quality of service management; this engine takes data from the detection mechanisms in instrumentation components, and drives the resource management mechanisms added to the architecture as part of this work, as discussed later in this section.

The basic idea behind this architecture is to provide a quality of service management process local to each host, responsible for the quality of service delivered by applications executing on the host. Application process coordinators communicate with these QoS Host Managers to report violations of quality of service requirements and to return monitored data upon request from its QoS Host Manager. Coordinators may also receive requests from its QoS Host Manager to manipulate application behaviour or tune instrumentation behaviour. When applications are distributed across multiple hosts in a computing environment, problems in satisfying quality of service requirements can span these hosts. To facilitate the quality of service management process in such cases, a separate domain manager process coordinates the activity of the individual local host managers. This QoS Domain Manager is responsible for overseeing the quality of service provided by a collection of hosts; it resides on one of these hosts. If problems span multiple such collections, additional coordination is necessary.

Having management processes on each host in this manner has several advantages over using a single centralized manager, despite incurring some resource overhead on each host. With a management process on every host, the quality of service management load is spread out, making the management system more scalable and more responsive. Further performance gains come from decreasing the physical distance and latency between the management system and managed process, and from being able to use more efficient local inter-process communication mechanisms, as opposed to remote communication mechanisms, which can be several orders of magnitude slower. Lastly, it is technically very difficult and incredibly insecure to allow remote applications to control local resource allocations. Consequently, this approach is the logical choice to keep with our requirements from Section 2.3.

#### 4.3.2.1 QoS Host Manager

A QoS Host Manager is located on every host in a distributed computing environment on which user applications needing quality of service execute. When an instrumented application process detects that the conditions of one of its expectation policies have been violated, a notification of this violation is sent to the local host manager. Upon receiving the notification, the host manager diagnoses the problem, perhaps using additional information queried from the host or the application process, to determine its cause. The host manager then determines the set of actions required to correct the problem, and then either makes the necessary resource adjustments or instructs the application process to adapt to the situation, if the problem was determined to be local. If the cause of the problem does not lie on the host, the host manager forwards the notification to the domain manager for its domain for further action. Thus, host managers are crucial in location, planning, and manipulation stages of policy based quality of service management.

To carry out these tasks, the host manager is composed of a variety of components, including a fact base, a rule base, an inference engine, a resource management controller, and a collection of resource managers. These components are discussed below in further detail.

#### Communication

The quality of service manager communication module is used to facilitate communication between the manager and other processes in the computing environment. The communication module for host managers allows it to communicate with both application processes and domain managers. Local inter-process communication mechanisms are used to communicate with processes since they reside on the same host as the host manager, providing efficient and reliable communication. Remote communication mechanisms are used to communicate with the domain manager since it may reside on any host in the domain.

#### Fact Base

The fact base is used to contain information known by a quality of service manager for diagnosis and planning purposes. Example facts can be found be consulting Appendix A.

The information contained in the fact base includes the following:

- Information on managed processes, including the process identifier, name, owner, start time, and so on. Domain managers also track the host that each process executes on.
- Data provided through violation reports, including the expectation policy under examination, the process where the violation originated from, and the events leading to the violation, especially relevant process attributes collected by sensors.
- Additional data queried from the operating system or other application processes as necessary.

Continuing our client-server video streaming example from previous sections, data on the application will be asserted into the fact base upon initialization, and on every violation reported. On initialization, the coordinator for each process registers the process, providing information such as the process identifier, name, owner, starting time, configuration, and so on, which is stored in the fact base. If the client's delivered frame rate falls below the expected 24 frames per second, or rises above 28 frames per second, the violation notification sent from the client process's coordinator to the local QoS Host Manager would result in the insertion of the following facts in the fact base:

- Frame rate violation notification.
- Time of the violation.
- Process identifier of the video client as the process in violation (can be matched with registration information).
- The target frame rate and actual delivered frame rate.
- Length of incoming video stream buffer for diagnosis purposes.

### Rule Base

The rule base contains a set of rules used in diagnosing and planning to determine corrective actions to quality of service problems. Example rules can be found be consulting Appendix A.

Each rule contains a set of conditions on when to apply the rule (which facts must be present) and the actions to take when the rule is applied. Rules are developed with some type of expert knowledge, such as application design, network or system architecture, operating system design, administrative needs, and so on. Policies required by a QoS Host or Domain Manager are mapped to one or more rules loaded into the rule base.

For example, we continue the example scenario from the previous section. Rules pertaining to this application in the rule base could include the following:

- If the frame rate is too high, then the process should be given fewer resources, as determined by the resource management controller.
- If the frame rate is too low, and the client incoming buffer size is above some threshold (implying that the process is not able to handle incoming frames quickly enough), then the process should be given additional resources, as determined by the resource management controller.
- If the frame rate is too low, and the client incoming buffer size is below some other threshold (implying that the process is more than able to handle incoming frames

quickly enough), then a notification should be sent to the domain manager, as the problem likely lies in the network or the server system delaying frames to the client. Additional rules for the domain manager would help locate the problem and formulate corrective actions.

For further details of this component, please refer to [132].

### Inference Engine

Inferencing is a recursive process whereby the inference engine determines the set of rules applicable based on the facts in the fact base, and applies them to formulate new facts for the fact base or determine corrective actions. The inference engine continues to apply rules as long as there are rules that are applicable. When done, a set of corrective actions has been accumulated and is ready to fire. The manipulations required are forwarded to the resource management controller, if resource adjustments are necessary, or to application processes, if other adaptations are deemed necessary.

The inference engine can use a variety of approaches to its inferencing. These include forward chaining (where the engine starts with known facts and works towards a solution), reverse chaining (where the engine starts with possible solutions and finds supporting facts), or relatively simple lookups. Since [132] took the approach of forward chaining, that approach will be used here.

For further details of this component, please refer to [132].

### Resource Management Controller

The resource management controller module oversees quality of service resource management for the local host. It makes general resource management decisions, leaving the details to individual resource managers.

The resource management controller basically encapsulates resource management algorithms. The controller can have multiple algorithms at its disposal. In Chapter 5 of this work, we will present three families of algorithms that to be used in the controller, each with their own characteristics.

Based on facts in the fact base, rules in the rule base, and inference engine input, the resource management controller uses the algorithms at its disposal to determine which resource or resources should be adjusted and by how much the resource allocations should be adjusted. Timing and synchronization issues are also handled by the resource management controller. Actual changes are left to individual resource managers, thereby hiding the resource management controller from specifics of the underlying resources.

The resource management controller module contains some code, but also depends on policies, making use of the fact base, rule base, and inference engine. This enables several possible scenarios for resource management:

- Code-only algorithms—relatively static algorithms that are composed only of code. As such, they can be made quite efficient, but cannot be changed or tailored easily. They can only be added by rebuilding and restarting host managers.
- Policy-only algorithms—dynamic algorithms that are composed only of policies. While not as efficient as code-only algorithms, as additional inferencing is necessary, they can be changed and tailored to situations quite easily, and can be added at run-time.
- Policy-code algorithms—algorithms that are composed of both code and policies. They have some of the efficiency gains provided by code-only algorithms, and also have the ability to change or tailor as policy-only algorithms do. While they can only be added by rebuilding and restarting host managers, they can be modified at run-time.
- It is possible to create new management policies and algorithms at run-time to cope with new quality of service management problems, using policy-only algorithms, and later improve the efficiency of the algorithms by substituting in a code-only or policy-code algorithm when it is convenient to do so.

This allows for a good combination of flexible and dynamic behaviour from policies, as well as speed and efficiency from native code.

#### Resource Managers

Resource managers are used to provide control over the management of local resources. This includes the allocation, scheduling, and accounting for these resources. Resource managers typically manage an individual resource, but they may also manage multiple resources. Managers encapsulate all of the details of working with the actual resources; consequently, the resource management controller does not need to concern itself with these details. This split of concern makes it easy to develop resource management algorithms independently of the mechanisms used to allocate them, and also allows new resource management mechanisms to be developed and put in place without having to substantially re-engineer the algorithmic logic.

Resource managers can be generic or application specific. Generic resource managers can manage computing resources for any application, including CPU, network, disk, memory, and other resources. Application specific resource managers are geared towards tuning resource management for particular applications. For example, an application specific resource manager for a database such as IBM's DB/2 could tune caching parameters or buffer pools, which are memory resources critical to database performance [122].

#### 4.3.2.2 QoS Domain Manager

Hosts in a distributed computing environment are often grouped into collections, that are referred to as domains. Domains are arbitrary groupings, but are usually based on network structure (for example, every sub-network is a domain), administrative control (for example, each department in an organization is a domain), or some other logical distinction. A QoS Domain Manager oversees each such domain, and is responsible for coordinating the solution of quality of service problems spanning multiple hosts in its domain. Typically, a QoS Domain Manager resides on one of the hosts in its domain.

When a host manager believes that the cause of a violation of a quality of service requirement is not local, it communicates with the domain manager for its domain to assist in the process. The domain manager analyses the information it receives to diagnose the situation. If necessary, it can then request additional information from other host managers, which in turn collect it from their host or application processes accordingly. When diagnosis is complete, the domain manager constructs an action plan to resolve the problem. Unlike host managers, a domain manager only performs these location and planning elements of policy based quality of service management, and does not directly perform corrective manipulations. Instead, it instructs the appropriate host managers to carry out the manipulations locally. Consequently, domain managers do not have the resource manipulation components that host managers have, but rather have only policy, diagnosis, and planning components. Domain managers make use of specialized rules in their rule bases for solving quality of service problems spanning multiple hosts in their domains. Examples of such rules are as follows:

- If the client-side host manager reports a quality of service requirement violation, retrieve the server's video transmission buffer length by querying the server-side host manager, which in turn obtains the information from the server process.
- This would result in the checking of another rule: If the client-side host manager has reported a problem and the server transmission buffer length is above some threshold (implying that the network is congested as the server has frames to send and the

client has room to receive), then additional network resources are allocated to the application by instructing the host managers to act accordingly. Otherwise, if the server transmission buffer length is below some threshold (implying the server is not able to transmit frames quickly enough) then the server-side host manager is directed to allocate the server additional resources, again as determined by the local resource management controller.

In addition to having only a single domain manager, it is possible for there to be multiple domain managers, either in a single domain or in multiple domains. In a single domain, a domain manager may be replicated for fault tolerance and performance reasons, with host managers balancing workload among the domain managers. In multiple domains, each domain will have its own domain manager. When quality of service problems span these domains, the domain managers will need to communicate with each other to resolve the problem. This could be done through direct peer-to-peer communication between the managers, or through a hierarchy of domain managers, each overseeing collections of domains. The former would likely perform better with fewer domains, while the latter would be more scalable. Since such issues are left as future work for [132], they will not be examined further here, as a single domain provides sufficiently complex resource management issues for this work.

## 4.4 Bringing it All Together

To give an impression of how all the different architectural components from the previous sections function together, we provide in this section several scenarios showing these components interacting to manage and maintain quality of service. These scenarios include the initialization of the management system, the initialization of an application, the occurrence of a quality of service problem, and the termination of an application.

### 4.4.1 Management System Initialization

As soon as the computing environment is started, the quality of service resource management system is initialized. This must be done to ensure that the management system is available before user applications are launched and require quality of service support.

First, domain managers start on their designated hosts. They start with an empty fact base, and retrieve rules and policies to construct the rule base. Policies can be retrieved directly from some kind of policy repository (such as a directory, database, or flat file), or using some sort of policy service. Next, host managers start on each host in the computing environment. They likewise begin with an empty fact base, and retrieve rules and policies to build their initial rule base. Host managers also locate their domain manager (or domain managers, if replication is used) for ready access when it becomes necessary.

### 4.4.2 Application Initialization

When an application process is started by the user, the first thing the process does is initialize its instrumentation coordinator. This coordinator registers the process with the local QoS Host Manager; providing it with the process's identifier, name, owner, start time, application membership, and so on. This registration is stored as facts in its fact base and is then passed on to the QoS Domain Manager by the host manager. The domain manager likewise stores the registration in its fact base.

After registration, the instrumentation coordinator retrieves the set of policies that apply to the application process. The coordinator then enables instrumentation sensors and actuators necessary to support these policies, and initializes them accordingly. This includes setting sensor thresholds to check for policy violations, data collection intervals, reporting intervals, and so on.

At this point, the application process is ready to be managed by the quality of service resource management system. Regular application processing begins, with measured application behaviour checked against policies for quality of service violations.

### 4.4.3 Quality of Service Problem Handling

When the conditions for a quality of service expectation policy are violated for a given application process, the process is no longer performing as expected. Consequently, at least one sensor tracking the attributes of this policy will have its threshold violated, causing the problem to be initially detected. The sensor notifies the coordinator for the process of the threshold violation, providing it with details of the violation. The coordinator determines which policies have been violated and gathers any additional information needed from sensors, specified in the action part of the expectation policy, packaging this information into a notification. This violation notification is then sent to the local QoS Host Manager.

When the QoS Host Manager receives the notification, the facts surrounding the violation (contained within the notification) are asserted into the manager's fact base. The QoS Host Manager's inference engine is then called upon to apply its rule base to its fact base to determine the cause of the quality of service problem generating the violation, and to generate corrective actions to solve the problem.

If the problem is found to be local, either resource adjustments or application adjustments are necessary. If resource adjustments are necessary, the resource management controller is called upon to determine which resources to adjust, and by how much, according to the resource management algorithm used and enforcement policies in place. Appropriate individual resource managers are invoked to carry out the adjustments as necessary. If application adjustments are necessary, the QoS Host Manager informs the application process's coordinator of the adjustments to be made. The coordinator then invokes the corresponding actuators to make it so.

On the other hand, if the problem is thought to be remote, the violation notification is forwarded to the QoS Domain Manager. The QoS Domain Manager asserts the facts surrounding the violation into its fact base. By applying rules to these facts, the domain manager's inference engine determines if additional information is needed from a remote host for diagnosis purposes. If so, it contacts the remote QoS Host Manager to request the information. The remote manager then queries the coordinator of the appropriate process, which in turn queries the appropriate sensor for the information. The results trickle back in the opposite direction, where the QoS Domain Manager asserts them into its fact base.

With additional facts and, consequently, a more complete view of the situation, the QoS Domain Manager has its inference engine apply its rule base against its fact base to uncover the cause of the quality of service violation and to determine what corrective actions are necessary. When done, the QoS Domain Manager notifies the appropriate QoS Host Manager (or managers, as the case may be) of the corrective adjustments necessary. The appropriate resource or application adjustments are then carried out in a similar fashion as when the problem was determined to be purely local.

### 4.4.4 Application Termination

When an application process is shutdown by the user, normal application processing is first to stop. This is to ensure that an application is manageable throughout its entire life time. The instrumentation coordinator disables all of its other instrumentation components, including its sensors and actuators.

The application process coordinator then deregisters itself with the local QoS Host Manager. The manager retracts facts surrounding the process from its fact base, and passes the deregistration request to the QoS Domain Manager. The domain manager likewise retracts facts surrounding the process from its fact base.

At this point, the application process is no longer being managed. The application process's coordinator finally shuts down, and the process terminates.

## 4.5 Summary

In this chapter, we have presented an architecture for dealing with the quality of service resource management problem in a dynamic and flexible fashion. It provides a framework for supporting a variety of resource management algorithms to deliver acceptable quality with reasonable computational needs. In the next chapter, we will present and examine several such algorithms.

# Chapter 5

# Algorithm Development

The previous chapter provided an architectural framework for quality of service resource management. In this chapter, we build upon these results to develop and refine several algorithms that can be used within the resource management controller of this framework to deliver acceptable quality of service to application users. Since, as pointed out in Chapter 3, direct algorithmic solutions to the problem are not feasible, the resource management algorithms discussed in this chapter all make use of some kind of heuristic technique instead.

## 5.1 Overview

Before presenting resource management algorithms, we first describe the general algorithm structure and similarities across all algorithms, the selection of the various algorithms, and the criteria used to discuss these algorithms.

### 5.1.1 General Algorithm Structure

All resource management algorithms discussed in this chapter share some common structural elements, as they are all dynamic approaches to resource management, and they must all reside within the resource management controller in the architecture. All algorithms are also integrated approaches to resource management, in that they manage all computing resources at once, as opposed to more traditional approaches to resource management that manage each resource independently.

When one considers the monitoring and resource management for dynamic quality of service management, we have Algorithm 1. This algorithm contains steps to measure the appropriate application's delivered quality of service, compare this service level to the target in its expectation policy, and carry out resource management. If the delivered quality of service exceeds requirements by a large margin, the application could make do with a less generous resource allocation. On the other hand, if the delivered quality of service is below requirements, the application requires a better resource allocation, either through the allocation of additional resources, or a better distribution of previously allocated resources.

**Algorithm 1** General Resource Management

**Input:** $p_i \in$ PROCESS
**Input:** $e_i \in$ EXPECTATION
**Output:** $a_i \in$ ALLOCATION $\mid e_i(q_i(a_i)) = 0$

1: **while** $p_i$ not done **do**
2:    $q \leftarrow$ Measure $p_i$'s quality of service
3:    **if** $e_i(q_i(a_i)) \neq 0$ **then**
4:       Invoke resource management specific algorithm
5:    **end if**
6: **end while**

This general resource management algorithm is the starting point for the resource management algorithms presented later in this chapter. In these algorithms, algorithm-specific steps are called from step 4 from the general algorithm.

### 5.1.2 Selecting the Algorithms

As indicated in Section 4.3.2.1, each resource management algorithm must decide which resource or resources require their allocations to be adjusted, and the amount of adjustment required, so that quality of service requirements are satisfied for a given process. How these decisions are made and carried out is what distinguishes one algorithm from another.

One would expect, to a certain extent, a trade-off between the quality of resource management decisions, and the amount of time required to make the decisions. Higher quality decisions are more likely to improve quality of service better than poorer quality decisions. At the same time, however, higher quality decisions require additional analyses and tend to take longer and are more expensive than poorer quality decisions. Indeed, one could potentially make several poorer quality decisions in the time required to make a single high quality decision, and deliver better overall quality as a result. Which approach is preferable remains to be seen.

Consequently, for this work, we study three families of algorithms. The metric-heuristic approach is aimed at making higher quality decisions through run-time diagnostics (fault location) based on information obtained through the management architecture discussed in the previous chapter. On the other end of the spectrum is the round robin approach, which carries out no diagnostics or analyses, but rather attempts to make resource adjustments as quickly as possible. In the middle is the multidimensional search approach, which attempts to make reasonable resource management decisions quickly based on predetermined diagnoses (or diagnoses conducted in the first few applications of the algorithm). Details of these different algorithmic approaches are discussed in remaining sections of this chapter.

### 5.1.3 Discussing Algorithms

Several criteria are important in discussing algorithms presented in this work. This includes:

**Algorithmic Complexity:** Since Chapter 3 proved most realistic variations of the quality of service resource management problem to be NP-complete, algorithmic complexity is naturally of great concern. Since we are taking an iterative approach to resource management, as noted in Section 4.1, there are two notions of algorithm complexity that are important. The first is single iteration complexity, or the complexity of a single execution through the resource management specific algorithm as invoked from step 4 of Algorithm 1. The second is the number of iterations required for the resource management specific algorithm to determine the resource allocation necessary to meet quality requirements. Both are important factors in determining the overall complexity of an algorithm.

**Adaptability:** For this work, we term an algorithm to be adaptive if the algorithm can tune or alter its own behaviour over time to improve its decisions or efficiency. Adaptive algorithms, then, can be very powerful and very flexible. Because their own behaviour can be adjusted, they tend to be more resilient and able to operate independently without administrator intervention. Unfortunately, these algorithms can also be significantly more complex and can also incur higher overhead than non-adaptive algorithms.

**Algorithm State:** This issue deals with whether information must be kept from iteration to iteration of a resource management algorithm's execution. If none is required, the algorithm is stateless; on the other hand, if information is required, the algorithm is said to be state-based. Maintaining state information tends to add some complexity and overhead to an algorithm, particularly when a great deal of information must be maintained, or when several applications are being managed. In dynamic systems, state can grow stale over time and no longer accurately reflect the nature of the application or its environment, which can cause further complications. For these reasons, stateless algorithms have their advantages. However, if an algorithm is to be adaptive, it must maintain state information; otherwise, it would never be able to determine when it must make an adjustment to itself. Consequently, state-based algorithms can have their advantages as well.

**Prior Knowledge:** Some dynamic quality of service resource management algorithms can make use of prior knowledge of managed applications and their behaviour to make better management decisions. In such cases, it is generally best to limit the amount of information that is required or ensure that this information is generally available or inexpensive to acquire; otherwise, the algorithm may be difficult to use in practice. Depending on the algorithm and the information required, it may be possible to collect some of this information as the application executes, and make the appropriate adjustments and adaptations over time. If necessary, this collected information could also be stored for future executions.

While some insight can be gained from analysing the algorithms on paper using the above criteria, it is also important to examine the performance of the various algorithms under thorough and systematic experimentation. This will be conducted later in Chapter 7 of this work.

## 5.2 The Metric-Heuristic Algorithm

Quite often, considerable information useful to resource management decisions can be acquired from applications and their computing environments. Unfortunately, in most cases, this information largely goes unused, and resource management decisions are typically made in a "blind" fashion, without the benefit of any observable application or system behaviour. For this reason, the metric-heuristic resource management algorithm has been developed.

In the metric-heuristic algorithm, quality of service resource management decisions are based on metrics measured from the application and computing environment using the management architecture, and heuristics dictating how these metrics should be interpreted. This includes both which resource allocation should be adjusted, and by how much. Examples of useful metrics and heuristics include the following:

- For CPU resources, a possible metric would be the percentage of time an application is in a runnable state (it is able to run, but it is not currently on the processor). In this case, a heuristic would be that if this percentage is high enough, then the process requires additional CPU resources.

- For memory resources, one possible metric would be the page fault rate for a process. In this case, a heuristic would be that if this rate is too high, then the process requires a better allocation of physical memory resources. If one were also to observe a high global page fault rate for the system, a high I/O or swap load on the system, or a very low number of free virtual memory pages remaining on the system, there could be a general memory shortage for all application processes.

- For I/O resources, a possible metric would be the blocking time for a process. In this case, a heuristic would be that if the blocking time was too high, the process could use a better allocation of I/O resources. These can be generic resources such as buffer space, or specific resources depending on the type of I/O being done. For disks, this could mean preferential disk access or caching. For networks, this could mean improved bandwidth allocation, queue priority, and so on.

  Another possible metric for I/O resources, particularly for network communication, would be buffer filling patterns. Several heuristics would be applicable in this case. If an input buffer is filling, additional CPU cycles are needed to read the data; likewise, a draining output buffer might mean that additional CPU cycles are needed to keep up with the data transmission, especially if the input buffer on the other end of the communication is consistently empty. An empty input buffer on one end of a communication channel and a full output buffer on the other end likely means that additional network resources are needed to move the data around.

### 5.2.1 The Algorithm

The metric-heuristic resource management algorithm is presented in Algorithm 2.

**Algorithm 2** Metric-Heuristic Resource Management

**Input:** $p_i \in$ Process
**Input:** $e_i \in$ Expectation
**Input:** $q$ /* $p_i$'s measured quality of service */
**Output:** $a_i \in$ Allocation $\mid e_i(q_i(a_i)) = 0$

```
for all r ∈ Resource do
   m ← Measure r's resource metrics
end for
r' ← Apply heuristics h to m
for all r ∈ r' do
   Compute adjustment a according to heuristics h
   a_i ← a_i + a
end for
```

Essentially, the algorithm takes a diagnostic approach to resource management. The algorithm makes measurements about each resource that the application uses, and analyses them using a variety of heuristics. The result is the list of resources that should be adjusted, which is then processed and a new resource allocation is derived. This process repeats inside the general resource management algorithm in Algorithm 1 until delivered quality meets requirements.

### 5.2.2 Analysis of the Algorithm

In this section, we analyse the metric-heuristic algorithm. This is done in terms of both its complexity, and a variety of other issues, as discussed earlier.

#### 5.2.2.1 Single Iteration Complexity

There are three main sequential stages to the metric-heuristic resource management algorithm: metric collection (steps 1–3), heuristic analysis (step 4), and adjustment (steps 5–8). Each of these steps contributes to the overall complexity for each iteration. Over the entire metric-heuristic resource management algorithm, each iteration has a complexity that is the sum of each of the complexities resulting from each of the above stages.

It becomes difficult, however, to accurately pinpoint the complexity, as this complexity is tightly coupled to the implementation of the algorithm and computing environment. Other algorithmic decisions can affect the complexity as well. For example, suppose we allow each iteration of the algorithm to adjust multiple resources instead of a single resource; allowing multiple adjustments could make each iteration more costly, but has the potential to reduce the total number of iterations required in the process.

Suppose that there are $n$ resources to manage. In the metric collection stage of the algorithm, we must retrieve metric data for each of the $n$ resources. The cost to do so then is $\mathrm{O}(n{\cdot}m())$, where the function $m()$ yields the complexity of each metric collection step. In the heuristic analysis phase, we must at least examine the $n$ metrics in this stage to check each metric's value. The cost to do so is $\mathrm{O}(h(n))$, where the function $h()$ yields the complexity of the heuristic analysis of the metric data collected from all $n$ resources. In the adjustment phase, we perform calculations to determine the amount of resource adjustment required, and then make this adjustment. If a single resource is being adjusted, the cost to do so is $\mathrm{O}(c()+a())$, where the function $c()$ yields the complexity of adjustment calculation, and the function $a()$ yields the complexity of making the adjustment itself. If multiple resources were adjusted at a time, the cost to do so is $\mathrm{O}(n(c()+a()))$

instead. Consequently, the overall complexity of each iteration of the metric-heuristic algorithm would be $O(nm()+h(n)+c()+a())$, if a single resource was adjusted each iteration, or $O(nm()+h(n)+n(c()+a()))$, if multiple adjustments were allowed instead.

Because metric collection, heuristic analyses, adjustment calculations, and the adjustment process itself are outside of our control, we cannot be more precise, unless we place undue restrictions on any of these procedures. Since such restrictions would be undesirable for many reasons, we can go no further.

#### 5.2.2.2 Number of Iterations

The number of iterations required by the metric-heuristic resource management algorithm is largely determined by two factors. The first is the quality of the metrics collected and used in analyses. This quality is affected by the accuracy of the measurements, the frequency of measurements, and any filtering of the measurements. If metrics are collected improperly, all decisions based on them could be adversely impacted. The second factor is the quality of the heuristics used in analysing collected metrics and determining resource adjustments. If heuristics are incorrect, this again has ramifications on the decisions made by the algorithm. For example, if a high page fault rate of a process does not necessarily indicate a need for memory resources, the resulting allocation of additional memory resources may not improve the situation, but rather make it worse for other applications. With this in mind, can we still provide rough estimates of the number of iterations required?

If we suppose that both the data collected and the heuristics analysing this data are of high quality, then the algorithm can do quite well. If our heuristics always selected the correct resources to adjust on every iteration, a relatively simple heuristic in selecting adjustment amounts using a binary search approach would be quite effective. Such an approach, for example, would yield logarithmic performance. If we were to use an interpolation approach instead, where we base the amount of adjustment required on the current quality level and the distance between the current level and the desired level, we could instead achieve log log performance. Since both of these heuristics are rather simple, we could conceivable obtain better performance from better heuristics.

On the other end of the spectrum, however, if data collected or heuristics were not good at all, performance could be horrendous. If we frequently chose the incorrect resource to adjust, or adjusted it by the incorrect amount, it is possible for the algorithm to never find the resource allocation required to meet quality of service requirements. In such cases, there is no guarantee that a solution would be found. At the very least, with poor metrics and heuristics, several more iterations are required to correct prior mistakes and to move towards the desired level of quality.

It is interesting to note that, in this algorithm, there is a trade-off between the number of iterations required, and the complexity of each iteration. Decisions to simplify each iteration can yield reduced time requirements for each iteration, but require many more iterations to achieve good results. Likewise, collecting additional metric data and performing more complex heuristic analysis would result in better choices each iteration, and fewer iterations in the long run. However, this would come at a cost of more expensive iterations each time.

#### 5.2.2.3 Further Discussion

Because the metric-heuristic algorithm collects the data required for resource management decisions at run-time, this approach requires no prior knowledge on the applications in use or their computing environment. Since these metrics are collected freshly on each iteration of the algorithm, the algorithm does not need to maintain any state. The information that is required is gathered on an as-needed basis, so nothing needs to be kept from iteration to iteration. Since resource adjustments only occur when quality does not meet requirements, adjustments will cease once acceptable quality is achieved. If disturbances occur in the dynamic environment, this algorithm will be able to collect fresh metrics to analyse to determine a new resource allocation under the new conditions.

On the other hand, since information must be collected at each iteration of the algorithm, this algorithm imposes more overhead than other approaches. If implemented carefully, this overhead can be minimized by collecting metrics in parallel or in the background between iterations of the algorithm. Heuristic analysis has the potential to impact overhead as well, depending on the heuristics used.

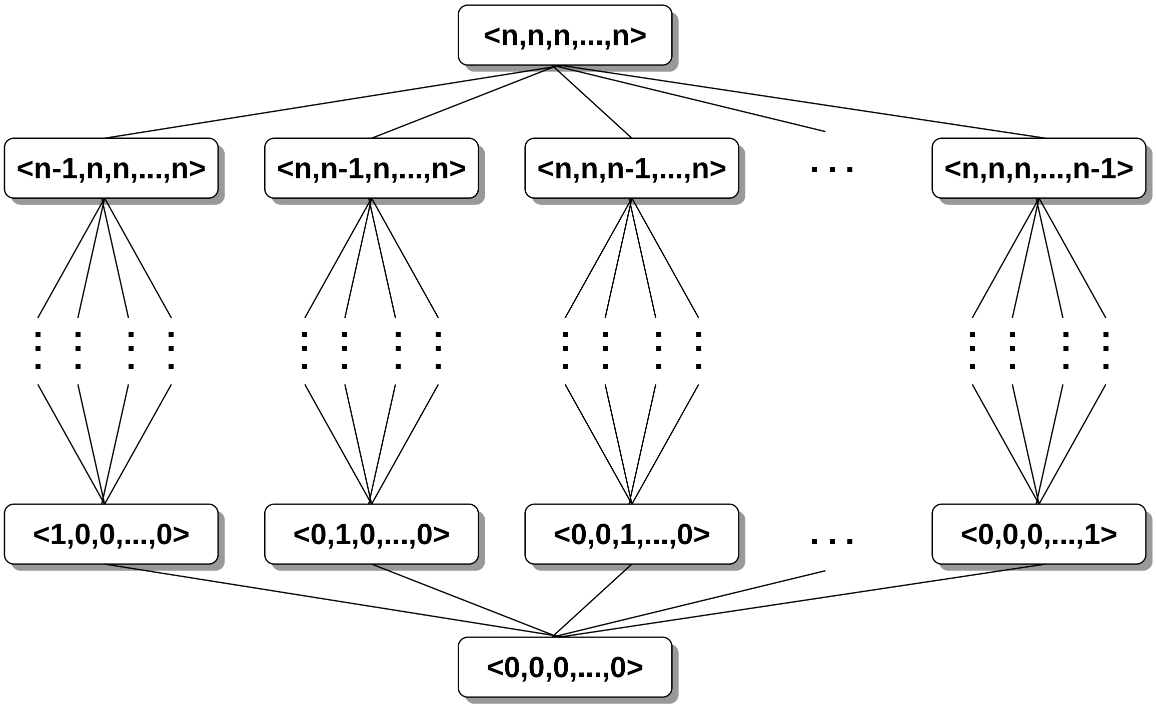
An interesting addition to this approach would be to allow the algorithm to adapt when its metrics and heuristics fail to make good resource management decisions. For example, if heuristics continually suggest the adjustment of a resource that does not result in improved quality of service, it is possible that the heuristics in use are in error. If it is difficult for the algorithm to adapt itself, then it should at least report its failings to allow someone to examine the situation and develop better heuristics. If this algorithm is implemented as a policy-code algorithm, as discussed in Section 4.3.2.1, even these administrator-assisted heuristic changes could be done at run time.

## 5.3 The Multidimensional Search Algorithm

Traditional searches, like binary searches or other brute force searches, are one dimensional. That is, there is a single key field that is being searched for a given target. Multidimensional searches [14, 15], on the other hand, involve data with multiple key fields being searched. All keys must match for a comparison to be successful, determining that the desired data has, in fact, been found. For quality of service management with multiple resources, we have multiple dimensions through which we which to search to find a resource allocation that meets a given quality of service expectation. Thus, a multidimensional search algorithm would seem to be suitable for solving this problem.

Instead of having a collection of data records being searched, the search space for quality of service resource management is composed entirely of every possible resource allocation combination. Consequently, we do not actually need to store any real data, which makes the process easier and less resource intensive. A two dimensional representation of the multidimensional logical search space of all resource allocations is shown in Figure 5.1, assuming each resource has $n$ levels of allocation.

**Figure 5.1** Multidimensional Search Space of All Resource Allocations




To find the resource allocation delivering our target level of quality of service, we utilize the same multidimensional search technique as referenced above. Instead of using a key comparison to determine if the appropriate resource allocation has been found to meet quality needs, we make the new allocation, measure the resulting quality delivered, and compare the new delivered quality to the target we are seeking. If quality matches and requirements are now met, we have found the desired resource allocation; otherwise, we simply continue the search, probing additional resource allocations.

While performing a multidimensional search for resource management is this simple conceptually, there is one rather large snag. Carrying out a multidimensional search absolutely requires that the target being sought is multidimensional—of the same size as the number of key fields. This is how the search algorithm decides which dimension to traverse, and in what direction. When using this multidimensional search process for quality of service resource management, we have a single dimensional quality target and multiple resource dimensions. In essence, we must find a way of splitting the quality target into multiple dimensions, one for each resource being managed, in a way that allows us to make traversal decisions to find the allocation satisfying quality constraints.

To make this split, we must be able to identify how much of the given quality target is determined by each resource. The target can then be split accordingly and searched for using the multidimensional search approach outlined above. This analysis can be done in a variety of ways, including an analysis of variance, as described in [83]. This experimental regression analysis yields the percentage influence each resource has on quality delivered, allowing us to split the quality target across resource dimensions. For example, suppose the expected frame rate for a video application is 30 frames per second, and 60% of this target can be explained by its CPU allocation, and 40% can be explained by its memory allocation. The target can be split into two values, requiring $30 \times 60\% = 18$ frames per second quality from its CPU resources, and $30 \times 40\% = 12$ frames per second quality from its memory resources. We can then use these values as the multiple keys for our search.

Updating our current "position" in our search space is also done in terms of our newly split multiple keys. When we make a resource allocation along one dimension, we keep track of this adjustment. The next time delivered quality is measured, we determine the amount of change in quality from its previous measurement—this change was induced by our adjustment. Consequently, we apply this change to the split key corresponding to the resource that was just adjusted to determine our new position in the search. For example, consider the situation given above. Suppose our current keys for CPU and memory are 12 and 10 frames per second respectively. Now, suppose that the CPU allocation is increased, since we have a target of 18 for that resource. Upon the next quality measurement, we find that this increase has lead to an increase of 3 frames per second in delivered quality. Consequently, we can adjust our current keys for CPU and memory to 15 and 10 respectively, since the 3 frames per second increase resulted from an adjustment to CPU resources.

### 5.3.1 The Algorithm

The multidimensional search resource management algorithm is presented in Algorithm 3.

**Algorithm 3** Multidimensional Search Resource Management

**Input:** $p_i \in$ PROCESS
**Input:** $e_i \in$ EXPECTATION
**Input:** $q$ /* $p_i$'s measured quality of service */
**Input:** $\vec{v}$ /* Percent variation explained by each resource $r \in$ RESOURCE */
**Output:** $a_i \in$ ALLOCATION $\mid e_i(q_i(a_i)) = 0$

**State:** $\vec{r} \leftarrow$ Enumeration of all $r \in$ RESOURCE
**State:** $n \leftarrow \mid$ RESOURCE $\mid$
**State:** $\vec{e} \leftarrow \vec{v} \times$ Quality target in $e_i$
**State:** $\vec{q} \leftarrow < 0, 0, \ldots, 0 >$
**State:** $\vec{a} \leftarrow < 0, 0, \ldots, 0 >$
**State:** $i \leftarrow 0$

1: **if** Number of iterations $> 1$ **then**
2: $\quad \vec{q}[(i - 1)(\bmod n)] \leftarrow \vec{q}[(i - 1)(\bmod n)] + (q - q')$
3: **end if**
4: Compute adjustment $a$ for resource $\vec{r}[i]$
5: $\vec{a}[i] \leftarrow \vec{a}[i] + a$
6: $a_i \leftarrow a_i + a$
7: $i \leftarrow (i + 1)(\bmod n)$
8: $q' \leftarrow q$

This algorithm is invoked from the general resource management algorithm in Algorithm 1. In this algorithm, additional information is given as input, and state is carried from iteration to iteration that is initialized before the first iteration. As an input, we are now also provided with the percentages of quality variation explained by each resource in an ordered vector. The number of elements in this vector is the number of resources being used by the application. As state initialized before executing the algorithm, we enumerate the resources being used by the application into another vector $\vec{r}$, determine the number of resources being dealt with $n$, split the expected quality target into a multidimensional key $\vec{e}$ for searching, initialize the current quality vector $\vec{q}$ to be 0, initialize the current resource allocation vector $\vec{a}$ to be 0, and set the index $i$ to the current resource being adjusted to the first resource. The application process is then started with the smallest resource allocation possible.

In steps 1–3 of the multidimensional search algorithm, we update the current quality vector to reflect the impact of the resource adjustment made in the previous iteration of the algorithm. Any quality changes are recorded into the vector slot corresponding to the resource adjusted. Steps 4–6 determine the adjustment for this resource, and put it in place. Lastly, steps 7–8 prepare for the next iteration of the algorithm by setting the next resource to examine to be the next resource in the vectors, and by storing the last measured quality for comparison in the next iteration.

### 5.3.2 Analysis of the Algorithm

In this section, we analyse the multidimensional search algorithm. This is done in terms of both its complexity, and a variety of other issues, as discussed earlier.

#### 5.3.2.1 Single Iteration Complexity

There are four main stages to the multidimensional search algorithm that contribute to single iteration complexity: attributing quality changes to the previous iteration, computing the resource allocation adjustment required, making the adjustment, and preparing for the next iteration. There are also additional state initialization steps for this algorithm, but since they are only done once, they do not factor in the single iteration complexity.

Attributing quality changes to the resource adjusted in the previous iteration involves only simple calculations and can be done in constant time. Computing the resource adjustment required can again be done in $O(c())$ time, as was the case with the metric-heuristic approach. (In fact, we provide several approaches later in this section that calculate adjustments for this algorithm in constant time.) The cost of making a resource adjustment is $O(a)$, as with the metric-heuristic approach. Lastly, preparing for the next iteration involves only simple calculations, and can also be done in constant time. Consequently each iteration of this algorithm can be carried out in $O(c()+a())$ time.

Once again, because elements of the multidimensional search algorithm are beyond our control, we cannot be more precise, unless we place undue restrictions on any of these procedures. Since such restrictions would be undesirable for many reasons, we can again go no further.

#### 5.3.2.2 Number of Iterations

The number of iterations required primarily depends upon two factors. The first factor is how adjustments are being made; two different approaches are discussed in detail below. The second factor is the accuracy of the variance analysis conducted to split quality of service targets into multidimensional keys. If this was not done accurately, achieving quality requirements may prove difficult since the split quality target may not translate into the overall quality objective expected. At the extreme, a poor analysis may mean that quality objectives are never satisfied. Fortunately, variance analyses also yield the level of error encountered, which could be used as a measure of reliability for use here.

#### 5.3.2.3 Further Discussion

Like the metric-heuristic algorithm, resource adjustments only occur when quality does not meet requirements. While the algorithm can handle most kind of environmental disturbances that force quality to no longer meet requirements, difficulties would arise if application resource needs change in such a way that the analysis of variance results are no longer accurate. In such a case, an adaptive approach, as described below, would be necessary.

Unlike the metric-heuristic algorithm, the multidimensional search resource management algorithm requires state to be maintained between iterations to track progress through the search space of resource allocations. This state however, is not substantial enough to pose a significant overhead to the algorithm. In fact, despite additional initialization steps, this algorithm has less run-time overhead in the best case because it does not require nearly as much metric collection or analyses as the metric-heuristic algorithm does. (Even if assumptions for the best case are not met, the metric-heuristic algorithm's best case also relies on the same assumptions and would be penalized similarly. Furthermore, the metric-heuristic algorithm's best case makes further assumptions not required for the multidimensional search algorithm, which could cause an even bigger difference in single iteration complexity when such assumptions are relaxed.)

In addition, prior knowledge on the applications being managed is required—the analysis of variance results. However, at the risk of further complicating the algorithm, it might be possible to make this algorithm adaptive. Not only would this permit the algorithm to be applied to applications without prior knowledge, but the same approach could potentially overcome inaccuracies in the variance analysis itself, overcoming the possible difficulties mentioned in the previous section.

For example, an application with unknown characteristics could be started with an even split across all resources (with 5 resources, each one would initially account for 20% of quality of service, for instance). If an adjustment to one resource does not result in a change in quality of service, its percentage would decrease and be distributed across the other resources. Likewise, if adjusting a particular resource results in a very significant change in quality, its percentage can be increased by taking away from less significant resources. (Taking away from or adding to other resources can be done evenly for simplicity, or unevenly based on the significance of the other resources to quality of service.) This approach may require additional iterations to get percentages adjusted appropriately, but eliminates the need for this information to be available a-priori, helps cope with changes in the process at run-time, and overcomes errors in the variance analysis process. In fact, once these percentages are computed for one run of a program, they can be stored and used in subsequent runs, with any penalties faced only once for possibly large gains.

It is important to note that there are two practical issues that might produce complications in some cases. The first comes from the state initialization of Algorithm 3, where the initial resource allocation is set to be 0. This is done to ensure so that there is no issue of how initial quality being delivered should be attributed to the various resources—all elements in the vector could also be safely set to 0. Unfortunately, actually providing an application process with no resources whatsoever will prevent the algorithm from working correctly—with no resources, the application cannot execute to provide quality metrics used in making subsequent resource allocations. Consequently, in practice, the algorithm must instead start with an initial resource allocation of at least $< 1, 1, \ldots, 1 >$ to allow the process to proceed. In most cases, this minimal allocation should result in near zero quality, allowing us to proceed as we normally would with the algorithm. If not, we would need to distribute the initial quality delivered according to the percentages provided from the variance analysis, and hope that this is suitable.

The second issue comes from the initial variance analysis itself. It is not uncommon for variance analyses to indicate that the combined influences of certain resources are statistically significant, in addition to individual effects. These scenarios could be handled by adding new dimensions for each significant combination; traversing these dimensions would simply result in increases or decreases to all of the resources in the combination.

### 5.3.3 Variation: The Multidimensional Binary Search Algorithm

A one dimensional binary search works by repetitively splitting the search space being searched in half. Since the data is known in advance to be in sorted order, the data to compare against is chosen to be exactly half-way through the remaining data. If the target key is greater than the chosen key, the lesser half of the search space is discarded. Likewise, if the target key is smaller than than the chosen key, the greater half is discarded. This process is repeated until the chosen key matches the target key, and the data record is found. A multidimensional binary search works in essentially the same manner, except that this process is done along each dimension in succession. Each comparison eliminates only half of the search space along one dimension, rather than half of the search space in its entirety.

Recall the quality of service ordering constraints introduced in Section 3.2—dictating that as resources are allocated to a process, quality delivered does not decrease, and that as resources are taken away from a process, quality does not increase. These constraints permit us to consider the multidimensional search space of resource allocations as being sorted; allowing us to use a binary search approach for finding the appropriate resource allocation to meet quality targets. For example, suppose we have a given resource allocation that is known to result in quality below requirements. We know, therefore, that reducing resources further will not increase quality delivered, so we only need to examine those allocations greater, and we can discard the appropriate portion of the resource allocation space.

Algorithm 4 fills in the details left in step 4 of Algorithm 3, showing how a multidimensional binary search can make resource adjustments. In this algorithm, we additionally let $\lceil \vec{a}\,[i] \rceil$ denote the largest value of $\vec{a}\,[i]$ not tested to date, and $\lfloor \vec{a}\,[i] \rfloor$ denote the smallest. The largest value is initialized to the maximum allocation, while the smallest value is initialized to the minimum allocation possible.

**Algorithm 4** Multidimensional Binary Search Resource Management

**if** $e_i(q_i(a_i)) > 0$ **then**
  $a \leftarrow \frac{(\vec{a}[i] + \lceil \vec{a}[i] \rceil)}{2}$
  $\lfloor \vec{a}\,[i] \rfloor \leftarrow \vec{a}\,[i]$
**else**
  $a \leftarrow \frac{(\vec{a}[i] + \lfloor \vec{a}[i] \rfloor)}{2}$
  $\lceil \vec{a}\,[i] \rceil \leftarrow \vec{a}\,[i]$
**end if**

The efficiency of a multidimensional binary search is quite good, at $\mathrm{O}(\log(n))$ in general, where $n$ is the number of items being searched through [14, 15]. Unfortunately, $n$ is the number of possible resource allocations, which is $l^r$, where $r$ is the number of resources, and $l$ is the number of different levels of allocation. However, this approach is still relatively efficient, and at least has known run-time complexity, unlike the metric-heuristic algorithm.

There is one potential drawback of this approach to adjustments, however. If resource needs for an application being managed change, the new allocation required to meet requirements might fall into the portion of the resource allocation search space already discarded as impossible by this algorithm. In such cases, it would be important to detect these resource needs changes and start searching again to find the appropriate allocation—otherwise, it may be possible to never find it. This detection can be made, for example, by noting a sudden dissatisfaction in delivered quality, when quality was stable and meeting requirements, or by discovering that the target resource allocation lies in a portion of the resource allocation search space already ruled out by the search. This was not a problem with the metric-heuristic approach as it does not maintain any state to become stale or incorrect when such events occur.

### 5.3.4 Variation: The Multidimensional Interpolation Search Algorithm

When one looks through the telephone book to find the phone number for a particular individual, one usually does not use a binary search. If, for example, the individual's name begins with a "B", why would one start half-way through the phone book, when the letter "B" is known to be so near the front of the book? Instead, the search should start a little into the book, and proceed from there. This kind of search, where the search keys allow us to make better choices for comparisons, is known as an interpolation search. Multidimensional interpolation searches make the same kind of intelligent choices, except along each dimension of the search space in succession.

For quality of service resource management, we can estimate how far we are away from meeting our quality of service target (normalizing the difference between currently delivered quality and the target is one approach). Using this estimate, we can make better decisions as to the next resource allocation to attempt, allowing us to use a kind of interpolation search to find the resource allocation necessary to meet quality constraints.

Algorithm 5 fills in the details left in step 4 of Algorithm 3, showing how a multidimensional interpolation search can make resource adjustments. In this algorithm, we additionally let $\lceil \vec{q}\,[i] \rceil$ denote the largest value of $\vec{q}\,[i]$ not observed to date, and $\lfloor \vec{q}\,[i] \rfloor$ denote the smallest. The largest value is initialized to the maximum possible quality, while the smallest value is initialized to the minimum quality possible. The variable $d$ determines where the expected quality provided by the current resource lies between the range of unexplored quality; this value is then mapped into a resource allocation that should approximate the location of the allocation needed to provide this quality.

**Algorithm 5** Multidimensional Interpolation Search Resource Management

4.1: $d \leftarrow \frac{\vec{e}\,[i] - \lfloor \vec{q}\,[i] \rfloor}{\lceil \vec{q}\,[i] \rceil - \lfloor \vec{q}\,[i] \rfloor}$
4.2: $a \leftarrow \lfloor \vec{a}\,[i] \rfloor + (d \times (\lceil \vec{a}\,[i] \rceil - \lfloor \vec{a}\,[i] \rfloor)) - \vec{a}\,[i]$
4.3: **if** $e_i(q_i(a_i)) > 0$ **then**
4.4: $\quad \lfloor \vec{a}\,[i] \rfloor \leftarrow \vec{a}\,[i]$
4.5: $\quad \lfloor \vec{q}\,[i] \rfloor \leftarrow \vec{q}\,[i]$
4.6: **else**
4.7: $\quad \lceil \vec{a}\,[i] \rceil \leftarrow \vec{a}\,[i]$
4.8: $\quad \lceil \vec{q}\,[i] \rceil \leftarrow \vec{q}\,[i]$
4.9: **end if**

The efficiency of a multidimensional interpolation search is even better than its binary search counterpart, at O(log log($n$)), where $n$ is the number of items being searched through [120]. Consequently, this algorithm is a good approach, with even better performance.

Unfortunately, it suffers a similar drawback as the binary search approach in Section 5.3.3 under certain dynamic operating conditions. However, if we perform our interpolation using only the difference between delivered quality and expected quality, without consulting areas ruled out by prior searches, we could avoid this problem as we are no longer maintaining the troublesome state data. In doing so, we would also gain considerable simplicity and efficiency, but possibly sacrifice some interpolation accuracy, perhaps requiring additional iterations as a result. An example of such an algorithm appears in Algorithm 6. (This approach assumes that the expected quality of service contribution of each resource is non-zero; any resources that would have a zero contribution can simply be omitted from the process.)

**Algorithm 6** Multidimensional Interpolation Search Resource Management with Less State

4.1: $a \leftarrow \frac{\vec{e}[i] - \vec{q}[i]}{\vec{e}[i]}$

## 5.4 The Round Robin Algorithm

Round robin algorithms place the items being examined in a circular list that is traversed in sequential order; upon reaching the end of the list, the traversal continues at the beginning of the list again. Consequently they can be very simple and easy to implement. Because of this, these algorithms have the potential to be the basis of very lightweight and nimble quality of service resource management algorithms.

The pure round robin algorithm for quality of service resource management is a simple approach. Each resource is visited in succession whenever a deviation from targeted quality is detected. If delivered quality is below requirements, then the allocation of the current resource is increased. Otherwise, if delivered quality is above requirements, then the allocation of the current resource is decreased. After making the change in allocation, the algorithm proceeds to the next resource.

### 5.4.1 The Algorithm

The round robin resource management algorithm is presented in Algorithm 7.

This algorithm is invoked from within the general resource management algorithm in Algorithm 1. State is also carried from iteration to iteration of this algorithm; this state is initialized before executing the algorithm. As state, we enumerate the resources being used by the application into a vector $\vec{r}$, determine the number of resources being dealt with

**Algorithm 7** Round Robin Resource Management

**Input:** $p_i \in$ PROCESS
**Input:** $e_i \in$ EXPECTATION
**Input:** $q$ /* $p_i$'s measured quality of service */
**Output:** $a_i \in$ ALLOCATION | $e_i(q_i(a_i)) = 0$

**State:** $\overrightarrow{r} \leftarrow$ Enumeration of all $r \in$ RESOURCE
**State:** $n \leftarrow |$ RESOURCE $|$
**State:** $i \leftarrow 0$

Compute adjustment $a$ for resource $\overrightarrow{r}[i]$
$a_i \leftarrow a_i + a$
$i \leftarrow (i + 1)(\text{mod } n)$

$n$, and set the index $i$ to the current resource being adjusted to the first resource. The application process is then started.

As noted earlier, the round robin resource management algorithm is quite simple. We compute a resource adjustment for the current resource, make the adjustment, and update the resource index to the next resource for the next iteration. As can be seen, this is considerably simpler than the algorithms presented in the previous sections.

### 5.4.2 Analysis of the Algorithm

In this section, we analyse the round robin resource management algorithm. This is done in terms of both its complexity, and a variety of other issues, as discussed earlier.

#### 5.4.2.1 Single Iteration Complexity

There are three main stages to the round robin algorithm that contribute to single iteration complexity: computing the resource allocation adjustment required, making the adjustment, and preparing for the next iteration. There are also additional state initialization steps for this algorithm, but since they are only done once, they do not factor in the single iteration complexity.

Computing the resource adjustment required can again be done in O($c()$) time, as was the case with both the metric-heuristic and multidimensional search approaches. (We can adapt the constant time adjustment calculations from the previous section for this algorithm as well.) The cost of making a resource adjustment is O($a$), as with the previous approaches. Lastly, preparing for the next iteration involves only simple calculations, and can also be done in constant time. Consequently each iteration of this algorithm can be carried out in O($c()$+$a()$) time.

Once again, because elements of the round robin algorithm are beyond our control, we cannot be more precise, unless we place undue restrictions on any of these procedures. Since such restrictions would be undesirable for many reasons, we can again go no further.

#### 5.4.2.2 Number of Iterations

While each execution of the algorithm is inexpensive, it is likely that a considerable number of iterations is required to reach the expected level of quality of service. This is likely because resources are adjusted blindly in order, regardless of actual need, and several unnecessary adjustments may be attempted before the required adjustments are finally made. It is only a matter of chance whether the correct resource allocation is adjusted at each iteration; the more resources being managed, the lower the probability that a correct choice is made. Consequently, the algorithm may never arrive at the correct resource allocation to deliver quality of service at the expected level.

To help reduce the number of iterations required, smarter resource allocation adjustments can be made. For example, the interpolation approaches discussed in previous sections are quite easy to compute, and produce good results. This may offset some of the inefficiencies introduced through incorrect choices of resources to adjust.

#### 5.4.2.3 Further Discussion

While the round robin resource management algorithm's effectiveness is rather questionable, it has other interesting features that make further examination of variations of the algorithm worth while. The algorithm is very light-weight and incurs very low overhead. The state required is quite small (only needing to track the current position in the resource list), and no prior knowledge of the application is necessary. Since resource adjustments only occur when quality does not meet requirements, adjustments will cease once acceptable quality. If disturbances occur in the dynamic environment to cause quality to deviate from requirements, this algorithm will continue on as it did before, still with questionably effectiveness.

Despite its good points, the round robin algorithm may not be reliable enough in this basic form for resource management purposes. Instead, we must look to variations of this algorithm which maintain its benefits and overcome its weaknesses.

### 5.4.3 Variation: The Weighted Round Robin Algorithm

The weighted round robin algorithm is aimed at keeping the simplicity of the pure round robin algorithm, but adding intelligence to its choices of resources to adjust to improve the effectiveness of the algorithm. In the pure round robin algorithm, each resource is adjusted an equal number of times. In reality, some resources have larger impacts on application quality then others. These resources should be visited and adjusted more frequently than those that have little or no significant impact.

To do this, weights are assigned to each resource, indicating the relative number of times the algorithm should visit this resource for adjustments. (These adjustments could be consecutive, or spread out more evenly over time.) The larger the weight, the more frequently it is adjusted; the smaller the weight, the less often it is adjusted. This allows adjustments to be made where they matter the most.

Naturally, the question becomes, how does one compute these weights? One approach is to do an analysis of variance [83], as suggested for the multidimensional search algorithms above, to determine the relative importance of each resource to quality of service. More important resources receive higher weights, and those that are less important receive lower weights. Consequently, this variation of the round robin algorithm requires prior knowledge to effectively select weights, which requires experimentation in advance, or extensive knowledge of the application and computing environment.

The algorithm for the weighted round robin approach is given in Algorithm 8. Changes from the round robin algorithm in Algorithm 7 are minimal. A collection of integral weights is now provided as input to the algorithm to better determine which resources need allocations. Additional state is now required and is initialized before execution. Weights provided to the algorithm are used to produce a distribution of resource indices $\overrightarrow{d}$—this distribution can be stepped through sequentially to determine the next resource to allocate, based on the indices stored in the slots of this vector. This distribution can be uniform or clustered in arbitrary ways. The size of this distribution vector is initialized into $m$. Steps 1 and 2 now use this distribution vector to determine the next resource to allocate, instead of sequentially as before. Finally, step 3 updates the index to refer to the next entry in the distribution vector.

The weighted round robin approach has the potential to be more effective than the pure round robin approach, since adjustments are made where they count most with greater frequency. It must be noted however, that mistakes can still be made. If weights are not correctly computed initially, problems could occur with adjustments not being made at the correct resources. Even if the weights are correct, this approach may choose the wrong

**Algorithm 8** Weighted Round Robin Resource Management

**Input:** $p_i \in$ PROCESS
**Input:** $e_i \in$ EXPECTATION
**Input:** $q$ /* $p_i$'s measured quality of service */
**Input:** $\overrightarrow{w}$ /* Resource weightings for each resource $r \in$ RESOURCE */
**Output:** $a_i \in$ ALLOCATION $\mid e_i(q_i(a_i)) = 0$

**State:** $\overrightarrow{r} \leftarrow$ Enumeration of all $r \in$ RESOURCE
**State:** $\overrightarrow{d} \leftarrow$ Distribution of $\overrightarrow{w}$ storing indices to $\overrightarrow{r}$
**State:** $n \leftarrow \mid$ RESOURCE $\mid$
**State:** $m \leftarrow \sum_{j=1}^{n} \overrightarrow{w}[j]$
**State:** $i \leftarrow 0$

```
Compute adjustment a for resource r[d[i]]
a_i ← a_i + a
i ← (i + 1)(mod m)
```

resources to increase or decrease at the wrong time, and in the worst case, the algorithm could still be unable to find the appropriate resource allocation. For example, suppose that an application has too many CPU resources, but sufficient memory resources. If memory is a relatively important resource to the application, the algorithm may still choose to decrease the application's memory allocation, if it happens to be that resource's turn for adjustment, which could have adverse effects on quality of service.

It is also important to note that the addition of weights to the algorithm does not significantly increase the overhead of the algorithm. The only substantial change comes from computing the distribution vector used to determine which resources to allocate and the order to allocate them in. Simple distributions where all allocations of each resource are done in clusters can be done in $O(m)$ steps, with $m$ computed as in Algorithm 8. Uniform distributions, which are more complex to compute, can be created in $O(n \times m)$ steps. Either way, this cost is only incurred once, at application initialization time, and is quite acceptable. Furthermore, overall efficiency would likely improve with this approach, since fewer executions of the algorithm would be required to make adjustments, since they are now carried out in a more informed manner.

### 5.4.4 Variation: The Dynamically Weighted Round Robin Algorithm

There are two problems with the weighted round robin approach. First, the weights used by the algorithm need to be computed in advance before using the application. This can be inconvenient and impractical in some cases, and at the very least could be a time consuming process. Second, the weights used by the algorithm are static and do not change throughout the life time of the application. Consequently, this algorithm cannot cope very easily with changes in resource needs, or errors in the process of computing the weights in the first place. The dynamically weighted round robin algorithm is aimed at dealing with these shortcomings.

Instead of pre-computing weights through variance analysis or some other means, each resource is initially assigned an equal weight. As the algorithm executes, adjustments are made to resource allocations as before. If an adjustment yields significant improvements in delivered quality, the weight of the resource is increased to encourage further adjustments to this resource in the future. Likewise, if an adjustment has little impact on delivered quality, the weight of the resource is decreased accordingly. In tuning resource weights in this way, we can arrive at good weight values without prior knowledge or experimentation. Further, the ability to dynamically adjust weights helps keep up with application changes in resource needs as the weights in use can be adjusted to reflect the new needs.

**Algorithm 9** Dynamically Weighted Round Robin Resource Management

**Input:** $p_i \in$ PROCESS
**Input:** $e_i \in$ EXPECTATION
**Input:** $q$ /* $p_i$'s measured quality of service */
**Output:** $a_i \in$ ALLOCATION $\mid e_i(q_i(a_i)) = 0$

**State:** $\vec{r} \leftarrow$ Enumeration of all $r \in$ RESOURCE
**State:** $\vec{w} \leftarrow < 1, 1, \ldots, 1 >$
**State:** $\vec{d} \leftarrow$ Distribution of $\vec{w}$ storing indices to $\vec{r}$
**State:** $n \leftarrow \mid$ RESOURCE $\mid$
**State:** $m \leftarrow \sum_{j=1}^{n} \vec{w}[j]$
**State:** $i \leftarrow 0$

Compute adjustment $a$ for resource $\vec{r}[\vec{d}[i]]$
$a_i \leftarrow a_i + a$
**if** Number of iterations $> 1$ **then**
  **if** $\mid q - q' \mid <$ Lower Adaptation Threshold **then**
    $\vec{w}[(i - 1)(\bmod m)] \leftarrow \vec{w}[(i - 1)(\bmod m)]$ - Adaptation Value
    $\vec{d} \leftarrow$ Redistribution of $\vec{w}$ storing indices to $\vec{r}$ starting at $i$
    $m \leftarrow \sum_{j=1}^{n} \vec{w}[j]$
  **else if** $\mid q - q' \mid >$ Upper Adaptation Threshold **then**
    $\vec{w}[(i - 1)(\bmod m)] \leftarrow \vec{w}[(i - 1)(\bmod m)]$ + Adaptation Value
    $\vec{d} \leftarrow$ Redistribution of $\vec{w}$ storing indices to $\vec{r}$ starting at $i$
    $m \leftarrow \sum_{j=1}^{n} \vec{w}[j]$
  **end if**
  $q' \leftarrow q$
  $i \leftarrow (i + 1)(\bmod m)$
**end if**

The algorithm for the dynamically weighted round robin approach is given in Algorithm 9. Changes from the weighted round robin algorithm in Algorithm 8 are again minimal. Weights are no longer provided as input to the algorithm, but are initialized as state in vector $\vec{w}$ to all have the same value. These weights are then used in initializing state vector $\vec{d}$ to produce a distribution of resource indices as before. Steps 3–13 have been added to dynamically adjust the weightings and recreate the distribution vector; these steps execute only after the first iteration, as we need at least one resource allocation adjustment to determine if our weighting need to be changed accordingly. Steps 4–7 execute if the last resource adjustment made resulted in a difference in quality delivered that is below a lower threshold. This signifies that the last resource adjusted is not as significant as other resources, and its weighting should be decreased. Its weighting is decreased by some adaptation value, which could be constant, or based on the difference in quality provided by this prior resource allocation adjustment. The weightings are then redistributed, starting from this point onward in the distribution vector, to ensure that the resource adjusted in this iteration will still be the prior resource adjusted in the next iteration. Similarly, steps 8–12 execute if the last resource adjustment made resulted in a difference in quality delivered that is above an upper threshold. This signifies that the last resource adjusted is more significant than other resources, and its weighting should be increased. A similar weight adaptation as above is then computed and put in place. Lastly step 13 records the quality delivered measured in this iteration for future comparisons and weight adaptations.

Since the algorithm can adjust its behaviour to recover from poor weights, dynamic operating conditions, errors, or poor prior adjustment choices, the effectiveness and efficiency of the algorithm should be improved, as fewer executions of the algorithm are required to produce the desired results. The algorithm has only slightly higher overhead than its predecessors as it must maintain more state, and make adjustments to weights when conditions warrant them. It is important to note, however, that the algorithm can still potentially make incorrect decisions, as discussed in the previous section, in choosing the wrong resource allocation to adjust at the wrong time. It is also still possible for this algorithm to be unable to find the correct resource allocation to meet quality requirements in the worst case. However, because of the adaptive nature of this algorithm, the chances of this are much less than with its predecessors.

As another interesting extension, if we did have prior knowledge or experimental results available, precomputed weights could make the process smoother and quicker. In such cases, we could start an application at these weights, as opposed to an even distribution, and would consequently required fewer adaptations to obtain acceptable weights. Consequently, it may be beneficial to store weights used in one execution of an application for its next execution—less tuning would be required, and acceptable results could be achieved sooner.

## 5.5 Summary

In this chapter, we examined three main classes of algorithms: metric-heuristic algorithms, multidimensional search algorithms, and round robin algorithms. While each algorithm has its own strengths and weaknesses, all are worthy of further examination. In the next chapter, we examine a prototype implementation of our architecture framework containing these resource management algorithms. Using this prototype, we can study these algorithms in more detail through experimentation.

# Chapter 6

# Prototype Implementation

In this chapter, we discuss a prototype system that has been developed to implement the architecture and algorithms described in Chapters 4 and 5.

## 6.1 Instrumentation Implementation

Instrumentation is provided to application developers through a C++ class library. C wrappers to the library are also provided to mask the object interface in C programs. Each sensor, actuator, and coordinator were implemented as a C++ class; probes were implemented as either methods or functions, depending on the objects with which the probes were associated. The library contains a variety of standard sensors, actuators, and coordinators for a variety of management functions. The library can also be specialized to develop new classes of instrumentation components to meet the unique needs of specific applications. A full OMT diagram [167] of the library is too large for inclusion here; relevant portions are diagramed and discussed in the sections below.

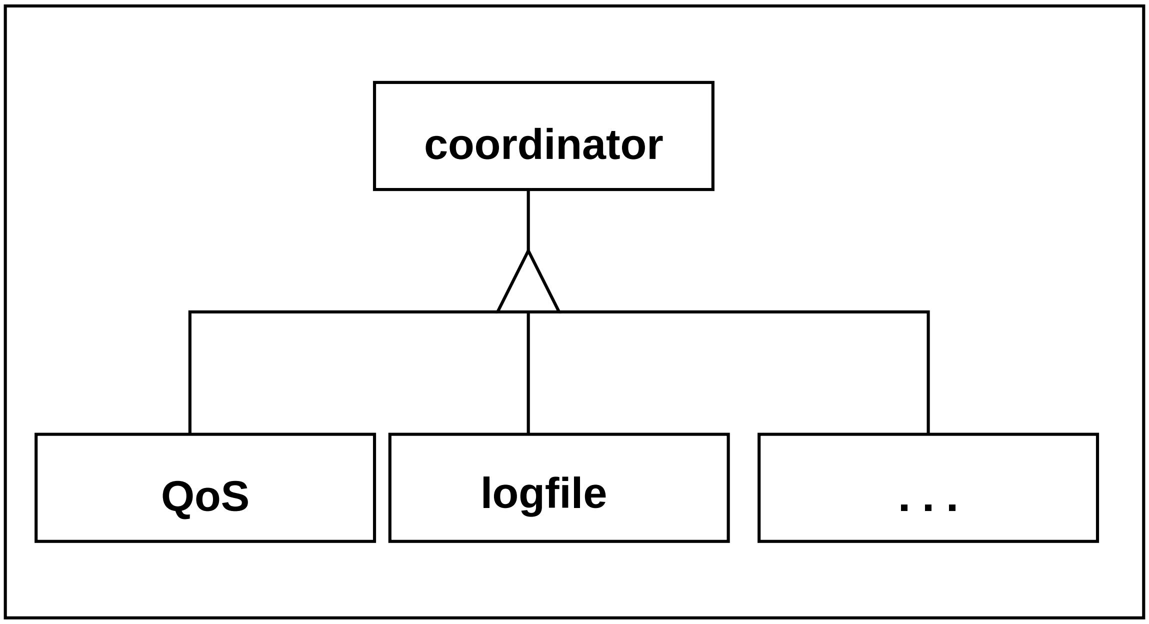
### 6.1.1 Coordinators

A variety of coordinators are provided as part of our instrumentation library. Of interest to this current work are the base coordinator class, the QoS coordinator, and the logfile coordinator, which are shown in Figure 6.1 and discussed in detail below.

#### 6.1.1.1 Base Coordinator Class

The base coordinator class is a C++ abstract class from which all other coordinator classes are derived. As such, it provides the common elements that each coordinator needs, as

**Figure 6.1** Partial OMT Diagram of Implemented Coordinators




well as abstract methods to provide a standard interface for working with coordinators, and other instrumentation components. These elements include the following:

- A mechanism to ensure that only one coordinator can be instantiated at any given time.
- Mechanisms to disable and re-enable all of the management functions within the process by disabling and re-enabling the coordinator itself.
- Multiple threads of execution to support several management system requests, along with the handling of sensor reports and actuator calls, concurrently. Locking facilities, provided through the use of mutexes, are available to maintain consistency within the coordinator.
- Registration lists of sensors and actuators instrumented into the application processes. These lists contain the unique identifier, type, name, description, and object handle of each sensor and actuator. This information is used to support the routing of management requests to individual instrumentation components, and to identify components submitting management reports to the coordinator.

#### 6.1.1.2 QoS Coordinator

The quality of service coordinator class has been derived from the base coordinator class. In addition to the coordinator facilities described above, this class has been provided with extensions to support quality of service management through local host managers.

This coordinator class can read and parse policies from a file into an internal format for use in violation detection. Upon receiving a report from a sensor with a violated threshold, these policies are consulted to determine which have been violated, and the appropriate policy violation reports are sent to the local QoS Host Manager.

Communication of policy violation reports from the quality of service coordinator to host managers is accomplished using message queues, as they are very efficient unidirectional communication channels. Communication originating from host managers back to quality of service coordinators uses sockets instead, as this communication tends to be bidirectional. More complex schemes involving multiple message queues or other mechanisms more efficient than sockets were considered, but rejected as these mechanisms are available in significantly more limited quantities than sockets, and the coordinator for each process would require one. As multiple quality of service enabled processes can execute on a single host, a fixed port number for the quality of service coordinator cannot be used since each process would need to listen to this port. It simply selects one at random and includes this with its registration to the local host manager.

Further details of this coordinator can be found in [132], where it was originally developed.

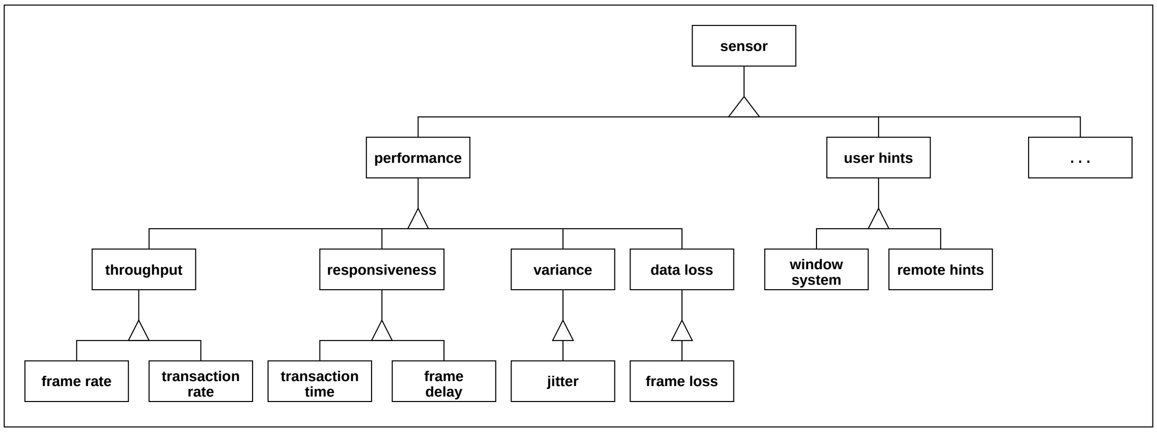
#### 6.1.1.3 Logfile Coordinator

The logfile coordinator class is derived from the base coordinator class. Essentially, this class takes all sensor reports and other messages, and logs these to either a file or the controlling terminal that launched the process containing the coordinator. While the management capabilities of this coordinator are limited (it does not communicate with any external management systems), it is quite useful for tracing and debugging application processes.

### 6.1.2 Sensors

A variety of sensors are also provided as part of our instrumentation library for facilitating application measurement. Of interest to this current work are the base sensor class, performance metric collection sensors, and user hint sensors, which are shown in Figure 6.2 and discussed in detail below.

**Figure 6.2** Partial OMT Diagram of Implemented Sensors




#### 6.1.2.1 Base Sensor Class

The base sensor class is a C++ abstract class from which all other sensor classes are derived. As such, it provides the common elements that each sensor needs, as well as abstract methods to provide a standard interface for working with sensors. These elements include the following:

- A generic asynchronous reporting mechanism to periodically report measured sensor attributes.
- A generic thresholding mechanism to generate sensor reports when threshold conditions are violated.
- Mechanisms to disable and re-enable sensors.
- Multiple threads of execution to support asynchronous reporting and other activities simultaneously. Locking facilities, provided through the use of mutexes, are available to maintain consistency within each sensor.

#### 6.1.2.2 Performance Sensors

The performance sensor class is a C++ abstract class derived from the base sensor class. The main purpose of this class is for organizing the sensor class hierarchy; all sensors measuring performance metrics are derived from this class. Consequently, these sensors are all key to the dynamic quality of service management process, as they collect the values of attributes specified in quality of service expectation policies. These sensors include the following:

**Throughput Sensors:** Throughput can be generally defined as the rate at which units of work can be completed, per unit of time. While the unit of work in question is application dependent, the bulk of the processing, analysis, and reporting of throughput metrics is common across all metrics. These commonalities are captured in a generic throughput sensor class that is derived from the performance sensor class.

Since the collection portion of the sensor depends on the type of metric being collected, several throughput sensors have been derived for our work. For video applications, the rate at which media frames are captured, decoded, or presented to the user is quite important. Consequently, a frame rate sensor has been derived from the throughput sensor class for this purpose. Likewise, for transaction-based applications, such as databases and electronic commerce applications, a transaction rate sensor has been derived from the throughput sensor class.

**Responsiveness Sensors:** Responsiveness can be generally defined as the amount of time required to complete a unit of work, or a logical collection of units of work. Once again, while the unit of work in question is application dependent, the bulk of the processing, analysis, and reporting of responsiveness metrics is common across all metrics. These commonalities are captured in a generic responsiveness sensor class that is derived from the performance sensor class.

Again, since the collection portion of the sensor depends on the type of metric being collected, several responsiveness sensors have been derived for our work. For transaction-based applications, a transaction time sensor has been derived from the responsiveness sensor class to measure the length of a transaction, from start to completion. For video applications, a frame delay sensor has been derived to measure the length of time between receipt of video frames for display. Many other similar sensors have also been developed.

**Variance Sensors:** Variance can be generally defined as the amount of change observed in some other metric being measured. Once again, while the metric in question is application dependent, the bulk of the processing, analysis, and reporting of variance is common across all metrics. These commonalities are captured in a generic variance sensor class that is derived from the performance sensor class.

Again, since the collection portion of the sensor depends on the type of metric being observed, several variance sensors have been derived for our work. The most important of these sensors is a jitter sensor used for measuring the variance in frame delay in video streams for video applications. As indicated in [142], video jitter can have a more significant impact on perceived quality of service than merely the mean frame delay or frame rate alone. Consequently, being able to measure and enforce quality policies involving jitter can be very important.

**Data Loss Sensors:** Data loss occurs in a variety of applications for a variety of reasons. In some cases, the slightest data loss is catastrophic; in others, certain levels of loss can be tolerated. Once again, while the situation is application dependent, the bulk of the processing, analysis, and reporting of variance is common across all applications. These commonalities are captured in a generic data loss sensor class that is derived from the performance sensor class.

Again, since the collection portion of the sensor depends on the application in question, several data loss sensors have been derived for our work. The sensor most widely used was the frame loss sensor for video applications that measured the number of video frames dropped from a streamed video source. Our experimentation has indicated that such a sensor can be quite important, as certain overload conditions can lead to nearly a 100% frame loss in this kind of application.

#### 6.1.2.3 User Hint Sensors

A study of computing environment usage by faculty, staff, and graduate students in our department [92] revealed that users frequently ignore executing applications for extended periods of time, which can result in significant wastage of scarce computing resources. Each user had on average 17 application windows open at any given time; less than 15% of the study group had less than 10 windows open at any given time. Less than 15% of the study group logged off when they were done a session; the rest used a screen saver or lock of some kind. As a result, these users had their screens saved or locked nearly 65% of the time, with applications still executing and consuming resources underneath. With a screen saver or lock not active, only 22% of the applications executing on average at any time were completely visible and usable to the user. Nearly 55% of the applications running were minimized, and the rest were partially or completely obscured from view, being covered by other windows. In the end, the average application spends only about 40% of its life completely visible and usable; the rest is spent either minimized or covered in some way. In this 60% of its life, however, the application still consumes resources, even though the application is essentially useless and is not providing meaningful service to users. Other studies and evaluations [10, 57, 88] echo these results. These scenarios result in resources being wasted and spread thin over some applications unnecessarily, and quality requirements for other applications not being satisfied.

Applications that are the focus of user interest and activity should be favoured in resource allocation and scheduling to provide the best possible quality of service, as opposed to those applications that users do not care about and essentially ignore. Fortunately, users often provide hints of their interest and activity through interactions with their computing environments or applications directly. For example, as demonstrated above, applications that users are not attending to tend to get minimized or covered over on the window desktop in favour of those on which the user is focussing.

This realization led to the development of user hint sensors. These sensors are used to detect relevant user activity and report this activity through instrumentation coordinators to the local QoS Host Manager. Based on submitted activity reports, the manager can then make informed decisions to reallocate and reschedule resources to applications that are the focus of user activity; thereby allowing quality of service to be provided where it is truly needed, making more effective use of computing resources, and reducing wastage. Taking advantage of user hints has an additional, beneficial side effect. By directing resources towards applications that are the focus of user activity, we are effectively reducing the number of applications being managed. This, in turn, increases the efficiency and reduces the overhead of the resource management process.

The user hint sensor class is a C++ abstract class that has been derived from the base sensor class. It provides the common analysis, processing, and reporting mechanisms for all user hint sensors. Subclasses of the user hint sensor class are derived to provide collection and additional analysis mechanisms tuned to particular types of user hints. These include the following:

**Window System User Hint Sensor:** This sensor is derived from the base user hint sensor to detect and track user interactions with their windowing environment. These interactions include the minimizing and restoring of windows, covering and uncovering of windows, and activation and deactivation of screen savers and locks. Depending on the hint detected, this could be an indication that the instrumented application has either gained or lost the focus of the user's attention.

The window system user hint sensor is designed for use with the X Window System [161]. User activity is translated by the computing environment and X server into X events that are sent to the instrumented X client application. User hints are collected by this sensor directly from the event stream or from probes embedded in the original process code. Based on the type of event and related event data, the appropriate user hint report is generated and given to the coordinator for further processing and forwarding.

**Remote User Hint Sensor:** The remote user hint sensor is essentially a proxy sensor for use in distributed applications. In distributed applications, the behaviour of one component process can determine the behaviour of other processes. Consequently, user hints processed by user hint sensors in some processes may need to be propagated to other remote processes. In such cases, a remote user hint sensor resides in the remote processes to receive and process these user hints accordingly. Propagation of user hints can be done through direct communication, or through the use of the QoS Domain Manager.

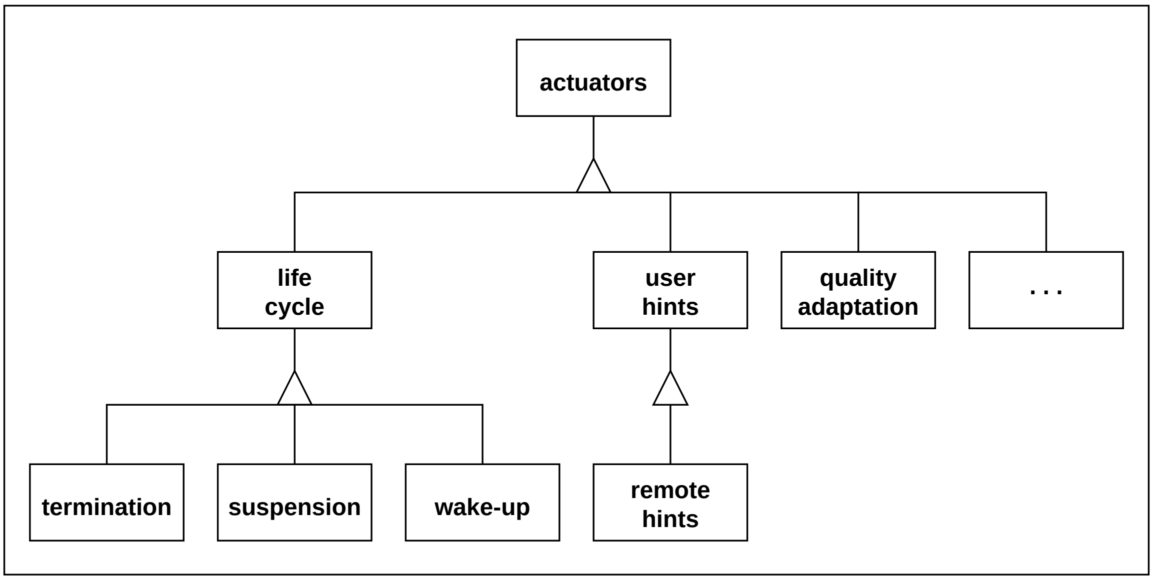
For example, consider a client-server video-on-demand application, in which a video server streams a selected movie to a remote client running on a user's workstation. If the user walks away from the workstation, and a screensaver is activated, this indicates the user's current lack of interest in the application. A window user hint sensor in the client process detects the screen saver activation and has local resources redistributed accordingly. However, the server is still consuming resources on its host, as well as network bandwidth in transmitting the video stream that is currently of low priority. By propagating the user hint through a remote user hint sensor, the video-on-demand server can trigger similar resource redistributions on its own end.

It is important to point out that both of these user hint sensor classes are generic, and can be used in a wide variety of applications. It is also possible to derive application specific user hint sensor classes from the base class. For example, a user hint sensor can be derived to detect when audio volume has been turned down by the user of a multimedia application, indicating that this aspect of the application is no longer important. Furthermore, existing classes of sensors may require behavioural specialization for particular applications. For example, a video playback application may require minimal or no resources when it is minimized by the user. However, a web browser may be minimized by the user while it is retrieving a document over the network—taking all of its resources away so it cannot download the document in the background would not at all reflect the user's intentions and expectations. On the other hand, if the browser is not downloading a document, its resources may be safely taken away so they are not wasted on animations or applets in which the user currently has no interest (we have conducted experiments which found that a Netscape browser animating an $80 \times 80$ pixel GIF image could easily consume 25% of the CPU cycles on a 140 Mhz UltraSparc 1 compute server). Additional fine tuning of instrumentation behaviour can be done at run time by adjusting the thresholding and filtering of data, as well as disabling and re-enabling instrumentation. This allows the collection and handling of user hints from applications to be tailored as needed as user and environment conditions change over time.

### 6.1.3 Actuators

A variety of actuators are also provided as part of our instrumentation library for facilitating application control. Of interest to this current work are the base actuator class, life-cycle control actuators, user hint actuators, and quality adaptation actuators, which are shown in Figure 6.3 and discussed in detail below.

**Figure 6.3** Partial OMT Diagram of Implemented Actuators




#### 6.1.3.1 Base Actuator Class

The base actuator class is a C++ abstract class from which all other actuator classes are derived. As such, it provides the common elements that each actuator needs, as well as abstract methods to provide a standard interface for working with actuators. These elements include the following:

- A generic action invocation mechanism to carry out an actuator's tasks.
- Mechanisms to disable and re-enable actuators.
- Multiple threads of execution to support asynchronous action invocations and other activities simultaneously. Locking facilities, provided through the use of mutexes, are available to maintain consistency within each actuator.

#### 6.1.3.2 Life Cycle Actuators

The life cycle actuator class is an abstract class derived from the base actuator class. The main purpose of this class is for organizing the actuator class hierarchy; all actuators controlling the life cycle of an application process are derived from this class. These actuators include the following:

**Termination Actuators:** The purpose of a termination actuator is to provide the ability to shut down an application in a variety of different ways. Depending on the options given to the actuation, this shutdown can be gentle, or very abrupt, without giving the application a chance to respond. Optionally, the application can register termination callback functions with this actuator to ensure that the appropriate clean-up operations are completed before the process terminates.

**Suspension Actuators:** The purpose of a suspension actuator is to provide a management system with the ability to suspend an application process's operations. The suspension can apply to the entire process including management, the entire process excluding management, and individual process threads.

**Wake-up Actuators:** The purpose of a wake-up actuator is to undo the effects of using a suspension actuator to suspend a process's activity. If the entire process including management was suspended, however, this actuator will not be able to resume process execution, as this actuator will also be suspended.

#### 6.1.3.3 User Hint Actuators

User hint actuators are used to generate, or rather simulate, user hints in an application process. The user hint does not actually occur, but an event is generated to make the application believe that a hint really did occur. This is primarily used within the remote user hint actuator to help propagate user hints from remote application processes, and is typically used in conjunction with a remote user hint sensor, described above.

#### 6.1.3.4 Quality Adaptation Actuators

The quality adaptation actuator class is derived from the base actuator class. The purpose of this class is to provide a facility where an application process can register adaptation functions to be called whenever quality of service cannot be delivered through resource management alone. If sufficient resources do not exist to support all applications in a distributed computing environment, some applications may be requested to adapt to fewer resources. When the situation is remedied, they may be requested to adapt to having additional resources available. For example, a video application may be able to adapt to changing resource availability by tuning video resolution, colour depth, compression quality, and a variety of other parameters appropriately.

At this point, this particular actuator is not used extensively in this work. It does, however, provide valuable functionality for future investigations in this area.

### 6.1.4 Probes

Probes are the mechanisms used to link instrumentation sensors and actuators into application process code; in fact, as mentioned earlier, probes are the only instrumentation components that must be added to the original code. Sensor probes are implemented as C++ methods, that are invoked to collect the relevant information. In C applications, these methods are suitably wrapped in C functions that hide the underlying C++ code. Actuator probes are usually provided in the form of functions or methods that an application process registers with an actuator to assist the actuator in carrying out its actions. It is not uncommon, however, for actuators to not require any probes at all, such as life cycle actuators, for example.

## 6.2 Host and Domain Managers

Both the host and domain managers are multithreaded daemon processes implemented in C and C++. As daemon processes, these managers are servers that are continuously running in the background; they are executed by the system when it boots, and not executed explicitly by users. Both daemons spawn children processes to handle multiple concurrent policy violations and various other requests as necessary. Brief descriptions of the various components of the managers is given below; further details can be found in [132].

### 6.2.1 Communication

As mentioned in the previous section, instrumented processes only communicate with local host managers. Communication from the instrumented processes to the host manager is done through message queues for quick and efficient notification of violations. Communication initiated from the host manager to an instrumented process is done through a socket connection to the process's coordinator. This allows simple bi-directional communication for retrieving additional process information or for carrying out actions and returning results.

Host managers do not directly communicate with one another, but rather do so through a domain manager which has a more global perspective of the computing environment and the applications executing within it. Communication between host managers and the domain manager is done through TCP sockets, regardless of which manager initiates the communication.

### 6.2.2 Fact Base, Rule Base, and Inference Engine

The fact base, rule base, and inference engine components of host and domain managers are implemented through the use of the CLIPS (C Language Integrated Production System) inference engine [63]. CLIPS uses first order predicate logic to implement an expert system. Because it is easily extended, embeddable in applications, freely available in source form, and supports dynamic loading and changing of facts and rules, CLIPS was a good candidate for this implementation. We make use of a slightly stripped and optimized version of CLIPS for greater performance and efficiency.

The fact base is expressed as a collection of CLIPS tuples, with one fact per tuple. When a quality of service violation report is received by the host manager from an instrumented process, a collection of tuples representing the relevant information from the report are inserted in to the fact base for later processing, including the quality of service metrics measured, the threshold that was violated, and information on the process where the violation occurred. Example facts can be found be consulting Appendix A.

Rules in the rule base are expressed as a left-hand-side and a right-hand-side. Essentially, if facts exist in the fact base to match the left-hand-side of a rule, the actions listed in the right-hand-side are executed. Rules are written in a CLIPS language, which are compiled into a binary format for quicker execution. Actions in rules can either be written in CLIPS, or functions written in C or C++. While greater usage of CLIPS provides better run-time flexibility and dynamic characteristics, C or C++ code executes with greater efficiency and performance. Consequently, a balance must be struck accordingly. Example rules can be found be consulting Appendix A.

## 6.3 Resource Management Controller

The resource management controller in host managers has been implemented as a combination of C and C++ code, as well as CLIPS facts and rules. Shared memory is used to allow resource management state to be shared between the multiple host manager processes and threads, which is required for state-based algorithms.

The controller is designed to be flexible in its support for resource management algorithms and management mechanisms. Several resource management algorithms can be supported by the controller at the same time; in fact, different application processes can be managed by different algorithms simultaneously. The same algorithm can also be applied to different processes in the different ways by tuning the algorithm parameters appropriately. Furthermore, algorithms can be fine-tuned or switched completely during the life-time of a process. There may also be multiple resource managers capable of managing the same computing resources. Again, this allows different processes to be managed in different ways; this can also be changed throughout the life-time of an application process. Much of this flexibility is due to the use of CLIPS—by using facts and rules to guide this process, we can use a variety of enforcement policies to dictate how resource management is done. By backing up this process with C and C++ code underneath, we can ensure efficiency and good performance as well.

For experimentation purposes, resource management algorithms from Chapter 5 have been implemented—the metric-heuristic algorithm, a multidimensional search algorithm, and a round robin algorithm. This provides a wide sampling of different algorithm characteristics to explore and analyse. The algorithms that have been implemented to date are discussed in the sections below.

### 6.3.1 The Metric-Heuristic Algorithm

The metric-heuristic algorithm is implemented primarily using CLIPS facts and rules. This allows the heuristics used in analysis to be changed easily at run-time by simply changing rules, or by changing facts that affect the rules' computations. While the set of possible metrics that can be used as input to the heuristic analysis process are fixed by the instrumentation available and resource manager capabilities, the actual metrics used can also be changed in a similar fashion. Functions to actually collect metrics are implemented in C, as CLIPS does not provide these capabilities itself.

### 6.3.2 The Multidimensional Interpolation Search Algorithm

Unlike the metric-heuristic algorithm, the core logic of the multidimensional interpolation search algorithm is fixed. Consequently, this algorithm is almost entirely coded in C and C++ for efficiency. CLIPS rules are used to invoke this code when violations are processed for applications being managed by this algorithm. Because there is still this CLIPS element to the algorithm, all variable parameters to this algorithm can be made available through CLIPS facts and rules, and can therefore be dynamically changed at run-time and handled on a per application basis. For example, the percentages of quality variation explained by each resource are given as CLIPS facts that vary from application to application.

The multidimensional interpolation search variation of the multidimensional search algorithm was selected as its method of determining a resource adjustment amount is theoretically more efficient than that used in a multidimensional binary search. We further selected the variation of the algorithm that required less state (given in Algorithm 6) as it is more efficient and significantly less complex than the other possible variations.

### 6.3.3 The Dynamically Weighted Round Robin Algorithm

Again, unlike the metric-heuristic algorithm, the core logic of the dynamically weighted round robin algorithm is fixed. Consequently, this algorithm is almost entirely coded in C and C++ for efficiency. CLIPS rules are used to invoke this code when violations are processed for applications being managed by this algorithm. Because there is still this CLIPS element to the algorithm, all variable parameters to this algorithm can be made available through CLIPS facts and rules, and can therefore be dynamically changed at run-time and handled on a per application basis. For example, the thresholds used to adapt weight values used in the algorithm are given as CLIPS facts that can vary from application to application, and can be adjusted during the life-time of each application.

The dynamically weighted round robin variation of the round robin algorithm was selected as it is potentially the most powerful and most flexible of the variations. Furthermore, the other variations of this algorithm discussed in Section 5.4 are simplifications of this algorithm; providing an implementation of the dynamically weighted variation effectively provides an implementation of the others. (We could simply keep all weights at 1 to provide the base round robin algorithm, and keep all weights at some other constant values to provide the weighted round robin variation.) For experimental purposes, however, we will restrict ourselves to the dynamically weighted variation.

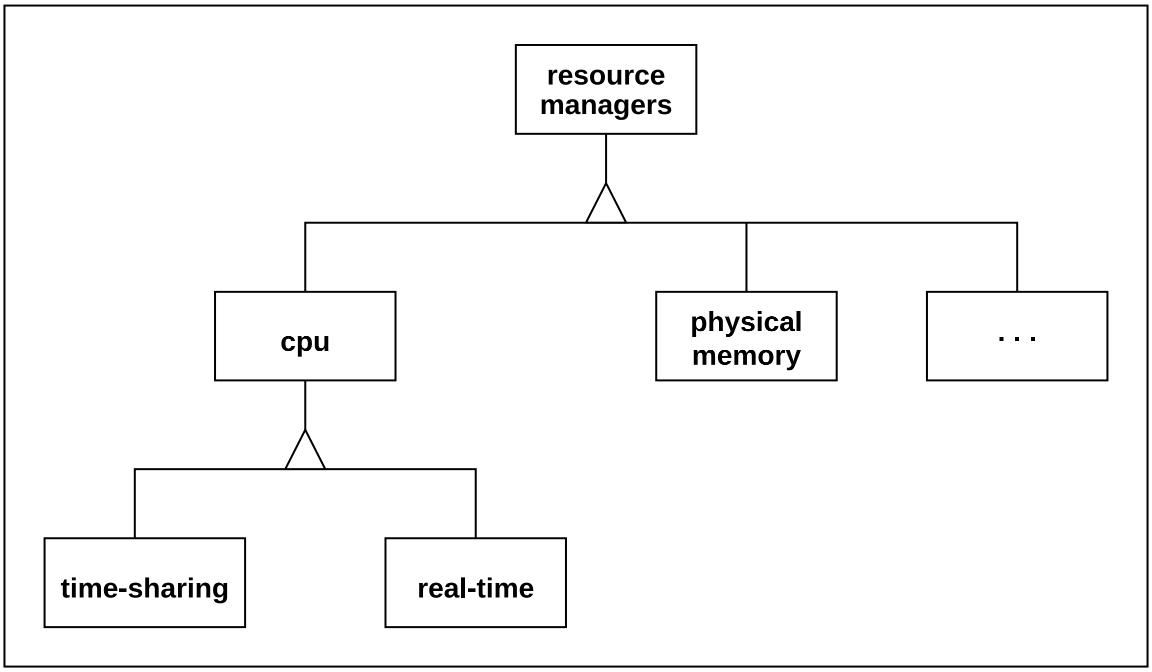
## 6.4 Resource Managers

For actually interfacing with and managing computing resources, several resource managers have been implemented. These are implemented using C and C++ and are targeted primarily at the Solaris 2.6 environment.

The resource managers are designed in an object-oriented fashion. All resource managers are derived from a base resource manager class that provides a uniform C++ API to interface with the managers. By using this base class, all resource managers are automatically placed in an enumeration as they are created, as this enumeration is needed by several resource management algorithms supported by the resource management controller. The resource management controller interfaces with individual resource managers through this API to access functionality to register and deregister processes with resource managers, disable and re-enable management for a process, adjust resource allocations, and provide information on resource usage and availability.

To date, resource managers have been implemented to manage CPU resources in a time-sharing and in a real-time fashion, and to manage physical memory resources, as shown in Figure 6.4. Adding more resource managers is a reasonably simple task, since the only requirement is that they conform to the API. This is demonstrated by the current work by other researchers in our research group developing network resource managers for use in this prototype. This includes one manager to adjust the priority of network traffic as an extension to [3] using DiffServ [18], which is nearly complete, and another to adjust bandwidth allocations using RSVP [211], which is just beginning. Further details of implemented resource managers are given in the sections below.

**Figure 6.4** OMT Diagram of Implemented Resource Managers




### 6.4.1 Time-sharing CPU Manager

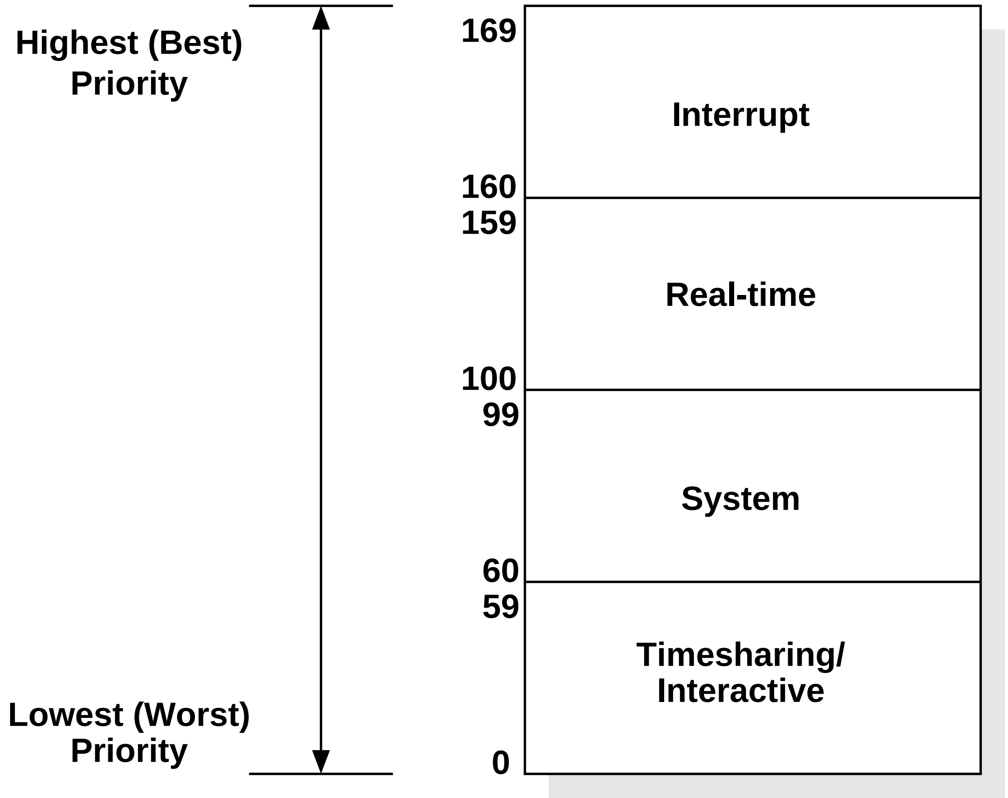
The time-sharing CPU manager was the first resource manager implemented. This resource manager is implemented using C and C++, and is contained entirely within the resource manager object in the host manager process (unlike the other resource managers implemented, as discussed below). Consequently, this resource manager exists in the user space, and does not require any kernel or system modifications to use.

To complement the CPU manager, a separate daemon has been developed that is capable of measuring the percentage of time that an application process is in a runnable state, waiting to access the CPU. This information is useful in helping to determine if CPU resources need adjustment in the metric-heuristic algorithm. However, this daemon is not used in practice as it is not usually necessary, and its accuracy can be somewhat limited. Since all other resource managers are capable of providing useful metrics to the algorithm, a simple process of elimination determines that when no other resources need adjustment, the CPU resource must require adjustment instead. This allows us to dispense with the overhead incurred by collecting this information, as it must be done for every thread in each process, as state is maintained on a per-thread basis only. Accuracy is somewhat limited since we cannot measure when a given process actually runs—when this daemon is making its measurements, it must be the process currently on the CPU, so according to its measurements, no other process is ever seen to be running. Consequently, we see an incomplete view of the processes being monitored. For example, if an application process requires a good portion of CPU resources and actually has them, this daemon will actually report that the process is in a runnable state, requiring additional resources, as its sampling will not see the time actually spent executing.

As shown in Figure 6.5, Solaris 2.6 provides multiple scheduling classes and priorities for application processes [125, 196]. Each class has its own local class priority range, which maps into some portion of the global priority range that encompasses all classes. (For example, by default, processes in the real-time class can have priorities from 0-59 within the class, which maps into priorities 100-159 globally.) Generally speaking, runnable processes with the highest global priority are allowed to execute before those with lower priorities (with exceptions, naturally). By default, all user applications start execution as time-sharing processes. Super-user privileges are required to place processes into the real-time class; processes cannot be added to the system or interrupt classes.

The purpose of the time-sharing resource manager is to control the allocation of CPU resources within the time-sharing class. It essentially works by adjusting time-sharing priorities assigned to processes using the `priocntl()` system call [184]; the existing Solaris scheduler is still in charge of actual process scheduling. When additional CPU resources are required by a process to meet quality demands, the process's time-sharing priority is increased. Likewise, when fewer resources are required, the process's time-sharing priority is decreased. The amount of adjustment necessary, as deemed by the resource management

**Figure 6.5** Solaris Scheduling Classes and Priorities




controller, determines the number of priority levels to increase or decrease. Solaris 2.6 provides 60 time-sharing priority levels by default, providing a range of possible quality levels. Further adjustments could be made to scheduling dispatch parameters (listed in [125, 196]), but these were avoided as this could affect the performance of all processes on the system, not just the ones requiring additional quality of service support.

This approach to managing CPU resources is very simple, and very efficient. Unfortunately, this simplicity comes at a cost in terms of limited scheduling capabilities. When scheduling through adjusting priorities, it is difficult to have fine grained or tight control over resources—it is difficult to say just what portion of CPU cycles a process receives with a given priority. With multiple applications with quality of service requirements, interference is an issue as well, as the CPU adjustments made to one process can affect all other processes—increasing a process's priority effectively decreases the CPU resources of others, while decreasing a process's priority likewise effectively increases the CPU resources of others. This makes providing stable quality of service difficult. Furthermore, the operating system is still free to adjust priorities of applications on its own (which it often does [125, 196]), which provides another potential source of scheduling interference.

Nevertheless, this resource manager is reasonably effective. Work on a real-time CPU manager was started shortly afterwards to overcome this manager's shortcomings. Despite having this new manager, however, the time-sharing CPU manager is still useful, as a complement to, or in conjunction with, the real-time manager.

### 6.4.2 Real-time CPU Manager

The real-time CPU manager is split into two components—an interface and a scheduling daemon. Both components exist in user space, and do not require any system or kernel modifications to use. (Once again, the daemon used to determine a process's runnable state, discussed in the previous section, could be used with this manager, but is not used in practice for the same reasons discussed earlier.)

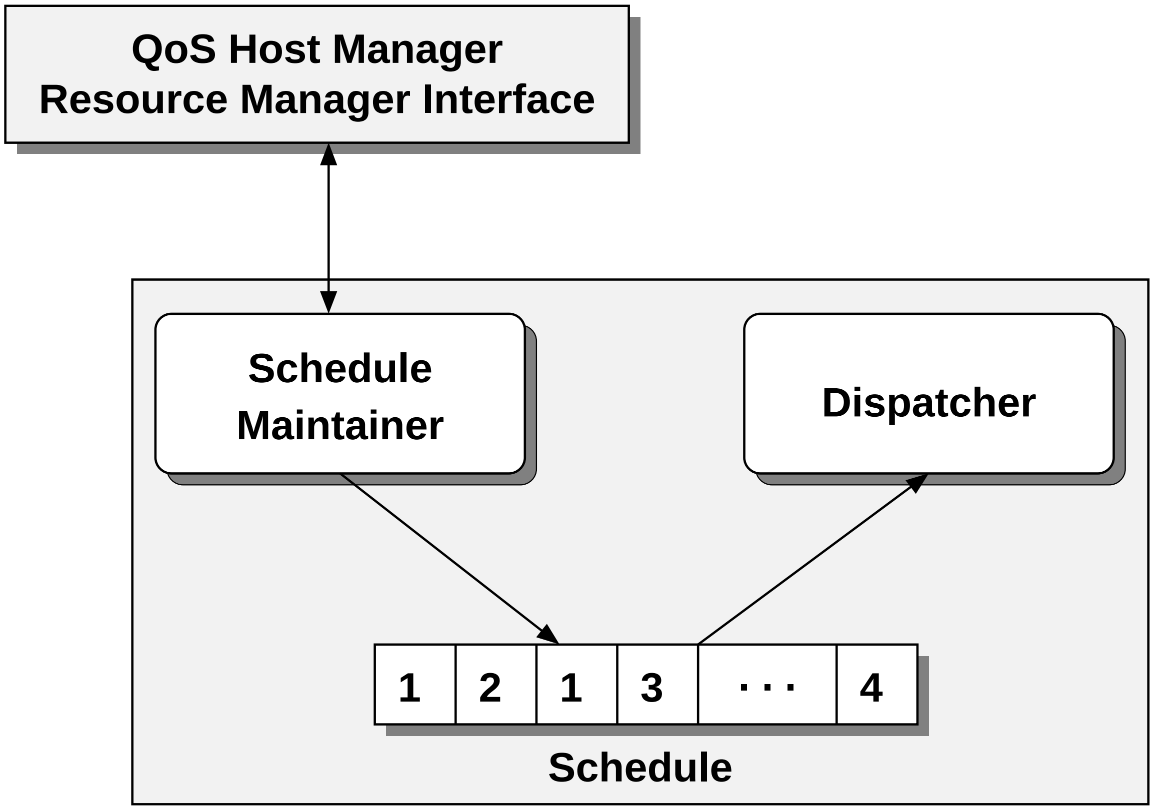
The interface portion resides in a resource manager object in the QoS Host Manager process. This interface supports the resource manager API, and relays information back and forth from the scheduling daemon. This portion is quite lightweight, and is written in C and C++.

The scheduling daemon, on the other hand, is in a separate multithreaded process. To keep the resource management process efficient, a single entity must maintain state and control the resources allocated to all processes; otherwise, the costs of coordination become too high. Since QoS Host Managers fork new processes for handling many requests, a separate single process was required to handle these activities. Consequently, a separate scheduling daemon was used, and multithreaded to support concurrent requests which are coordinated internally more efficiently. Requests from its interface in the host manager, such as registering and deregistering processes, performing resource adjustments, and so on, are received using a message queue mechanism. Feedback on available resources, resource usage, and so on, are made available to its interface using a shared memory region. The scheduling daemon is written entirely in C.

The purpose of this resource manager was to have tighter control over CPU resources than the time-sharing CPU manager, and to eliminate interference problems between allocations and adjustments to multiple application processes. In a similar approach to [33, 87], we take advantage of the scheduling priorities in Solaris to give the scheduling daemon more control over the system. To do this, we simply execute the daemon at the highest real-time priority on the system. As can be seen from Figure 6.5, this makes the daemon the highest priority process in the system, even above the operating system itself. This allows the daemon to execute essentially whenever it wants to or needs to. With the appropriate logic, we can supersede the system scheduler whenever necessary to make scheduling decisions within the daemon.

Scheduling tasks within the daemon can be essentially divided into two main components, each with its own thread of execution. The schedule maintainer component maintains the schedule of processes, based on quality feedback from the processes obtained through requests from the resource manager object interface. The dispatcher component implements the schedule by dispatching processes accordingly. This is shown in Figure 6.6, and is discussed in the sections below.

**Figure 6.6** Real-time CPU Manager




### 6.4.3 Schedule Maintenance

A separate thread of execution is used to make scheduling decisions and maintain the schedule that is used by the dispatcher. The main idea is to pull all schedule maintenance activities into this thread so that the daemon's dispatcher thread can focus on the actual dispatching of processes according to this schedule, thereby streamlining the dispatching process and improving efficiency.

The schedule itself is maintained as a table, protected by a mutex, with each slot in the table worth a certain amount of CPU time. By default, the table contains 100 slots, with each slot worth 0.01 seconds of CPU time (or 1%); these values can be adjusted at run-time as necessary. When a process to be managed is initialized, it registers with the local host manager, which passes its registration to the scheduling daemon, which allocates a default number of slots to the process. If a process is not meeting its quality of service requirements, it can be given more CPU resources by allocating it more slots in scheduling table. Likewise, a process exceeding its requirements can be given fewer resources by decreasing the number of slots allocated to it. A dedicated number of slots are reserved for time-sharing processes without any quality of service constraints; again, this value can be changed at run-time.

Schedule slots allocated to processes can be distributed throughout the schedule in a number of possible ways. By default, these slots are distributed as evenly as possible in an attempt to provide a steady level of performance throughout each second of CPU time. This is done by providing a simple mapping file which instructs the schedule maintainer how to distribute the slots of an allocation for each possible resource allocation. This file maps a given percentage of CPU allocation to a pattern used by the maintainer to produce a distribution of schedule slots. A separate tool was constructed to generate the mappings to produce even distributions. Naturally, by providing different mapping files, it is possible to cluster allocated slots together in arbitrary fashions, to minimize the number of context switches required, for example. However, most multimedia applications, which are often the focus of quality of service management experimentation, benefit from the default even distribution.

Since it could become prohibitively expensive to produce an entirely new schedule table each time a resource adjustment is necessary, the schedule is only regenerated after a certain number of adjustments have been made; again, this value can be changed at run-time. In the mean-time, adjustments are made to the schedule in the quickest manner possible, possibly resulting in a sub-optimal schedule until regeneration takes place.

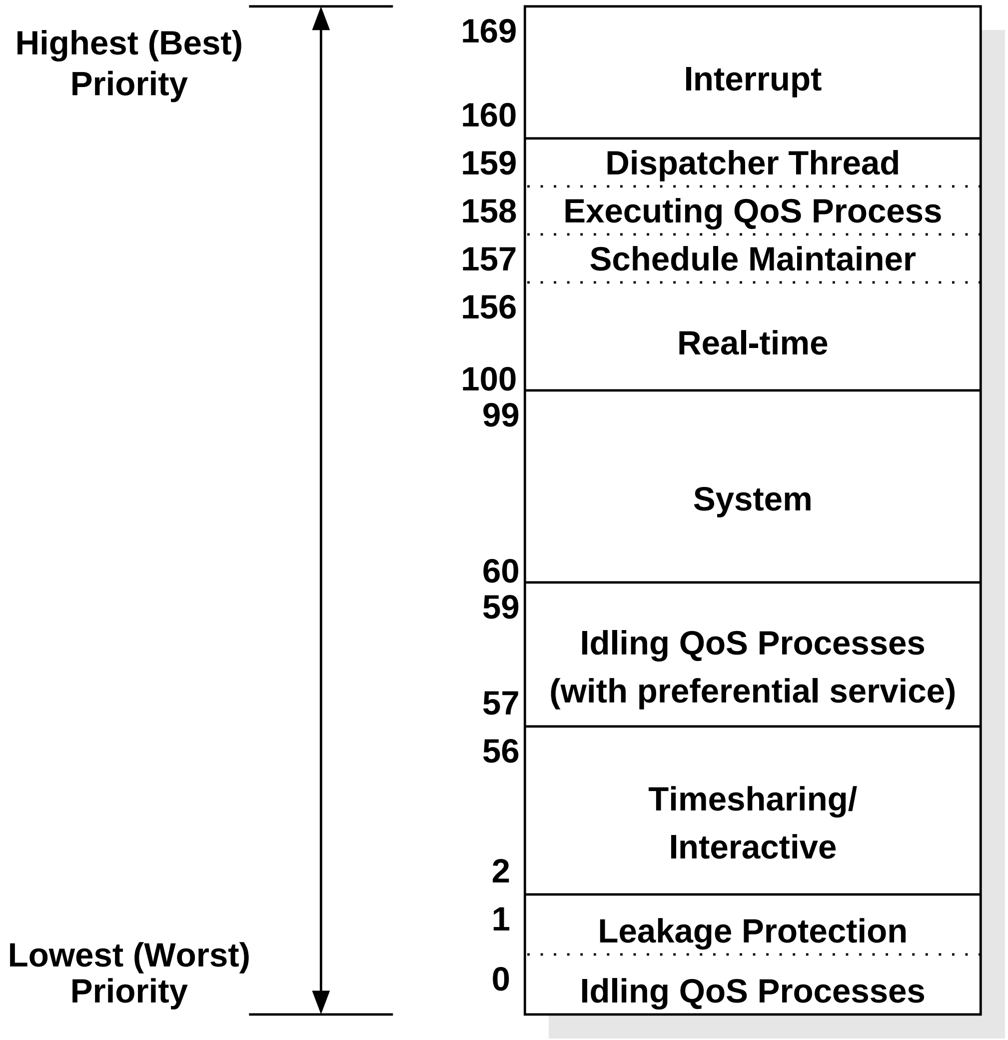
### 6.4.4 Schedule Dispatching

As described above, schedule maintenance produces a schedule table; a linear traversal of this table gives the order in which processes should be executed. Dispatching itself then is a straightforward process that does not require any further analysis or decision making. Consequently, this procedure can be highly optimized and very efficient.

Since we are not modifying the operating system itself, we are still working in conjunction with the existing scheduler and its priority system as shown in Figure 6.5. The only way processes can be scheduled are through modifications to this priority scheme and to the priorities of individual processes. Fortunately, this is more than sufficient.

We begin by modifying the system's time-sharing and real-time dispatch tables which are consulted when the system is booted to determine how the different scheduling classes should behave [125, 196]. The result is the priority scheme shown in Figure 6.7.

**Figure 6.7** Revised Solaris Scheduling Classes and Priorities




In this new priority scheme, no new scheduling classes have been added. Instead, the time-sharing class has been shrunk somewhat, and the real-time class has been split and dispersed throughout the global priorities. The highest real-time priority (159) is dedicated to the scheduler dispatch thread, as discussed above. The third highest real-time priority (157) is dedicated to the schedule maintainer to ensure that it can run in a timely fashion (if it cannot, it can potentially delay the dispatch thread when the schedule is being updated, which could affect quality of service). The second highest real-time priority (158) is reserved for executing quality of service processes. Thus, to dispatch a process from the scheduling table, we merely modify its priority and have the dispatcher sleep through the time slots allocated to the process. Since the dispatched process is now the runnable process with the highest global priority, it can execute as long as it remains runnable. When its time has expired, the dispatcher thread awakens. Since it now the runnable process with the highest global priority, it effectively interrupts the executing process and is ready to dispatch the next process requiring quality of service support. To deschedule the current process, the dispatcher thread sets its priority to the lowest real-time priority, which has been mapped to global priority 0, below time-sharing processes. Thus, processes descheduled by the dispatcher only execute when the scheduling daemon, operating system, and all time-sharing processes cannot.

To handle schedule table slots reserved for time-sharing processes, the dispatcher merely does not dispatch any processes to the second highest real-time priority. Consequently, when it puts itself to sleep, the next highest priority processes to execute are the time-sharing processes, still being scheduled by the system scheduler. When the time-sharing slot is done, the dispatcher thread wakes up and resumes control. Time-sharing processes also make use of CPU cycles leaked when the executing dispatched process blocks (for I/O, for sleeping, and so on).

A few other priority slots in the global priority table have been modified for further use by the scheduling daemon:

- At times, it might be harmful for a process to exceed its quality of service requirements, and it becomes important to ensure this does not happen. In such cases, a leakage protection thread is run at the second lowest real-time priority, which has been mapped to global priority 1. This lies below time-sharing processes, but above descheduled processes. By simply executing an endless loop at this priority level, no extra CPU cycles can fall down to descheduled processes. Consequently, they only execute when dictated to do so by the dispatcher thread. Since this can waste precious CPU cycles, and this protection is not necessary for the applications we are primarily concerned with, this thread is not run by default.

- Under cases of overload, it is possible under the above priority scheme that time-sharing processes with no quality of service constraints achieve better service than those processes with such constraints. This is typically the case when multiple processes divide available CPU cycles amongst themselves, and when these applications block frequently (which can be the case with I/O bound multimedia applications). In such cases, a running time-sharing application can obtain both the time slots dedicated for time-sharing applications and free cycles left from dispatched processes blocking;

the sum of which can exceed the amount of CPU time dedicated to a single process with quality of service needs. To resolve this issue, the dispatcher can optionally deschedule dispatched processes with quality of service constraints to global priorities above the time-sharing class. Consequently, these descheduled processes will be able to reap the CPU cycles from blocked dispatched processes, instead of time-sharing processes. This option can be set at run-time as well.

### 6.4.5 Physical Memory Manager

The physical memory manager is split into multiple components—an interface and memory management daemons. All components exist in user space, and do not require any system or kernel modifications to use.

The interface portion resides in a resource manager object in the host manager process. This interface supports the resource manager API, and relays information back and forth from the memory management daemons. This portion is quite lightweight, and is written in C and C++.

The memory management daemons, on the other hand, are in separate multithreaded processes. To keep the resource management process efficient, a single entity must maintain state and control the resources allocated to all processes; otherwise, the costs of coordination become too high. Since QoS Host Managers fork new processes for handling many requests, a separate single process was required to handle these activities. Consequently, separate daemons were used, and multithreaded to support concurrent requests which are coordinated internally more efficiently. Requests from its interface in the host manager, such as registering and deregistering processes, performing resource adjustments, and so on, are received using a message queue mechanism. Feedback on available resources, resource usage, and so on, are made available to its interface using a shared memory region. The memory management daemons are written in C, with some assembly language code where necessary.

The general purpose of this resource manager is to provide better management of physical memory for processes requiring quality of service considerations. Since we are not modifying the operating system, we must work with the existing memory management system, and take advantages of its facilities wherever possible. Before exploring the details of the physical memory manager, we first give a brief overview of memory management in Solaris 2.6.

#### 6.4.5.1 Existing Solaris 2.6 Memory Management

Solaris 2.6 employs two-handed clock selection [48] for guiding memory management decisions [125, 196]. This approach treats all resident pages exactly the same, regardless of which processes currently own them. In this approach, the system page table is logically arranged as a circular list. A virtual clock hand scans through the circular list, clearing referenced bits for pages on its way. A fixed distance behind, a second hand follows the first hand, checking the referenced bits. If the referenced bit for a page is still clear when the second hand reaches it, it has not been accessed recently, and is eligible for eviction. Otherwise, it has been accessed recently, and should remain in memory, if possible.

In Solaris 2.6 and later, the system page table is structured as a linked list of contiguous memory segments, to support the addition of memory to live systems. For two handed clock selection, a kernel process pageout calls interface functions to this linked list to traverse the page table and advance the hands of the virtual clock. The first hand clears hardware referenced bits maintained by the memory management unit; the second hand later synchronizes the hardware referenced bits (which may now be set once again) with those in the corresponding page table entries, to have the page table reflect the current hardware state. The second hand then checks these bits to decide if a page is eligible for eviction, according to the standard two handed clock approach. Eligible pages are added to a list of "free" pages, but are only actually evicted when another page needs to be paged in to handle a page fault. Pageout only runs when the size of the free list dips below system defined thresholds; it spends the rest of its time asleep, without carrying out two handed clock selection.

The exception to the above procedure is when pages are locked or pinned into memory. Such pages are designated as being unpageable, and are always resident in physical memory. Locked pages are essentially ignored by pageout when conducting page table scans. Typically, only vital kernel pages, such as the page table itself, are locked into memory. However, it is possible for processes with super-user privileges to do so as well, using library functions such as mlock(), mlockall(), and plock().

#### 6.4.5.2 Implementation Overview

Memory management in Solaris 2.6 is quite efficient, due to both its underlying clock selection approach, and its implementation. However, the two handed clock approach is not the best algorithm at minimizing page faults. Furthermore, since it treats all pages equally regardless of process ownership, this approach is not suitable for quality of service management, where the needs of processes must be examined individually to meet requirements.

The working set approach [42], however, is better suited to meeting the requirements of individual processes, but incurs far too much overhead to be used on its own. A combination of the two approaches is necessary for quality of service to be effectively supported.

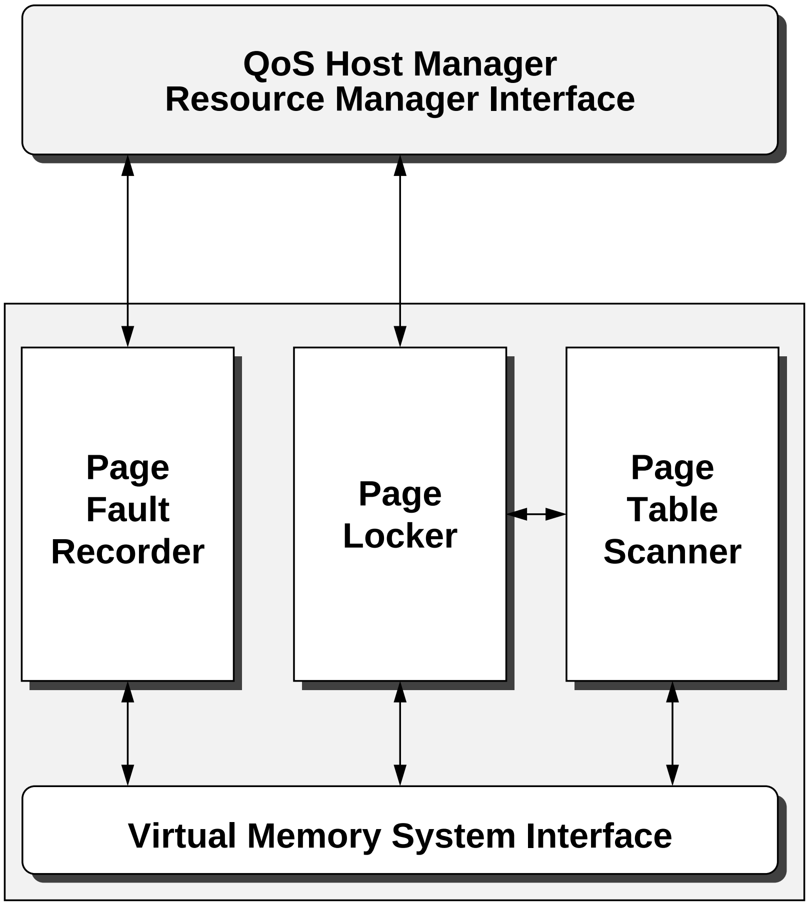
The key to this combination lies in the ability of Solaris to lock pages of processes into memory and make them unpageable. By selecting and locking critical pages of processes into physical memory, we can effectively augment the paging decisions already being efficiently made by the system. To determine which pages are most important to processes at any given time, we can collect page reference information from kernel virtual memory management structures, including the system page table. This information need only be collected for processes with quality of service requirements, and those that are important to the user (determined through user hints, as discussed in Section 6.1.2.3), which is a small fraction of the processes on a system. The collection of this information can be done at a low priority in the background, so that little overhead is generated. Based on this information, we can estimate the working sets of processes, while the system is still paging efficiently using two handed clock selection.

When a process is not meeting its quality of service requirements, and it is either page faulting heavily or memory is scarce on the system, increasing its memory resident size will likely alleviate the problem. Deducing the process's working set, we can lock in memory those pages that are most needed and used by the process to reduce its page faulting, and improve its quality of service. Likewise, we can unlock pages when they are no longer in a process's working set or when quality of service requirements are exceeded. As a result, we can effectively adjust physical memory resources to change quality of service. In doing so, we can ensure that processes have access to the memory they need when they need it on an individual basis, that memory is made available to other processes when no longer needed, and that this can be done in an efficient manner with little overhead, thereby meeting our quality of service requirements.

#### 6.4.5.3 Implementation Details

The physical memory manager to support the above extensions to Solaris 2.6 memory management makes use of two daemon processes—one that records page fault rates, useful in diagnosing memory problems, and another that actually locks and unlocks physical memory pages on demand. The memory manager daemons must run with super-user privileges to get the necessary kernel access for retrieving process and page table information, locking and unlocking pages of memory, and a variety of other tasks. The structure of the physical memory manager is shown in Figure 6.8. Its various components are discussed below.

**Figure 6.8** Physical Memory Manager



- **QoS Host Manager Resource Manager Interface:** As mentioned earlier, the interface to the memory manager daemons lies in a resource manager object in the local host manager. When quality of service problems are reported to the host manager, and the resource management controller determines that memory resources must be adjusted, this interface is used to contact the page locker to make it so. When the metric-heuristic algorithm is being used, the resource management controller uses this interface to contact the page fault recorder to retrieve the page fault rate for the process in question, and determine if memory should be adjusted in the first place.

- **Page Fault Recorder:** For each process registered with the host manager, the page fault recorder measures its page fault rate over time as directed by the resource management controller. This is done by retrieving resource usage statistics for the processes stored in the /proc file system on Solaris systems. The page fault rate of an application cannot be measured directly, however. The page fault recorder can only retrieve the number of page faults that a process has incurred, and must determine the rate over time itself. In the future, statistical filtering and trend analyses could also be applied here to acquire additional information, if the resource management controller requires it in making its decisions.

  The page fault recorder also collects other related metrics used in determining the source of quality of service problems in the metric-heuristic algorithm. These include the global page fault rate for the entire system, the percentage of CPU time dedicated to I/O and swap activities (paging can account for a very large portion of I/O activities on the system), and the number of free virtual memory pages.

- **Page Locker:** The page locker is in charge of locking and unlocking memory pages to adjust physical memory resources as instructed by the resource manager interface. The amount of change in memory resources required is determined by the resource management controller in the host manager and is communicated through the resource manager interface whenever adjustments are required. To determine which pages should be locked or unlocked in particular, the page locker consults the page table scanner.

  The page locker also maintains accounting information on which pages are locked, how many pages of lockable memory are available in the system, and so on. The Solaris virtual memory management system already maintains similar information, but does so with too many restrictions for our needs. For example, as virtual memory is allocated to processes, Solaris reduces the number of pages it considers lockable, to avoid page faulting in cases where too many pages are locked, or the locker is making uninformed locking decisions. Consequently, under ordinary conditions, Solaris would refuse many of the page locker's lock requests unnecessarily.

To overcome these problems, the page locker updates Solaris virtual memory management accounting structures so that Solaris is unaware of the page locking it is carrying out. Consequently, the page locker must also keep lock accounting information of its own to ensure that locking is done in a responsible manner. This includes ensuring that there is always a sufficient supply of pages on the system that are never locked to prevent physical memory from being completely filled with locked pages, which would effectively wedge the system. With this done, the page locker can carry out its task of locking and unlocking pages unhindered.

- **Page Table Scanner:** The page table scanner is responsible for traversing the system page table to collect information on processes that have registered to the local host manager as having quality of service requirements needing to be enforced. As the page table scanner must run continuously, it is executed in a separate thread with a low priority with a tunable delay loop between traversal actions.

  As the page table is traversed, the scanner determines if each page is owned by a registered process. This check can be difficult as the Solaris virtual memory system was not designed to support it. Tracking page ownership in page structures would be complex and expensive, as it is possible for pages to be owned and used by several processes, leading to extra memory overhead and time-consuming maintenance procedures. Since the two handed clock selection approach used in Solaris does not consider page ownership in its eviction decisions anyways, this information is not included in paging structures. Instead, page structures keep `vnode` references that interface with the file system to name memory objects and assist in paging operations to disk. By keeping track of `vnode` references contained by the various segments of a process's address space, we can match page `vnode` references while scanning the system page table and indirectly tie pages to the processes that own them.

  If a page is not owned by a registered process, it is ignored. Otherwise, the value of the page is calculated; the higher the page value, the more likely the page is in the current working set of the process, and the more valuable the page is to be locked into memory. Similarly, when pages need to be unlocked, those with the lowest value are selected first. As a page remains resident in physical memory, its value increases by a fixed amount depending on its free list and referenced bits. If a page's free list bit is set, the page has been marked for eviction and is currently on the free list. This means that the page is currently not very valuable to the process, and the page value maintained by the page table scanner should increase by only a small amount. If a page's free list and referenced bits are both clear, the page is more valuable to the process than the previous case because it has been referenced after those on the free list, but not recently. As a result, the scanner should increase the page's value by a moderate amount. If a page's referenced bit is set, the page has been accessed recently, and is quite valuable to the process, so its value should be increased by the scanner more than either of the two previous cases. When a page is no longer found to be resident in physical memory, it has been paged out due to inactivity, so its value is decreased by a fixed amount by the scanner accordingly. Likewise, if a page has been locked into physical memory, its value is decreased by a fixed amount unless the page has been referenced. This is to prevent pages locked into memory from remaining locked when they are no longer being used, allowing the memory to be freed up for more valuable pages. As necessary, the page table scanner consults a number of process address space structures to augment the information collected from the system page table.

This approach to evaluating page values has several benefits. First, by calculating page values in this way, we can closely approximate the working set of a process at a relatively low cost, allowing the page locker to make more accurate locking and unlocking decisions. Second, several of the parameters used in carrying out these calculations can be tuned at run-time to adjust page evaluation as necessary. Third, the scanning process itself can be tuned to adjust performance and overhead as necessary. For example, if there is an abundance of physical memory available on the system, it is less likely that quality of service problems will result from memory issues, so the scanning process can decrease or halt entirely to conserve CPU cycles and other resources, much in the way pageout already does on the system.

- **Virtual Memory System Interface:** The virtual memory system interface provides the memory manager daemons with the mechanisms needed to interact with the underlying operating system. This includes kernel reading and writing access, as well as system calls to access kernel functions.

  Since user-level processes, such as the memory manager daemons, can only call functions residing in the kernel through a system call, functions without a system call interface defined in the kernel's system call table cannot be called. Unfortunately, there are many such functions that the daemons need to call, to scan the page table (page_first(), page_nextn()) and lock memory in other address spaces (as_ctl()), for example. In these cases, system call interfaces for the functions are added at run-time to the system call table in unused slots, to allow the virtual memory system interface to access them. Assembly code is used to complete the illusion of the new system calls.

The virtual memory system interface also hides the details of the underlying operating system, which can help facilitate future efforts in porting the physical memory manager to other platforms. Such efforts are currently under way.

## 6.5 Enforcement Policies Implemented

Several sets of enforcement policies are implemented for quality of service support. In most cases, these policies are provided to assist in situations when resource management algorithms alone cannot meet quality of service requirements, due to overload conditions or resource shortages. In all cases, enforcement policies are implemented as sets of CLIPS rules. In this section, we discuss these enforcement policies.

### 6.5.1 Admission Control

At times, administrators may need to impose admission control, as discussed in Section 2.1.2.4, to prevent the computing environment from becoming saturated with user applications. This saturation could potentially lead to overload situations and degraded quality of service delivery. To prevent environment saturation, a set of admission control rules have been developed.

When in place, admission control rules are applied whenever a new application starts and registers with its host manager. These rules check to see if there are sufficient resources available to support this application, based on the estimated resource needs of the application provided in the rules. Different applications may use different rules to allow resource needs to be tailored to individual applications.

If admission control rules indicate that the application can be safely supported, the application proceeds with normal managed execution. If the rules, however, indicate that the application cannot be safely admitted to the environment without jeopardizing quality of service, several options are available. As alternative scenarios, the application may be forced to terminate and not execute at all, or the application may be permitted to execute with reduced quality constraints, or as a best effort application with no quality constraints whatsoever. Which method is selected may depend on the application and the conditions in the computing environment when the admission control rules are applied.

For example, suppose we once again have a client-server video streaming application, and an admission control policy stating that if any resource has less than 5% of its capacity available for allocation, no new applications are permitted to execute. If the video client attempted to start when only 2% of CPU resources were available, the QoS Host Manager would instruct the termination actuator of the client to terminate as soon as the client registered with the manager. In doing so, admission to the system is effectively controlled.

Because CLIPS rules are used to encapsulate admission control functionality, admission decisions can be tuned at run-time, or possibly even disabled or re-enabled, depending on the situation. This allows a great deal of flexibility in regulating access to the computing environment.

### 6.5.2 User Hints

To support the handling and processing of user hints, as discussed in Section 6.1.2.3, additional rules are required to shift resources away from applications that are ignored by the user to applications that are the focus of attention. Additional rules are needed to handle the propagation of user hints when the applications in question are distributed across multiple hosts.

When a user hint is detected by a user hint sensor, the local host manager is informed by the process's coordinator, and facts about the hint are placed in the fact base. The inference engine is then run to determine which rules should be acted upon to handle the user hint. If the facts indicate that the process is no longer important to the user, the process is directed by the host manager to deactivate its instrumentation, except for user hint sensors. This effectively suspends quality of service expectation policies for the process, as violations are no longer detected and reported. If this was not done, the process would likely complain bitterly as its resources are shifted to other processes and its expectation policies are no longer met as a result, imposing an additional burden and overhead on the host manager. All resource managers are then instructed to free up resources allocated to the process, but track the prior allocation for later reinstatement, if the process regains the user's attention. By default, the resource managers effectively place the process back among other best effort processes without quality of service constraints; this behaviour can be easily changed at run-time to only scale back resource allocations, or something else more appropriate if necessary.

If facts, on the other hand, indicate that the process is important to the user once again, the opposite needs to occur. Resource managers are instructed to reinstate resource allocations to the process at prior levels. If this is no longer possible, as these resources could now be used elsewhere, the resource managers need to allocate resources as closely as possible to these levels. Once this is done, instrumentation in the process is reactivated, essentially reinstating previously disabled expectation policies. The process then resumes normal operations.

If facts indicate that the process in question is part of a distributed application, we may wish to forward the hint to other component processes of the application. To do so, the domain manager responsible for the host on which the process is executing is notified of the user hint. The domain manager locates other parts of the application and notifies their host managers of the user hints. The remote host managers now carry out the same actions as described above to handle the hint, as if it originated from one of its own managed processes.

For example, suppose we once again have a client-server video streaming application, and a user hints policy stating that any applications not fully visible to the user receive only best effort service. When the video client starts, it is at the top of the window stacking order and is completely visible. Consequently, it is eligible for preferential resource management to improve quality of service. If the user then launches a new application and covers the video client window, a user hint sensor in the client would detect this, and its coordinator would forward a report to the QoS Host Manager. The QoS Host Manager would then act on the user hints policy and revert the application to best effort service by removing its resource allocations and disabling all sensors in the client, except for the user hint sensor. If the application was later uncovered, the user hint sensor would recognize this, its coordinator would notify the QoS Host Manager, and this manager would re-enable all client sensors and restore the client's former resource allocation as best as possible.

Naturally, since user hint processing is handled by CLIPS rules and facts, user hint policies can be tailored at run time. Furthermore, user hint processing can be customized according to the application in question, which can be quite important, for reasons discussed in Section 6.1.2.3.

### 6.5.3 Service Differentiation

At times, there are simply insufficient computing resources to keep up with user demands for quality of service. In such cases, no amount of resource management will be able to satisfy all quality of service requirements. Acquiring and installing additional resources helps alleviate the problem somewhat, but in reality only delays the inevitable.

When not all quality of service requirements can be satisfied at once, difficult choices need to be made by those administrating the environment. Should all applications receive equal service, which could be below requirements, or should some applications receive preferential service to meet their requirements, while others suffer more harshly? If the latter route is chosen, how would the application to favour be selected? What would happen to the applications not selected? These are administrative decisions, which could depend on the organization, user population, application base, time of day, and any number of other factors.

The above decisions can be formalized as enforcement policies. These policies are implemented as rules used by the CLIPS engine to determine how to handle overload conditions. Two types of service policies have consequently been implemented, as discussed below.

#### 6.5.3.1 Uniform Service Policies

By default, a uniform service policy is employed, giving all applications equal opportunity for service. This means that it is possible for no applications to meet their expectation policies in some situations. This policy is fair in the sense that each application has equal opportunity in obtaining resources.

Currently, the policy used is a simple first-come, first-serve policy—applications that complain about quality of service first get additional resources first. When the supply of resources is exhausted, all further complaints are ignored until resources are freed up (through applications terminating, etc). Other uniform service policies are possible, including scaling back resource allocations to each application equally to free additional resources, and reverting every application to best effort service, until the situation can be resolved.

For example, suppose we once again have a client-server video streaming application, and are using a uniform service policy to allocate resources on a first-come, first-serve basis. If sufficient resources exist for all applications, the video client will be able to meet quality of service requirements. If there are insufficient resources, the video client may be able to meet quality requirements if it requests additional resources before other executing applications; if it is too late, it may be unable to meet its requirements.

#### 6.5.3.2 Differentiated Service Policies

Differentiated service policies provide preferential service to certain applications, at the cost of reduced service to other applications. To date, the inference engine can provide differentiated service to applications based on the type of application, the user executing the application and the time the application was started. Other possibilities would be to base decisions on the amount of resources each application consumes, the estimated finishing time of the application, etc. Depending on the facts on the application in the fact base, the inference engine will determine which applications to favour, and which applications should suffer as a result.

Currently, applications not selected for preferential service are either reverted back to best effort service, or have their resource allocations scaled back to free resources for favoured applications. Other possibilities would be to suspend or terminate the applications, or ask the applications to renegotiate expectation policies or adapt their behaviour to meet current expectation policies using fewer resources.

For example, suppose we once again have a client-server video streaming application, and are using a differentiated service policy where video applications are favoured over all other applications, and applications that are not selected for preferential service revert back to best effort service. If the supply of a computing resource is exhausted while the video client executes, the client will be favoured by the QoS Host Manager according to the differentiated service policy, and other applications would be reverted back to best effort applications as was done in the previous section, to free resources for the favoured video client. With these additional resources, the video client would then be able to meet its quality of service requirements.

Differentiated service policies only come into effect when the system's supply of a computing resource becomes exhausted. As long as there are sufficient resources, we do not need to choose applications to favour over others. It is important to note, however, that resources can be exhausted in a transient fashion, as part of the resource allocation process. For example, in attempting to find the resource allocation to meet quality of service expectation policies, a process could be temporarily over-allocated resources, and subsequently determine that its allocation is excessive and free them up at this point. In this temporary condition, however, it is possible to run out of the over-allocated resource. Since the situation is temporary, we do not likely want differentiated service policies to be invoked as we do not really need to discriminate against applications at this point. Consequently, differentiated service policies can be structured so that they are only invoked when conditions of resource exhaustion persist for a period of time, allowing these transient conditions to come and pass without taking drastic actions.

## 6.6 Summary

In this chapter, we discussed the various implementation details of our quality of service resource management prototype system. In the next chapter, we present the results from experimentation with this prototype system to date.

# Chapter 7

# Experimental Results and Experience

In the previous chapter, we discussed the details of prototype implementations of our quality of service resource management architecture and algorithms. In this chapter, we evaluate these prototypes through a series of experiments. We begin by providing an overview of our approach to experimentation in Section 7.1. Experiment results are detailed in Section 7.2 through Section 7.7. This chapter concludes with a summary of key results and observations in Section 7.8.

## 7.1 Overview

Before presenting and discussing results from experimentation with our quality of service management prototype, it is important first to discuss some of the background behind the experiments. In this section, we discuss the goals behind our experimentation, our experiment methodology, criteria used in our experiments, and the environment in which experiments were carried out.

### 7.1.1 Experiment Goals

There are three primary goals in experimentation for this work:

1. Demonstrate that significant improvements in the delivery of quality of service can be achieved using our approach and algorithms at little cost. We will discuss the evaluation criteria used to evaluate these improvements and cost in Section 7.1.3.

2. Compare the capabilities and performance of selected resource management algorithms from Chapter 5: metric-heuristic (MH), multidimensional interpolation search (MS), and dynamically weighted round robin (RR) under a variety of operating conditions.

3. Compare the above resource management algorithms under a variety of enforcement policies dictating various broader administrative goals, such has handling overload conditions caused by resource shortages.

By completing these goals, we hope to establish our approach as a leading solution to the quality of service resource management problem, and to determine which approaches to resource management work best under which circumstances. This understanding, along with additional insights gleaned from experimentation will help guide further work in this area in the future.

### 7.1.2 Experiment Methodology

There are two basic approaches to experimentation to reach the goals set in the previous section. One approach is to start with simple experiments, and incrementally work towards more complex experiments. The methodical nature of this approach makes results easier to compare, interpret, and understand, but may require more experiments to draw significant conclusions. A different approach would be to start with the most complex experimental scenario, and perform additional experimentation to investigate results that are incomplete, inconsistent, or difficult to explain. This approach may require less experimentation to uncover interesting conclusions, but tends to be more difficult to understand and may inadvertently lead to unproductive experimentation in an attempt to explain results.

In this work, we take the former approach. While this may require additional experimentation, it keeps the process simpler and easier to understand. It also allows us to identify potential problems in our experiments more easily, so alternatives can be selected. The general procedure taken in experimentation is as follows:

1. Perform initial experiments to validate the prototype system and algorithms to ensure that they are working correctly, and to gather preliminary data useful for initializing policies.

2. Start more rigorous experiments with the simplest possible experiment configuration: a single application on a single host with a single resource being managed. This experiment will be executed four times, once without any management support, and

once for each of the three resource management algorithms being studied. Each experiment must also be replicated.

3. Each of the restrictions set on this initial condition are individually relaxed in subsequent experiments: first allowing multiple resources, then multiple applications, and finally allowing multiple hosts. This leads to $2^3 = 8$ possible experiment configurations. Once again, each configuration is repeated under each management scenario and with replications, as noted in the previous step.

4. The above step leaves only one experimental goal from the previous section unaddressed—comparing the algorithms under a variety of enforcement policies. Four sets of enforcement policies will be used in experimentation: admission control, user hints, service differentiation, and a special set providing these three enforcement policies all at once. Once again, each configuration is repeated under using each resource management algorithm, and with replications.

### 7.1.3 Experiment Evaluation Criteria

In comparing the results of experiments to one another, to evaluate the various quality of service resource management solutions, we need to measure both the effectiveness of the solutions, as well as the costs incurred by the solutions. In the experimentation in this work, the effectiveness of the solution in place is measured in terms of mean performance, convergence, and stability, while the costs incurred are measured in terms of resource overhead. In this section, we discuss the criteria used in these different measures of evaluation. (Reasons behind the selection of values used in these criteria are discussed in Section 7.2.3.)

#### 7.1.3.1 Mean Performance

Mean performance is simply a measure of the average quality of service delivered by an application process. It is computed by taking the mean of quality measurements taken from the application process, usually in terms of the attributes used to express quality of service requirements in expectation policies for the process. Since we are experimenting with video applications, this will be the mean frame rate delivered by the application processes, measured in frames per second (FPS).

#### 7.1.3.2 Convergence

Convergence refers to the amount of time required for an application to achieve and maintain its desired quality level according to its expectation policies. The more quickly convergence occurs, the better the quality of service management solution. If convergence does not occur at all, or occurs very slowly, then the management solution in place is not suitable for ensuring the delivery of quality of service.

In experimentation for this work, we consider that an application has converged after satisfying quality requirements for 10 consecutive quality samples. Any missed requirements in this time period are indications that quality has not yet completely converged. To facilitate comparisons across experiments, convergence time is measured in terms of the percentage of an application's execution before quality of service delivered converges to meet requirements. Raw performance is also reported to show where quality converged in the acceptable range of quality dictated by expectation policies.

#### 7.1.3.3 Stability

Stability refers to the level of consistency in delivered quality after convergence occurs. If there is considerable variation in results, and quality delivered is not stable, the end user will not be satisfied with results, and the management service is not performing acceptably, even if quality meets requirements on average. On the other hand, if the level of quality delivered by the application is steady, and meets quality of service requirements, the quality of service management solution in place is doing a much better job.

For this work, we say that the quality delivered by an application is stable if, after converging, quality continues to meet requirements unless a disturbance occurs. (For our purposes in this work, a disturbance is defined as any event that causes quality to deviate from requirements after convergence has occured.) If a disturbance occurs and quality is no longer met, then quality of service requirements must be met again within 10 samples from the point of disturbance. Otherwise, quality of service is considered to be unstable. The level of stability provided is then further measured in terms of the percentage of quality samples that violated quality of service requirements (the number of frame rate measurements violating requirements divided by the total number of measurements taken) and the deviation in quality of service incurred after convergence (provided in terms of both standard and mean absolute deviations).

#### 7.1.3.4 Overhead

Overhead is measured in terms of resource consumption by our quality of service management system. While an interface exists to query memory usage, I/O requests, and a variety of other valuable metrics, this information is not actually provided by the Solaris 2.6 operating system. Consequently, we had to rely solely upon CPU consumption, normalized as the percentage of CPU time consumed by management throughout an experiment's execution.

### 7.1.4 Experimental Environment

To facilitate the execution of our experiments, we developed a quality of service management testbed for this work. The purpose of this testbed is to provide a tightly controllable, highly portable, and flexible environment suitable for experimentation in this area. Despite the progress in quality of service management research outlined in Section 2.2, experimental tools to support this work are not generally available. Consequently, we needed to construct these tools ourselves.

In this section, we briefly overview our quality of service management testbed, and delve into some of the details that are particularly relevant to this work. Further details of the testbed can be found in [93].

#### 7.1.4.1 Testbed Architectural Overview

The quality of service management testbed architecture shown in Figure 7.1 is the basis for our experimentation. This architecture consists of the testbed itself, a distributed computing environment containing load generators, environment components, and application processes whose quality of service is to be analysed, and testbed applications that make use of the testbed. These components are described below.

#### 7.1.4.2 Testbed Components

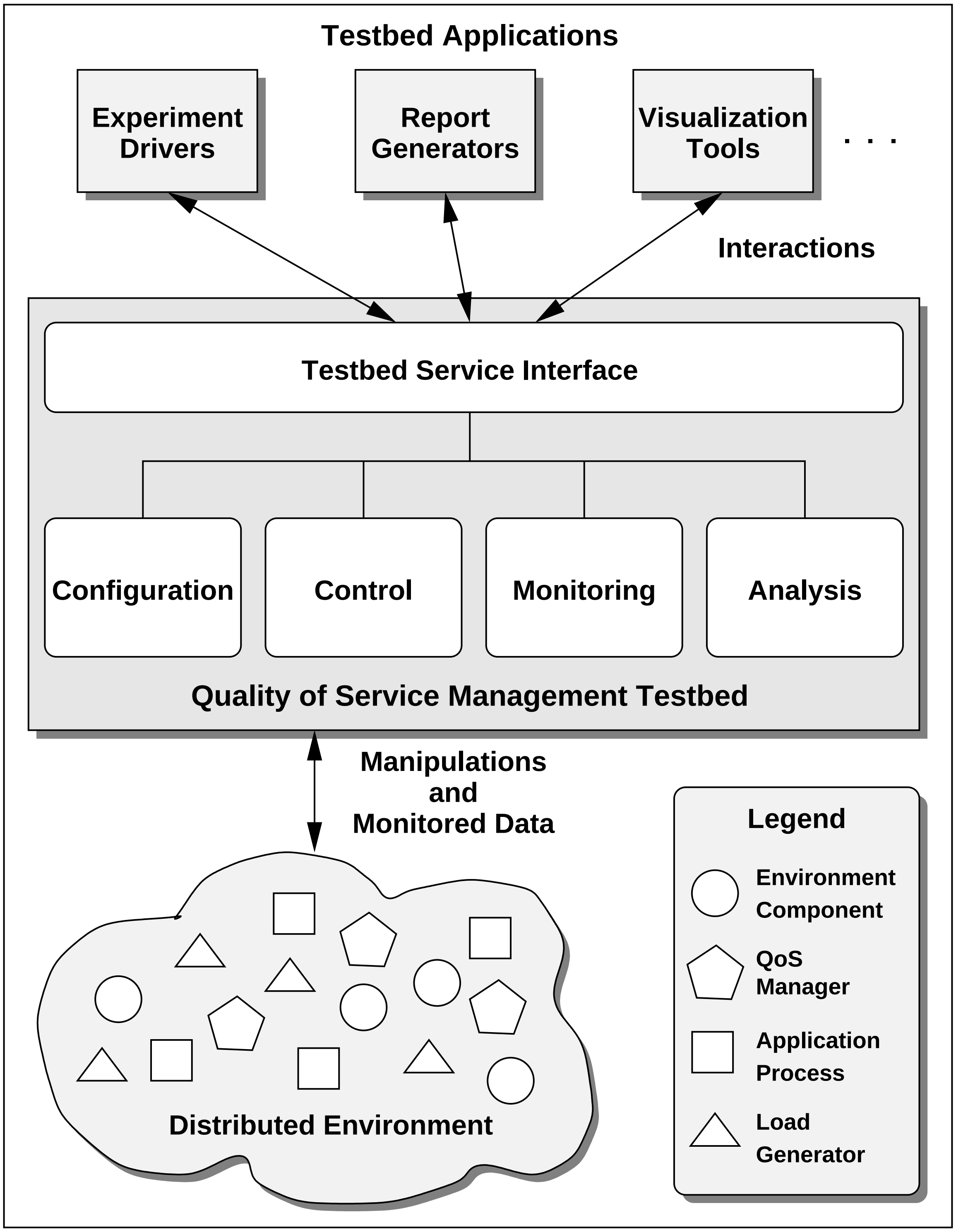
The testbed consists of a set of core components, as follows:

**Testbed Service Interface:** The services provided by the testbed are accessed by testbed applications through the testbed service interface. This interface coordinates activities within the testbed by routing requests, replies, and reports to the appropriate components.

**Configuration:** The configuration component is responsible for maintaining information on the configuration of the distributed environment and applications executing in it. Areas of responsibility include:

- Deployment — where environment components, application processes, and load generators are located in the distributed environment.

**Figure 7.1** Quality of Service Management Testbed Architecture



- Relationships — how environment components, application processes, and load generators are related to one another.
- Workload population — what kind of interactions occur between environment components, application processes, and load generators, what information is exchanged in each interaction, and how often these interactions occur.

**Control:** The control component provides a basic service for controlling experiments. It uses deployment and relationship information from the configuration component and provides operations to:

- Start an experiment and allocate the resources required, based on the defined configuration.
- Finish an experiment and deallocate experiment resources.
- Activate environment components according to their configuration.
- Deactivate environment components.
- Launch application processes and load generators according to their configuration.
- Terminate application processes and load generators.

**Monitoring:** The monitoring component collects the following information about environment components, application processes, and load generators in the distributed environment:

- Operational status information.
- Timing information, including responsiveness, throughput, and so on.
- Utilization statistics on how each environment component is used.

**Analysis:** Using configuration information and data collected from experiments, the analysis component can compute:

- Basic measures, such as totals, maximum values, minimum values, means, medians, and modes.
- Variability measures, such as standard deviation, sample variance, and skewness.

These functions can be applied to individual environment components or groups of components. More sophisticated analyses, such as regression analysis and trend computations, can be performed by higher level testbed applications.

#### 7.1.4.3 Distributed Environment

The distributed environment shown in Figure 7.1 contains a variety of environment components, quality of service managers, application processes, and load generators. Environment components consist of network and end system hardware resources used by application processes and load generators. Quality of service managers, as discussed in Section 4.3.2 are distributed on these end systems as necessary to manage the quality of service delivered by user application processes. Load generators are tools, both hardware and software, used to induce some kind of a load in the environment to utilize a variety of resources, such as CPU cycles, memory, network bandwidth, disk throughput, video bandwidth, and so on.

#### 7.1.4.4 Testbed Applications

Users access the services of the testbed through external testbed applications. These applications invoke the testbed operations through the testbed's service interface. Examples of such applications, are shown in Figure 7.1, and explained below.

**Experiment Drivers:** Carrying out an experiment using the testbed involves selecting the desired configuration, executing the experiments, and analysing the collected results. Even the simplest analysis requires several experiments with different configurations to be repeated over extended periods of time. Repeating the experiments manually would be extremely tedious. An experiment driver facilitates and automates this process.

**Report Generators:** Report generation applications can be used to aggregate the data from experiments, perform high-level analysis, and present the results.

**Visualization Tools:** Experiments can be configured and controlled more intuitively by manipulating visual abstractions of the components involved. Data can be presented more concisely and understandably in graphical forms.

#### 7.1.4.5 Testbed Implementation Details

The quality of service management testbed was developed in a laboratory setting with a dozen hosts of various architectures spread over several kinds of networks. Since our quality of service management system used resource managers targeted at the Solaris 2.6 environment, we primarily used our Sun UltraSparc workstations for experiments, although the testbed itself is supported on multiple platforms.

#### 7.1.4.6 Testbed and Testbed Services

To meet the flexibility and portability requirements demanded by the heterogeneity of the environment, we implemented the testbed as a structured collection of Unix shell scripts. Since the testbed operates outside of the application processes whose quality of service we are assessing, most of its interactions are with the operating system or by recording already available application behaviour. The ease of access to system services and utilities, and the fact that the performance of the testbed is of little concern, made shell scripts an obvious choice.

The testbed architecture is preserved by structuring each service as a collection of scripts providing the required operations. The interface to the testbed is defined by script names and their arguments. Testbed applications that invoke these scripts can be written in a variety of languages (for example, shell scripts, C, Tcl, Perl, or Java).

#### 7.1.4.7 Distributed Environment Components

The hardware infrastructure used for experimentation in this work consists of a collection of networked workstations. Each workstation is connected via 10/100 megabit Ethernet to a dedicated Ethernet switch. The workstations are all part of the same logical sub-network, isolated from other departmental computers used for teaching and research.

In the experimentation discussed in this work, three workstations are used:

`farquarson`: A Sun Microsystems UltraSparc 1 workstation with a 140 Mhz processor, 64 megabytes of physical memory, and 300 megabytes of virtual memory. `farquarson` is also equipped with a SunVideo board for video capture. To support our prototype environment, the workstation runs Solaris 2.6 as its operating system.

`strawberry`: A Sun Microsystems UltraSparc 5 workstation with a 269 Mhz processor, 128 megabytes of physical memory, and 480 megabytes of virtual memory. `strawberry` is also equipped with a SunVideo board for video capture. To support our prototype environment, this workstation also runs Solaris 2.6 as its operating system.

`vanilla`: A Sun Microsystems UltraSparc 5 workstation with a 269 Mhz processor, 256 megabytes of physical memory, and 760 megabytes of virtual memory. `vanilla` is also equipped with a SunVideo board for video capture. To support our prototype environment, this workstation also runs Solaris 2.6 as its operating system.

In most cases, we tended to use `farquarson` as a server machine when necessary, as it is SBus and SCSI based, as opposed to the UltraSparc 5 workstations, which were PCI and IDE based. The SBus and SCSI combination tended to make `farquarson` faster in I/O operations and serving data, despite having less memory and a slower processor.

#### 7.1.4.8 Applications

We have incorporated a variety of monomedia and multimedia applications into the testbed. This allows for a wide range of experiments, as each application has different resource requirements and different quality of service needs.

Traditionally, multimedia applications have been the focus of quality of service experimentation; consequently, we have done the same. For the most part, experiments in this work have focussed on video conferencing and video-on-demand applications.

The video conferencing application (RTVC) is a sample package distributed with SunVideo capture boards and cameras [187]. This application consists of a transmitter, which captures video from a camera on its host and broadcasts it using UDP over the network; and a receiver, which retrieves broadcast video, decodes it, and displays it on its host. The video is transmitted in Motion-JPEG format; again requiring CPU cycles on the client to decode and display the video. While alternate formats were available, such as CellB (a proprietary Sun video format), Motion-JPEG produces high quality results. (MPEG video could not be transmitted by the software for technical reasons.) The transmitter can optionally display the captured video locally without broadcasting the stream to a receiver, effectively allowing it to operate as a single process application as well.

The video-on-demand application (VOD) is derived from the Berkeley MPEG video player [152], which has been split into a simple client-server application. Upon initialization, the video-on-demand client contacts a video-on-demand server at a fixed location in the network and requests the delivery of a particular MPEG video file. This video is streamed over TCP to the client until completion. Since the video is delivered encoded to conserve network bandwidth, the client can consume a sizable number of CPU cycles decoding the MPEG video to display it on the local host. The pre-split version of the application has been preserved to function as a single process stand-alone video player application as well.

These applications were selected as they both tend to be resource intensive, are flexible, and support both single process and distributed modes of operation. Consequently, the applications were instrumented to track their video frame rate using a frame rate sensor, as discussed in Section 6.1.2.2, and report quality of service violations through a quality of service coordinator, as discussed in Section 6.1.1.2. As such, the applications were manageable and ready for experimentation.

#### 7.1.4.9 Load Generators

To be able to apply loads to distributed environment components for experimentation, several load generators have been developed. By generating a load using these programs, we can provide a variety of operating conditions under which application processes and quality of service managers can execute in the testbed.

Using the load generator programs implemented for the testbed, we can induce loads on processors, memory, network links, and disk resources. These programs can be configured to allow for different loads to be generated in a variety of user-definable patterns. The load generators can be used individually or in concert with one another to allow for a wide range of loads to be generated in the distributed environment. Since we have yet to fully implement network or disk resource managers, the corresponding load generators were not used extensively in this work. The processor and memory load generators, however, were used frequently, and are discussed in more detail below.

Using our CPU load generator, we can produce constant loads ranging from very light to incredibly heavy, random loads that vary uniformly between arbitrary bounds, and random loads that vary according to a Poisson distribution, for more realistic scenarios [83]. CPU load is measured and enforced in terms of load average—the mean number of processes waiting in the run queue to get access to the CPU. Load is increased by adding CPU bound jobs and decreased by terminating them. Smaller changes to load are made by adjusting the computations of each process.

Our memory load generator can generate load on the memory system in a variety of ways. Small amounts of load can be created by repetitive allocation and freeing of memory, or by locking some portion of physical memory to prevent paging. The most dramatic loading technique is to allocate a large portion of memory (larger than physical memory), and then touching each page individually to induce a page fault. When load generator parameters are tuned to have this process repeated quickly and randomly (to defeat locality-based preloading of pages), a very high load can be induced on the memory system. This load can be made even heavier by having the locator spawn copies of itself, each touching a smaller region of memory. The increased page faulting from this technique can force a system to easily devote over 95% of its time to paging activities.

#### 7.1.4.10 Testbed Applications

For initial testing and experimentation, a basic experiment driver and a simple report generator were constructed. Tools have also been built to convert experiment results into spreadsheets and graphs for simple visualization. The experiment driver uses scripts to guide the execution of several experiments in sequence. To complete each phase of an experiment, the driver makes use of the appropriate service scripts provided by the testbed. Each experiment consists of eight phases:

1. Starting the experiment and gaining sole access to the testbed.
2. Configuring the testbed, environment components, application processes, quality of service solutions, and load generators, depending on the goals of the experiment.
3. Starting the environment components, application processes, solutions, and load generators used in the experiment.
4. Monitoring the started application processes and collecting the required information.
5. Terminating the environment components, application processes, solutions, and load generators and collecting the state information needed.
6. Analysing the collected data as necessary.
7. Finishing the experiment.
8. Generating reports summarizing the results of the experiment.

A more complex experiment driver was written for experimentation in [132]. This driver was written in Java, and provides a graphical interface for easy experiment configuration.

## 7.2 Preliminary Experimentation

Before turning to more rigorous experiments involving the quality of service management system and the algorithms under study, we first briefly discuss prior experimentation used to validate our implementations and to determine reasonable values for experiment parameters. These initial experiments laid the ground-work for the experimentation discussed in detail in the remainder of the work.

### 7.2.1 Testbed Validation

The first experiments conducted were to verify that the quality of service management testbed itself was functioning correctly. This included thoroughly testing the instrumented applications, load generators, and experiment drivers and analysis tools to ensure that they were working. These experiments proved to be quite successful and demonstrated that the testbed was indeed suitable for supporting the experimentation required for this work. More details of these experiments can be found in [93].

### 7.2.2 Quality of Service Management System Validation

Having found that the testbed was working correctly, the next step was to use the testbed to ensure that the quality of service management system we developed was functional as well. This included thorough testing of the policy engine, all communication and interactions with managed applications, all implemented algorithms, and all resource managers. Once again, these experiments proved to be successful, and the quality of service management system was deemed ready for more rigorous experimentation. Further details of these experiments can be found in [89, 91, 94, 118, 134].

### 7.2.3 Selecting Values for Experiments

Before commencing more rigorous experiments involving the various resource management algorithms from Chapter 5, we first needed to use the quality of service management testbed to determine reasonable values for the wide variety of parameters that define and regulate our experiments. To help determine suitable values for parameters of a subjective nature, consultation was made with at least 5 additional individuals (typically those also working on quality of service issues or related problems within our department) as time and availability permitted.

These experiment parameters include the following:

- Realistic load levels to use when applying load to resources with our load generators. To keep experimentation simple we decided to select low and high load levels for both our CPU and memory resource.

For the CPU resource, we chose to apply a load of 0 for a low load, meaning that no load would be induced by our load generator, and a load of 8 for a high load. Experimentation showed that the systems and applications used in experiments grew progressively harder to control under loads higher than 8, so the load was capped at this value. It is important to note that a load value of 8 is a very high load; much higher than normal operating conditions for compute servers in our department. However, we decided to apply the higher load to gauge the performance of the management system in harsh conditions.

For the memory resource, we again chose to impose no additional memory activity for a low load. For a high load, we decided to induce a high paging load through continuous random page touching of a large amount of virtual memory. This amount was chosen so that there would be no virtual memory free after executing our quality of service management system and the largest video application to be used in experimentation. Sufficient memory load processes were created to touch random pages in this memory region to dedicate between 90% to 95% of system time to handling paging activities. This too was done to ensure control over experiments was maintained. On occasion, it was noted that the high memory load could make applications and the system itself unresponsive as the load reached its peak value; after a brief period of unresponsiveness, we could gain control of the system and applications once again. A cause for this phenomenon was not discovered.

- Realistic workloads for instrumented applications under study to execute. Two factors of interest were determined here: the length of video workloads, and the resolution of the video workloads. The content of the video was fixed, as we needed a constant stream of video, 24 hours a day, 7 days a week, and our department's operations counter provide the only suitable source meeting these requirements easily.

  Video length was selected to ensure that each experiment would last for at least 10 minutes. This time was selected as early experiments indicated that it was long enough for all required observations, but also not too long, so as to permit the number of experiments required in this work.

  Video resolution was selected on a per experiment basis. Essentially, the more video players required by experimentation on a single host, the lower resolution had to be dropped, so that the host could accommodate the additional workload. If resolution was kept at a single high level, it would be too difficult to find the resources to meet quality of service requirements, and every experiment would produce negative results. At the same time, if the resolution was kept at a single low value, it would be too easy to meet requirements without significant work, and it would be difficult to distinguish between the various alternative algorithms in such cases. Experiments indicated that, for most experiments, the default resolution provided by our SunVideo hardware produced the best balance of results; neither too high or too low a workload.

- Reasonable values for settings guiding the behaviour of the various resource management algorithms. This includes data sampling intervals and adjustment scaling factors.

  Frame rate data from all video applications was collected for each frame of video processed by each application. This data collected was sampled every 500 milliseconds and used to compute the frame rate being delivered at that time. This sampling interval was selected as it tended to provide balanced results—results that accurately reflected variations in the frame rate delivered on average, without producing too much variation caused by variable frame processing times. Higher sampling frequencies resulted in frame rates with too much variation, dominated by the fluctuations caused by individual frames. Lower sampling frequencies smoothed results too much, providing an unrealistic view of quality delivery, and reducing responsiveness to changes in the frame rate being provided.

  When computing the amounts of resource adjustments to be made by each algorithm, a general scaling factor was used to help determine the magnitude of adjustment made. Experiments indicated that a factor of 10 produced reasonably balanced results. Larger scaling factors resulted in faster changes to resource allocations, but also led to a significant number of over-compensations, when allocations were too larger. Smaller scaling factors would avoid over-compensation problems, but would take too long to deliver acceptable quality.

- Reasonable expectations for convergence times and stability for use in our later analyses, as discussed in Section 7.1.3. Attempting to find values for these parameters was quite difficult.

  Requiring an excessive number of samples in sequence for convergence would make it quite difficult for any algorithm to converge in most situations; however, requiring too few would make it trivially easy. Likewise, providing too much time for quality to settle after a disturbance would make it trivial for any algorithm to be stable; too little time would make it quite difficult. It was found through subjective testing that most people deemed convergence to occur after 5 seconds, on average, which translated into 10 quality samples. To be consistent, the same value was selected for stability requirements as well.

- Reasonable replications for experiments. Experiments must be repeated to ensure that results obtained are not out of the ordinary or caused by exceptional circumstances. However, considering the number of different experiments needed for this work to compare resource management algorithms and various enforcement policies, requiring too many replications would make completing experimentation impossible in the time allotted. Consequently, each experiment was replicated 5 times. This provides some assurance to the quality of results, while not imposing too much of a cost to experimentation either.

With such values determined, we were finally ready for further quality of service management experimentation. These experiments are discussed in detail in the sections to follow.

## 7.3 Single Application Experiments

With previous experiments validating our test environment and quality of service management system, and providing reasonable values for experiment parameters, we can now turn to more rigourous experiments. In this first set of experiments, we examine the performance of our three selected algorithms in managing a single application on a single host under various operating conditions.

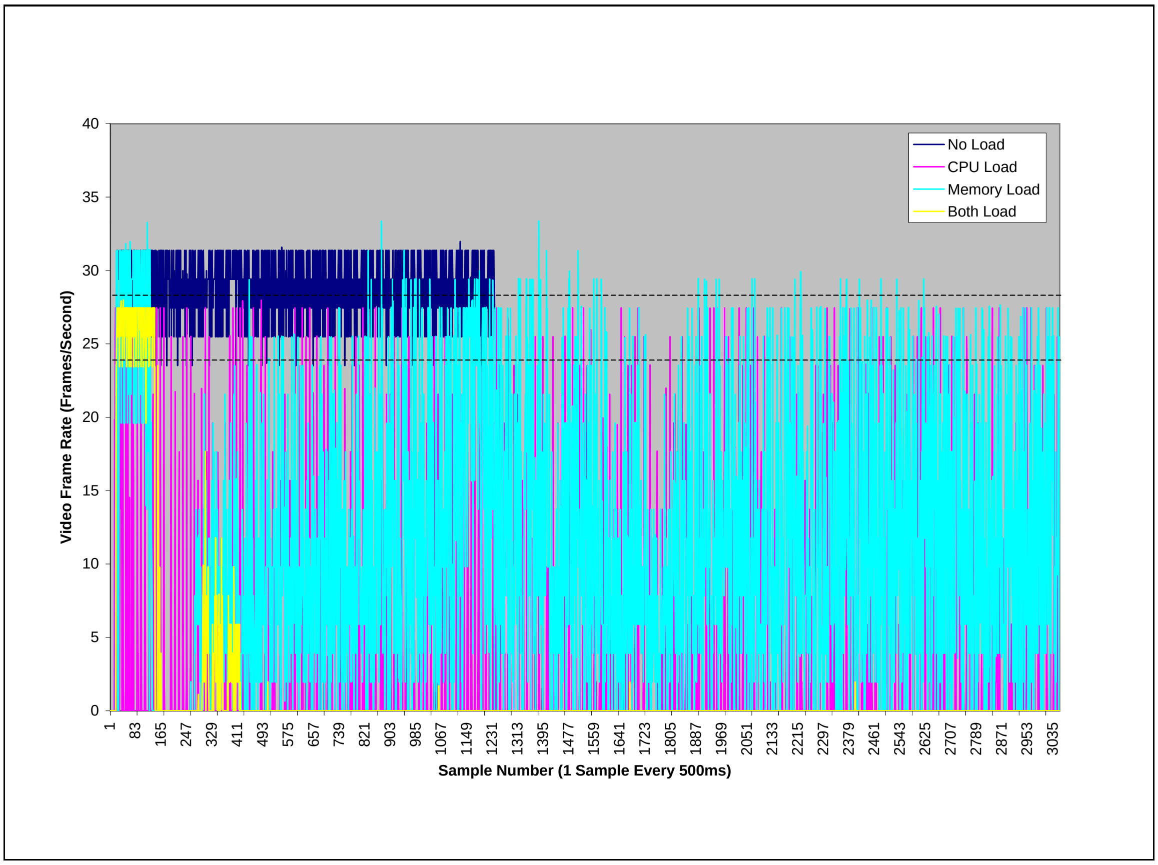
The application selected for these experiments was the stand-alone version of the RTVC video conference application. The target workstation for all experiments was `farquarson`. In all cases, the application was instructed to capture and display 18000 frames of a 320×240 8-bit video stream from a camera at the department's operations counter. Also in all cases, the application was given a quality of service expectation policy dictating that it must maintain a frame rate of 26 frames per second, plus or minus 2 frames (giving an acceptable range of 24 to 28 frames per second).

When management is required, a quality of service host manager is executed on `farquarson` as well. A domain manager is also executed for completeness, but is not actually involved in the management process at this point.

### 7.3.1 Base Case Experiments

The first set of experiments executed was a set of experiments to determine the behaviour of the RTVC application under different conditions without the benefit of our quality of service management. These experiments could then be used both as a base line for comparison, to determine the improvements brought by our management, and for a simple regression analysis to determine the initial resource split for the multidimensional search algorithm as discussed in Section 5.3.

**Figure 7.2** Single Application Base Case Experiments




For these experiments, we chose two levels of load to be applied to each of two resources, CPU cycles and memory, using our load generators. For the CPU, we chose loads of 0 and 8, as mentioned earlier. For memory, we chose no load and a very high paging load. Combining these loads, we have four possible experiment configurations: no load, CPU load only, memory load only, and both CPU and memory loads. By having unloaded and highly loaded conditions for both resources in this manner, we can easily examine results in a variety of repeatable and sustainable operating conditions. Each of these experiments was replicated.

Figure 7.2 presents a representative run from each load configuration, with frame rate sampled every 500 milliseconds. Key results are summarized in Table 7.1, including the mean frame rate delivered (in frames per second), and the percentage of quality samples that violated quality of service requirements (the number of frame rate measurements violating

**Table 7.1** Summary of Single Application Base Case Experiments

| Load Applied | Mean Frame Rate (FPS) | Policy Violations (%) |
|---|---|---|
| None | 28.53 | 82.25% |
| CPU only | 3.40 | 96.02% |
| Memory only | 7.79 | 92.24% |
| Both CPU and memory | 0.21 | 99.40% |

requirements divided by the total number of measurements taken).

The RTVC application could only converge and meet its quality requirements under the no load configuration. Even then, however, the quality delivered was not stable and a considerable percentage of samples violated quality of service requirements. (In all fairness, however, most violations were due to the application exceeding requirements, rather than failing to meet them.) In cases where load was applied, the RTVC application was unable to meet requirements. The application fared better under memory loading compared to CPU loading; with both loads combined delivering the worst overall performance.

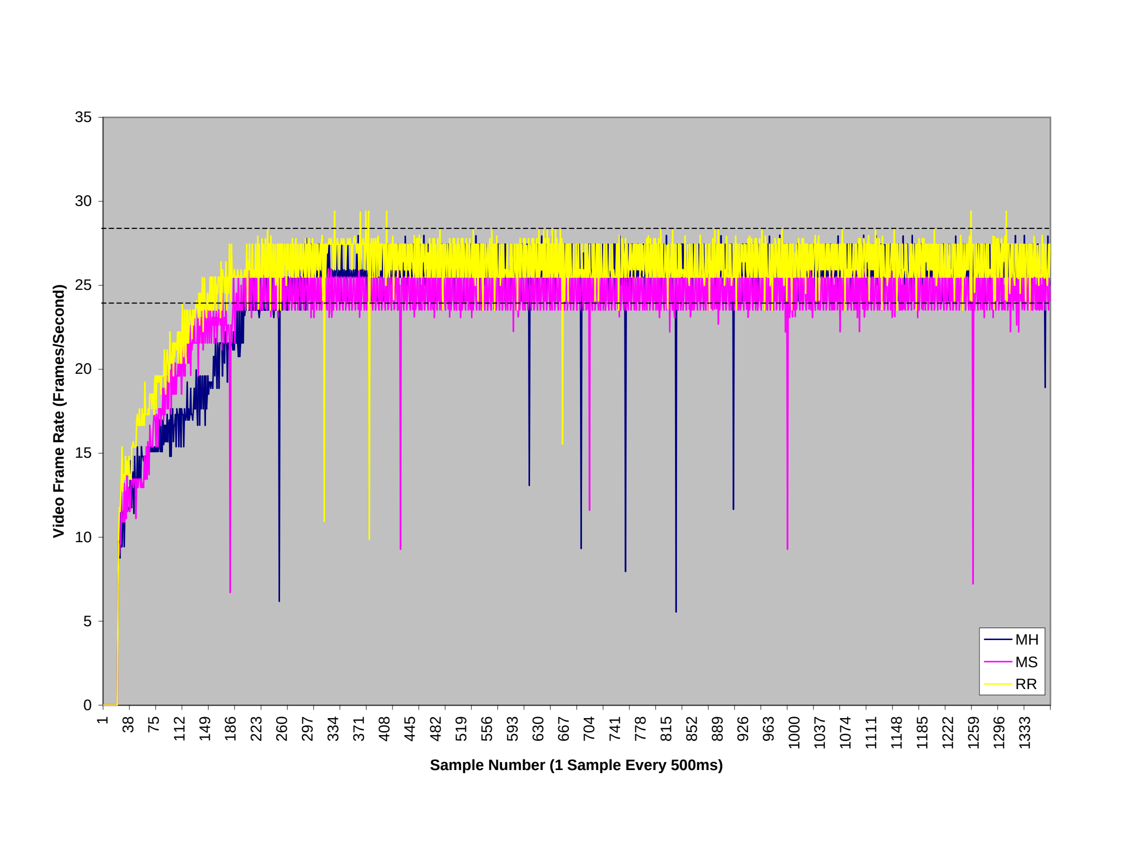
It must also be noted that memory load was only applied after the RTVC application had started; otherwise, the system would be so loaded that the application would have difficulty starting. Consequently, in the memory load alone and both resources load configurations, the applications had initial opportunity to perform in an unloaded situation until the load was applied in full force. (This is explains the brief period of good performance at the beginning of the sample runs in Figure 7.2.) This means that actual performance numbers for those configurations are slightly better than they should be, if their entire execution was under a loaded condition.

### 7.3.2 Single Resource Load Comparison Experiments

Having established base lines for quality of service delivery for the stand-alone RTVC application in the previous set of experiments, we now examine the performance of the application when our quality of service management system is in place. For these experiments, we compare our selected algorithms (metric-heuristic (MH), multidimensional interpolation search (MS), and dynamically weighted round robin (RR)) when load is applied to a single resource; experiments examining the effects of loading multiple resources are presented in the next section.

For our resource to load, we have selected the CPU. For consistency with previous experiments, we have chosen to apply a load of 8. Each experiment was again replicated.

**Figure 7.3** Single Application: CPU Load, CPU Management Alone Experiments




#### 7.3.2.1 CPU Management Alone

Figure 7.3 presents a representative run of each algorithm under this load configuration, with each algorithm managing only the CPU resource. Key results are summarized in Table 7.2, including the convergence time, mean frame rate (in frames per second), percentage of quality samples violating quality of service requirements, standard deviation of quality samples, and CPU overhead. When convergence occurred, the mean frame rate, policy violations, and standard deviation are listed for all samples after convergence; when convergence did not occur, values reported reflect the entire run of the application. Once again, video frame rates were sampled every 500 milliseconds.

In all cases, the metric-heuristic and round robin algorithms converged and were stable. The multidimensional search algorithm, on the other hand always came close to converging but could never quite make it. The round robin approach converged slightly faster on average

**Table 7.2** Summary of Single Application, CPU Load, CPU Management Alone Experiments

| Algorithm | Convergence Time (%) | Mean Frame Rate (FPS) | Policy Violations (%) | Standard Deviation (FPS) | CPU Overhead (%) |
|---|---|---|---|---|---|
| Metric-Heuristic | 17.69% | 26.26 | 1.12% | 1.09 | 0.31% |
| Multidimensional Search | N/A | 22.83 | 37.30% | 2.77 | 0.61% |
| Round Robin | 13.16% | 26.35 | 4.46% | 0.94 | 0.14% |

than the metric-heuristic approach, and had a lower standard deviation after convergence; however, the metric-heuristic approach encountered fewer policy violations after convergence than the round robin approach. This implies that while the metric-heuristic approach on average had fewer policy violations than the round robin approach, there was a slightly higher variation in its quality within policy boundaries. In terms of overhead, the round robin approach fared better than the metric-heuristic approach, which in turn had lower overhead than the multidimensional search approach. All in all, each algorithm appeared to be relatively efficient, with little overhead.

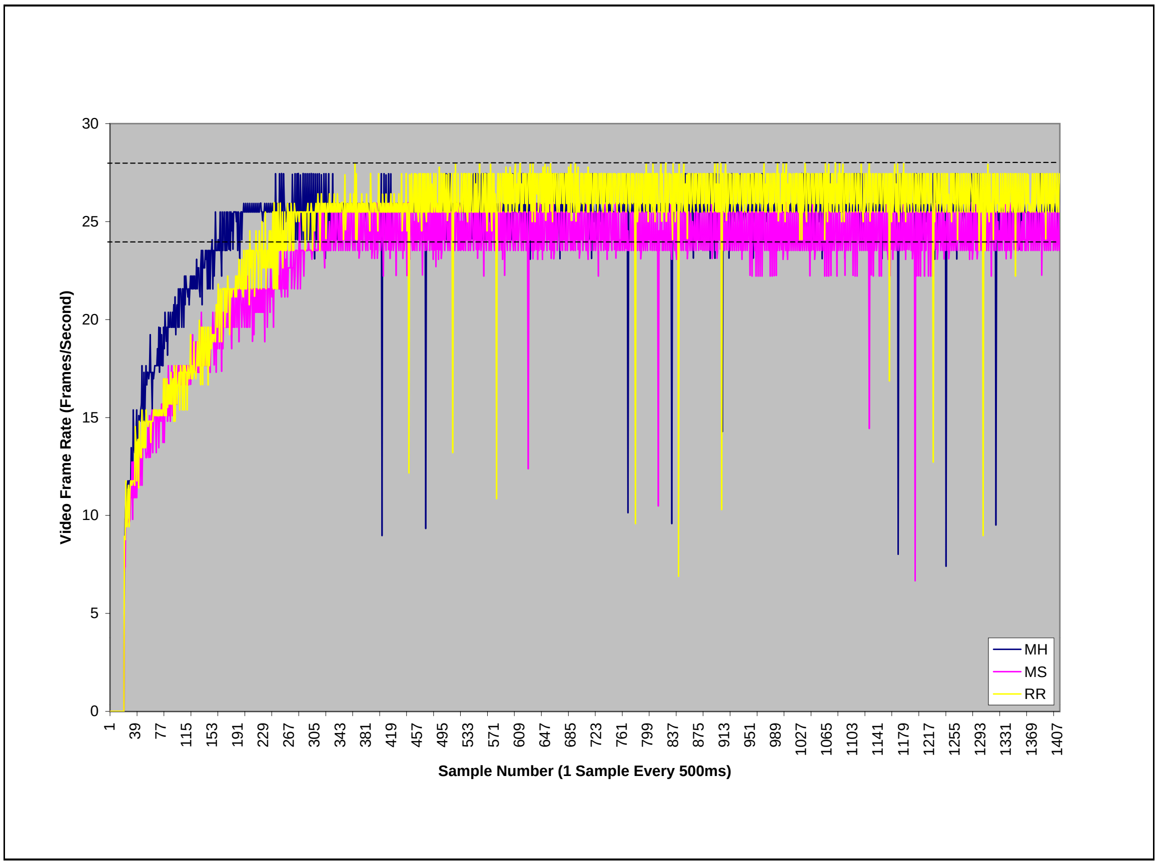
#### 7.3.2.2 Both CPU and Memory Management

In the previous set of experiments, we instructed all algorithms to manage only the CPU resource that we were loading. In this set of experiments, we repeat the prior set with the same loading conditions, except that now our resource management algorithms are managing both CPU and memory allocations.

Figure 7.4 presents a representative run of each algorithm under this scenario. Key results are summarized in Table 7.3, including the convergence time, mean frame rate (in frames per second), percentage of quality samples violating quality of service requirements, standard deviation of quality samples, and CPU overhead. When convergence occurred, the mean frame rate, policy violations, and standard deviation are listed for all samples after convergence; when convergence did not occur, values reported reflect the entire run of the application. Once again, frame rates were sampled every 500 milliseconds.

In all cases, the metric-heuristic and round robin algorithms again converged and were stable. The multidimensional search algorithm, on the other hand, still came close to converging but could never quite make it, and demonstrated slightly worse performance than when CPU alone was being managed. Unlike the prior experiments, the metric-heuristic approach converged more quickly than the round robin approach. Stability results for these two algorithms remained similar to previous results. In terms of overhead, the

**Figure 7.4** Single Application: CPU Load, Both CPU and Memory Management Experiments




metric-heuristic approach fared best this time, followed very closely by the round robin approach, with the multidimensional search approach trailing farther back.

Observed changes in convergence and performance from previous experiments are likely due to the nature of the algorithms. The round robin approach tends to be quicker in choosing the resource to adjust than the metric-heuristic approach. When only a single resource is managed, and that resource is being loaded, the round robin approach will always choose the correct resource to adjust. The metric-heuristic approach will as well, but will take slightly longer. When multiple resources are available for management, however, the round robin approach will make mistakes in choosing the resource to adjust, particularly until its weights are dynamically adjusted properly (even then, mistakes are still possible). Consequently, in these cases, the metric-heuristic approach would converge faster. If the multidimensional search approach had been able to converge properly, it would appear to

**Table 7.3** Summary of Single Application, CPU Load, Both CPU and Memory Management Experiments

| Algorithm | Convergence Time (%) | Mean Frame Rate (FPS) | Policy Violations (%) | Standard Deviation (FPS) | CPU Overhead (%) |
|---|---|---|---|---|---|
| Metric-Heuristic | 13.73% | 26.51 | 1.23% | 1.18 | 0.26% |
| Multidimensional Search | N/A | 22.42 | 48.41% | 2.88 | 0.64% |
| Round Robin | 19.45% | 26.23 | 2.20% | 1.11 | 0.27% |

have similar performance to the round robin approach, as it too would make individual adjustments quicker than the metric-heuristic approach, but could suffer from incorrect resource adjustment selections.

The reduction of overhead for the metric-heuristic approach in this case can be attributed to its decrease in average convergence time. Since it converges faster, it has less work to do in the long run, and consumes fewer resources. The decrease in convergence time could indicate that memory resources were insufficient in the previous experiment, even though the resource was not loaded, and that managing both resources improved the situation.

### 7.3.3 Multiple Resource Load Comparison Experiments

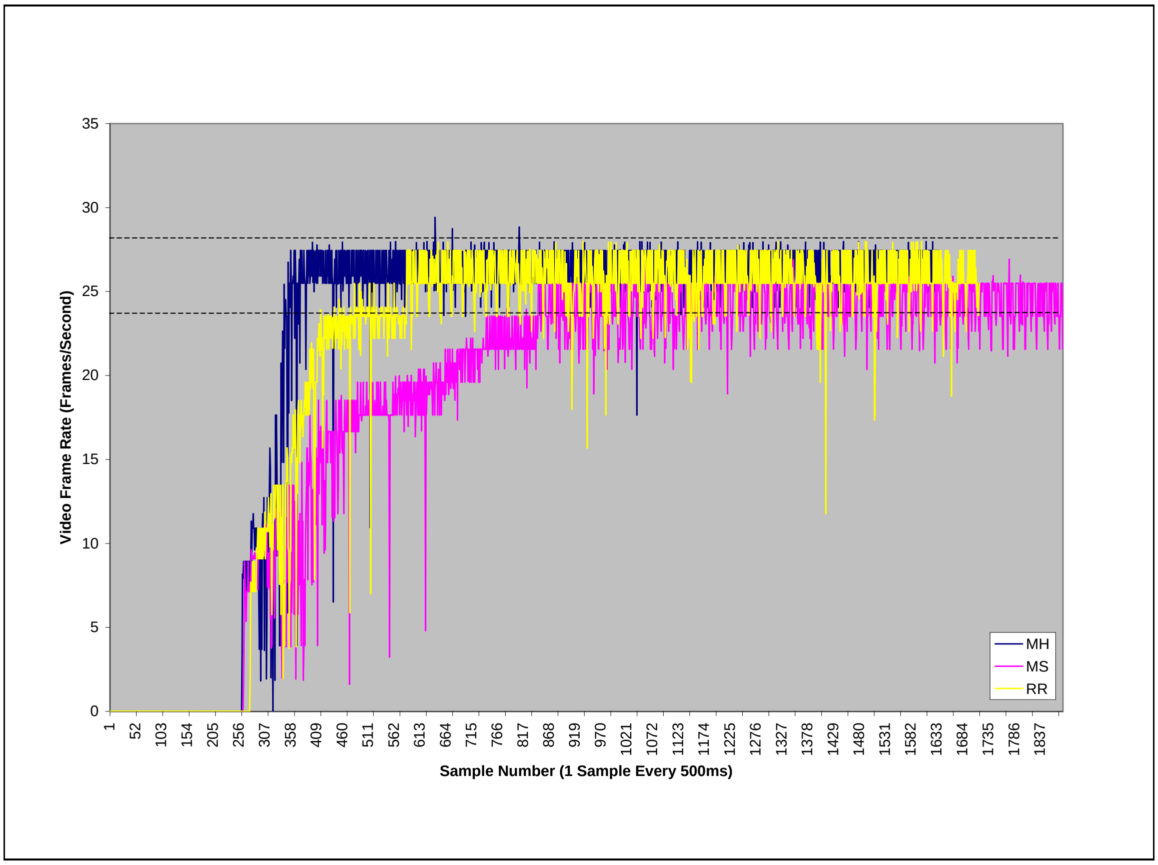
The previous experiments examined the impact of loading a single resource on the quality delivered by the stand-alone RTVC application when managed by our quality of service management system. In this set of experiments, we examine the effects of loading multiple resources on the same application when managed by our selected resource management algorithms.

For consistency with previous experiments we have chosen to load the CPU resource with a load of 8, and memory with the same heaving paging workload as in the base case experiments. Each experiment was again replicated.

Figure 7.5 presents a representative run of each algorithm under this scenario. Key results are summarized in Table 7.4, including the convergence time, mean frame rate (in frames per second), percentage of quality samples violating quality of service requirements, standard deviation of quality samples, and CPU overhead. For all cases, the mean frame rate, policy violations, and standard deviation are listed only for samples after convergence. Once again, frame rates were sampled every 500 milliseconds.

In all cases, the metric-heuristic and round robin algorithms converged and were stable. For the first time with the stand-alone RTVC application, the multidimensional search algorithm also was able to converge and be stable. Comparing the performance of the

**Figure 7.5** Single Application: Both Load, Both CPU and Memory Management Experiments




algorithms, we found that on average the metric-heuristic approach converged first, followed by the round robin approach, with the multidimensional search approach trailing farther behind. Convergence times given in Table 7.4 are not bad considering that the extreme load conditions prevented the application from detecting and reporting its first quality of service policy violation until roughly 15% through its execution time. Quality of service increased substantially from that point on, as seen in the runs shown in Figure 7.5. In terms of both stability and overhead, the metric-heuristic approach also fared best on average, followed by the round robin and multidimensional search approaches. While resource usage increased to handle management of both resources, overhead is still reasonably low.

The multidimensional search is believed to be successful in this case when it was not in earlier experiments because the cycles through the algorithm adjusting memory resources are now actually effective in advancing the quality of service provided by the RTVC application.

**Table 7.4** Summary of Single Application, Both Load, Both CPU and Memory Management Experiments

| Algorithm | Convergence Time (%) | Mean Frame Rate (FPS) | Policy Violations (%) | Standard Deviation (FPS) | CPU Overhead (%) |
|---|---|---|---|---|---|
| Metric-Heuristic | 22.95% | 26.62 | 0.87% | 0.91 | 0.61% |
| Multidimensional Search | 73.94% | 24.45 | 39.92% | 0.92 | 1.73% |
| Round Robin | 35.43% | 25.81 | 13.17% | 1.13 | 1.02% |

Previously, when both resources were managed and only CPU was loaded, the algorithm could meet the CPU portion of its target, but not the memory portion, as its memory allocations had insufficient positive impact on quality of service. Unfortunately, more CPU resources were needed than its target indicated, due to the high load on the CPU. Because the algorithm was not adaptive (as discussed in Section 5.3), it could not correct its split quality targets accordingly. This problem of split quality targets would not arise, however, when the CPU was the only resource being managed by the algorithm; yet, it was still unable to meet quality requirements in this case. This is likely because the multidimensional search algorithm induced a small memory load of its own, since it requires a larger amount of state information—with memory management disabled, the necessary adjustments could not be made to allow the RTVC application to meet quality targets. This produces an interesting irony—for this application, the multidimensional search required some form of memory management for the application to meet its target; actually managing the resource, however, put too much emphasis on that resource, so quality could not be met then either. Only when memory load was applied did memory resources become important enough to have quality actually meet requirements. These issues hint toward potential complexity with actually deploying and using the multidimensional search algorithm; this will be discussed further in Section 7.8.

## 7.4 Multiple Application Experiments

The previous set of experiments dealt with examining the performance of a single application, the stand-alone RTVC application. In these experiments, we investigate the performance of our three selected algorithms in managing multiple applications on a single host under various operating conditions.

When management is required, a quality of service host manager is executed on the host executing the applications. A domain manager is also executed for completeness, but is not actually involved in the management process at this point.

### 7.4.1 Stand-Alone RTVC with Stand-Alone MPEG Player Experiments

Because the RTVC application requires exclusive access to the host's SunVideo capture board, we cannot run multiple copies of the program on a single host. Consequently, we chose to use the stand-alone version of the MPEG player application for our multiple application experiments.

The target workstation was again `farquarson`. In all cases, the RTVC application was instructed to capture and display 18000 frames of a 320×240 8-bit video stream from a camera at the department's operations counter, connected to the SunVideo board. Also in all cases, this application was given a quality of service expectation policy dictating that it must maintain a frame rate of 26 frames per second, plus or minus 2 frames (giving an acceptable range of 24 to 28 frames per second). The MPEG player application was instructed to display a 160×120 8-bit video from a file. In all cases, this application was given an expectation policy dictating that it must maintain a frame rate of 24 frames per second, plus or minus 2 frames (giving an acceptable range of 22 to 26 frames per second).

#### 7.4.1.1 Base Case Experiments

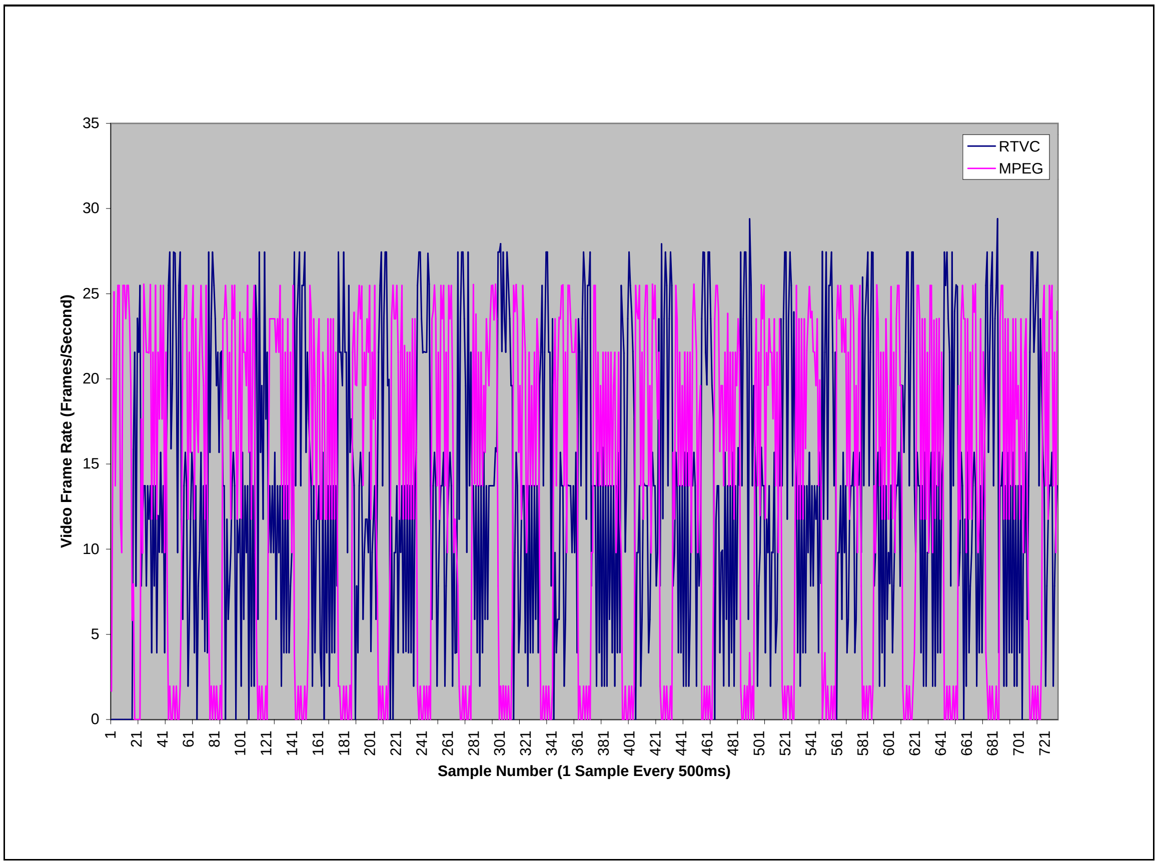
The first set of experiments executed was again a set of experiments to determine the behaviour of the RTVC and MPEG player applications under different conditions without the benefit of our quality of service management system. These experiments could then be used both as a base line for comparison, to determine the improvements brought by our management, and for a simple regression analysis to determine the initial resource split for the multidimensional search algorithm as discussed in Section 5.3.

For these experiments, we again chose two levels of load to be applied to each of two resources, CPU cycles and memory, using our load generators. For the CPU, we chose loads of 0 (no load induced by our generator) and 8. For memory, we chose no load and the same very high paging load as before. By combining these loads, we have four possible experiment configurations: no load, CPU load only, memory load only, and both CPU and memory loads. Each of these experiments was replicated.

Figure 7.6, Figure 7.7, Figure 7.8, and Figure 7.9 present representative runs from each application in the no load, CPU load, memory load, and both load configurations. Key results are summarized in Table 7.5, including the mean frame rate delivered (in frames per second), and the percentage of quality samples that violated quality of service requirements. Once again, frame rates were sampled every 500 milliseconds.

Even without any load applied to the system, neither application was able to meet quality requirements, and performance was quite poor. Since quality could not be delivered in such

**Figure 7.6** Multiple Application Base Case Experiments with No Resources Loaded

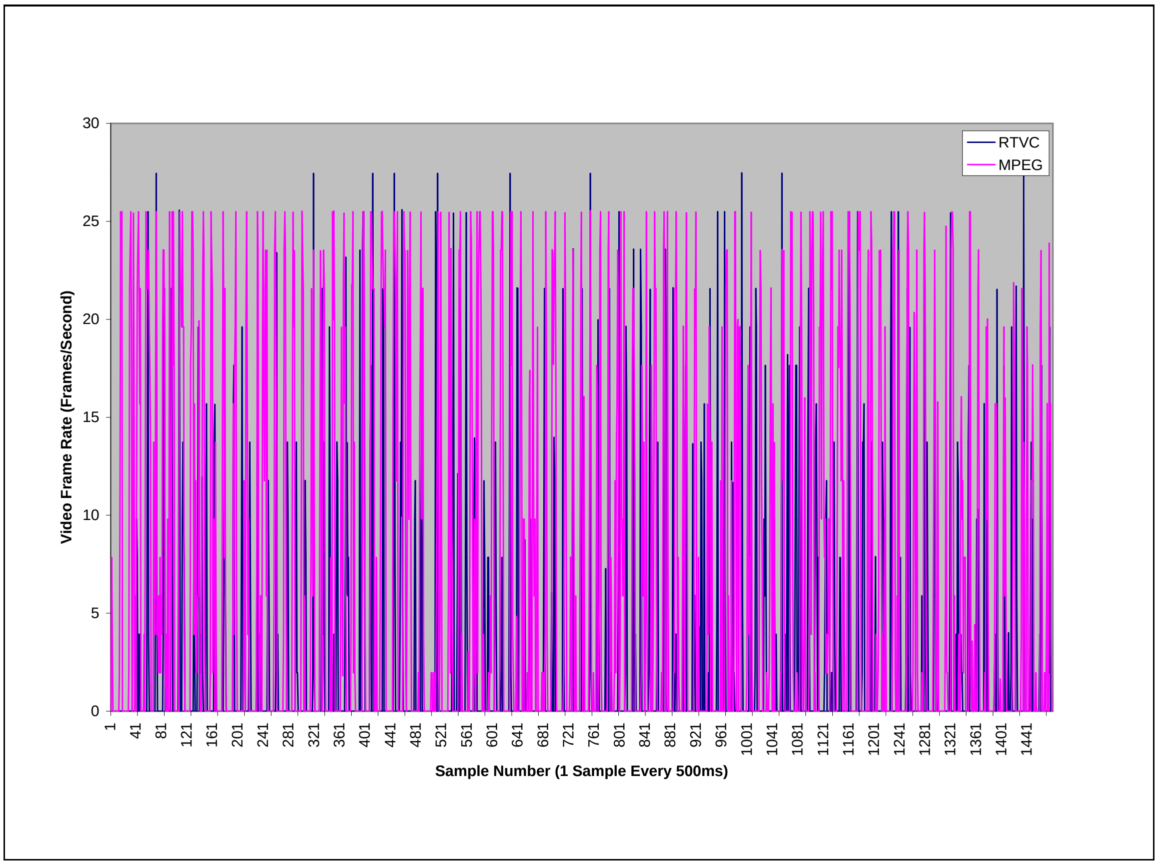



a case, one could only expect worse results in loaded situations. This, in fact, was exactly what happened—as additional load was applied, performance degraded even further. It is interesting to note, however, from both Figures 7.7, 7.8, and 7.9, and Table 7.5, that we can see that the two applications fared slightly differently in response to the different loading situations. For example, in cases where memory load was applied, the RTVC application made virtually no progress in its execution, while the MPEG application was less severely affected.

#### 7.4.1.2 No Resource Load Comparison Experiments

The base set of experiments discussed in the previous section demonstrated that the RTVC and MPEG player applications could not converge when executed at the same time, even when no other load was applied. There are two possible explanations for this: the tradi-

**Figure 7.7** Multiple Application Base Case Experiments with CPU Loaded

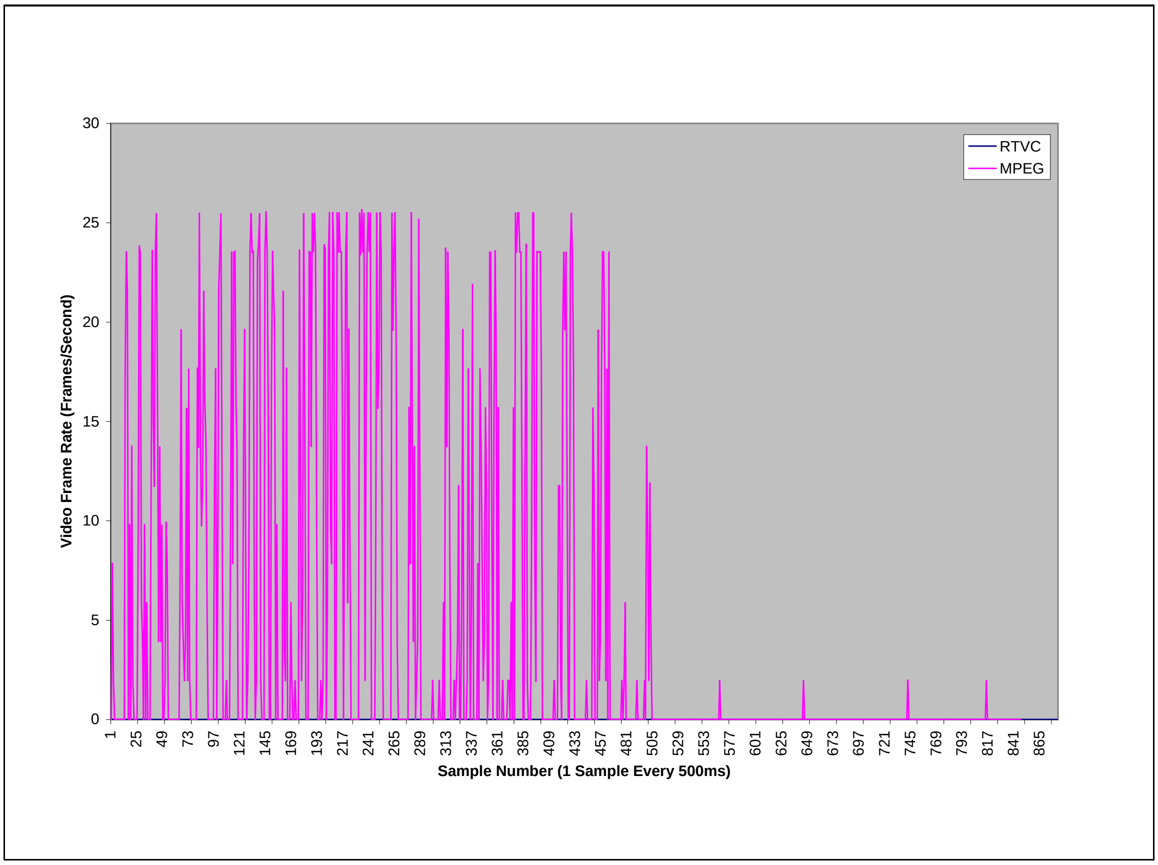



tional approaches to resource management on the system are unable to juggle and allocate resources well enough, or the sum of resources demand simply exceeds supply.

Measurements of CPU consumption of both applications indicate it is the latter case. Further support of this conclusion comes from examining Figure 7.6, in which one can easily see where the peaks of one application's performance coincide with the valleys of the other's performance as the system alternated between executing the two applications. Meeting the quality of one application consistently degraded the performance of the other to an extreme degree; this is an indication of insufficient resources.

If this is in fact the case, our resource management system will not likely be able to resolve the problem and bring quality in line with requirements. Since we are using a uniform service discipline in these experiments, it is likely both applications will suffer. This on its own, however, presents an interesting learning experience in which we can see how well the

**Figure 7.8** Multiple Application Base Case Experiments with Memory Loaded




various algorithms perform in such situations.

Figure 7.10 presents a representative run from each application managed by each resource management algorithm when no load has been applied, with frame rate sampled every 500 milliseconds. Key results are summarized in Table 7.6, including the mean frame rate (in frames per second), percentage of quality samples violating quality of service requirements, standard deviation of quality samples, and CPU overhead. As suspected, the combined resource demand of the two applications together exceeded supply; consequently, none of the algorithms were able to converge and provide quality meeting quality of service requirements.

On average, the metric-heuristic approach did the poorest of the three algorithms, with both lower quality results and higher resource overhead. The multidimensional search fared better on average, in both respects. On average, the round robin approach performed best

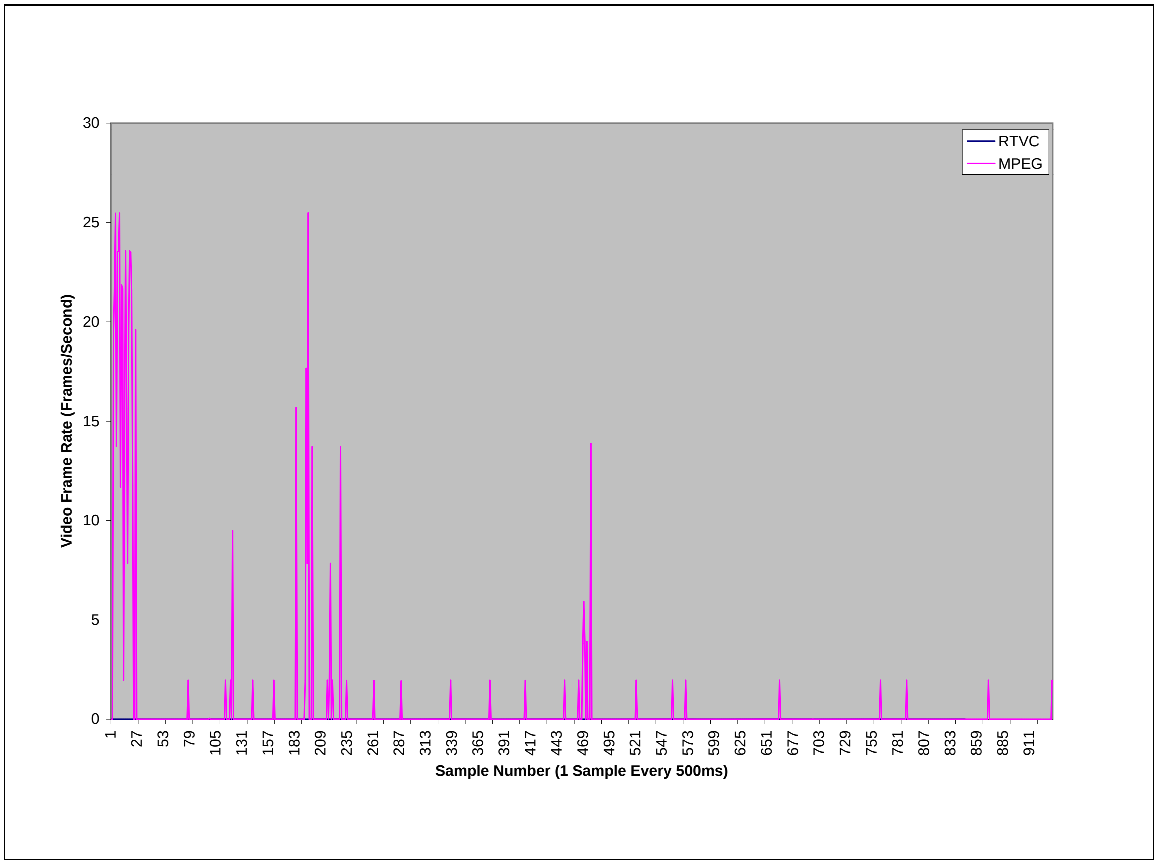
**Table 7.5** Summary of Multiple Application Base Case Experiments

| Load Applied | Application | Mean Frame Rate (FPS) | Policy Violations (%) |
|---|---|---|---|
| None | RTVC | 13.27 | 86.58% |
| None | MPEG | 12.76 | 73.15% |
| CPU only | RTVC | 2.65 | 98.24% |
| CPU only | MPEG | 6.27 | 88.38% |
| Memory only | RTVC | 0.03 | 100.00% |
| Memory only | MPEG | 4.30 | 89.73% |
| Both CPU and memory | RTVC | 0.01 | 100.00% |
| Both CPU and memory | MPEG | 0.61 | 99.14% |

**Table 7.6** Summary of Multiple Application Experiments with No Resources Loaded

| Algorithm | Application | Mean Frame Rate (FPS) | Policy Violations (%) | Standard Deviation (FPS) | CPU Overhead (%) |
|---|---|---|---|---|---|
| Metric-Heuristic | RTVC | 12.89 | 100.00% | 2.26 | 3.85% |
| Metric-Heuristic | MPEG | 17.18 | 98.49% | 2.79 | 3.85% |
| Multidimensional Search | RTVC | 15.83 | 100.00% | 2.21 | 2.22% |
| Multidimensional Search | MPEG | 18.91 | 80.04% | 2.65 | 2.22% |
| Round Robin | RTVC | 13.91 | 100.00% | 2.19 | 2.12% |
| Round Robin | MPEG | 20.13 | 67.89% | 2.61 | 2.12% |

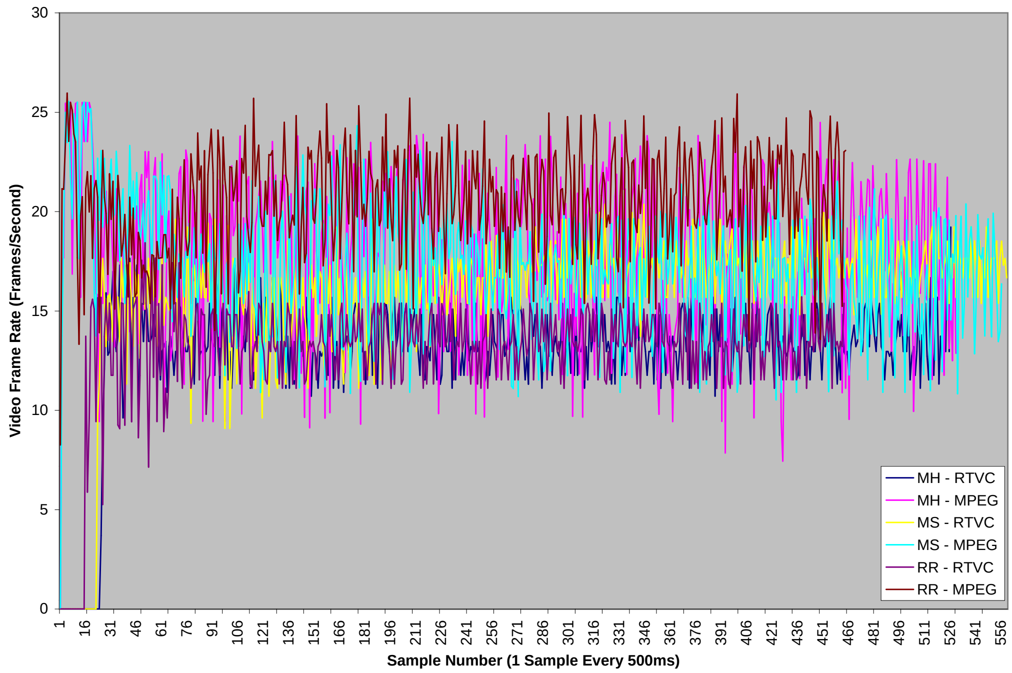
**Figure 7.9** Multiple Application Base Case Experiments with Both Resources Loaded




in most respects, delivering better quality at a lower overhead.

It was interesting to note that in this case, there was no substantial improvement in quality delivered than when no management was in place. In fact, in some aspects, quality of service was worse with management present. This is likely because the overhead from resource management when no load was present consumed additional scarce resources, leaving even less for the applications, resulting in reduced performance. Overhead was likely higher in this case, compared to previous experiments, because of the higher percentage of violations being reported by applications which simply could not meet quality of service requirements.

**Figure 7.10** Multiple Application Experiments with No Resources Loaded

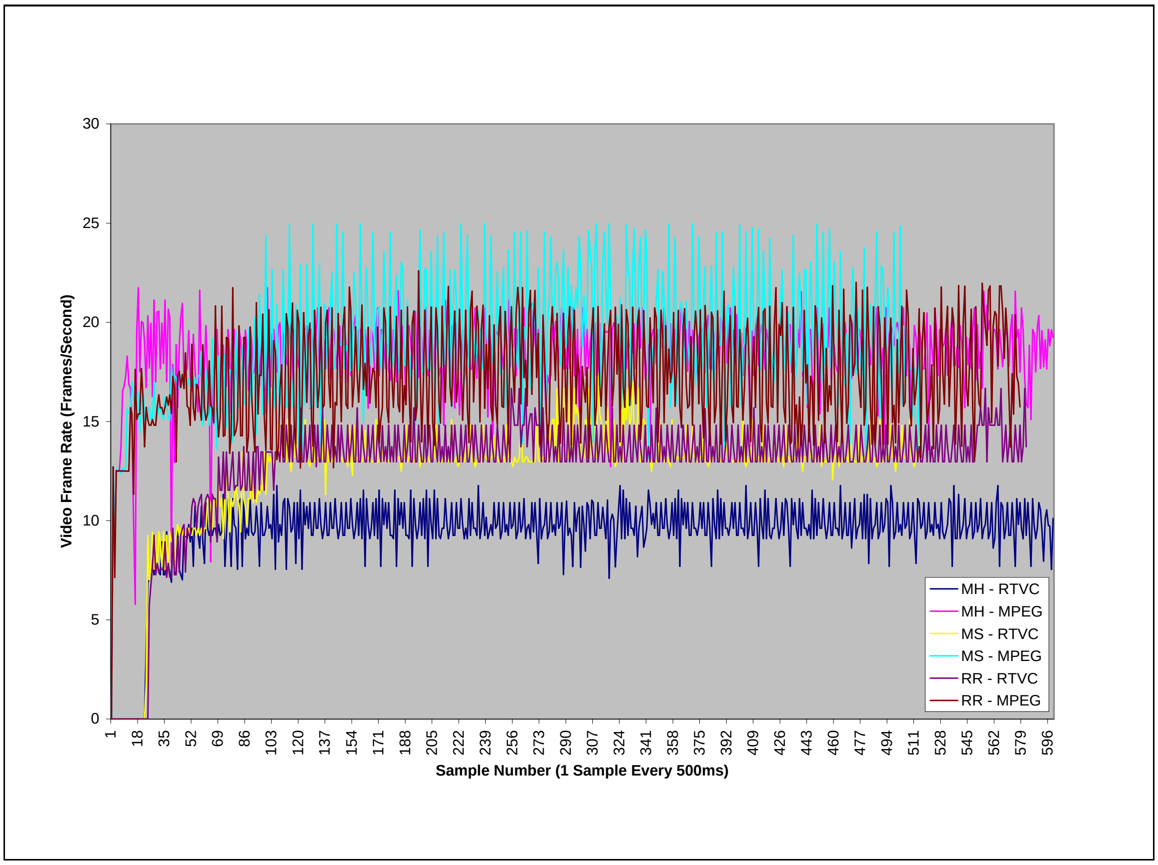



#### 7.4.1.3 Single Resource Load Comparison Experiments

The previous set of experiments demonstrated that management does not allow the MPEG player and RTVC applications to converge together when no load has been applied to the system. Consequently, management is not likely to succeed when the system has been loaded, but it will be interesting to see what happens regardless. To investigate this, we have again chosen to load the CPU resource with our load generator set to induce a CPU load of 8. Each experiment was replicated.

Figure 7.11 presents a representative run from each application managed by each resource management algorithm when load has been applied to the CPU, with frame rate sampled every 500 milliseconds. Key results are summarized in Table 7.7, including the mean frame rate (in frames per second), percentage of quality samples violating quality of service requirements, standard deviation of quality samples, and CPU overhead. As had been

**Figure 7.11** Multiple Application Experiments with CPU Loaded




thought, neither of the applications could meet quality requirements when managed by any of the algorithms.

On average, the metric-heuristic algorithm again fared worse than the other two approaches once again. The round robin approach fared better on average than the metric-heuristic algorithm. This time, the multidimensional search approach fared better than the other two algorithms on average in most respects; with the round robin approach posting better results in a few categories.

While none of the resource management algorithms offered significant improvements in terms of the percentage of time quality was met, all of the resource management algorithms decreased the gap between quality requirements and quality delivered, compared to using no management at all. This is an improvement over the no load experiments—because the overhead generated by the algorithms is small in comparison to the load applied to

**Table 7.7** Summary of Multiple Application Experiments with CPU Loaded

| Algorithm | Application | Mean Frame Rate (FPS) | Policy Violations (%) | Standard Deviation (FPS) | CPU Overhead (%) |
|---|---|---|---|---|---|
| Metric-Heuristic | RTVC | 9.41 | 100.00% | 2.19 | 3.85% |
| Metric-Heuristic | MPEG | 17.29 | 100.00% | 1.99 | 3.85% |
| Multidimensional Search | RTVC | 12.40 | 100.00% | 2.12 | 2.22% |
| Multidimensional Search | MPEG | 18.61 | 80.54% | 1.72 | 2.22% |
| Round Robin | RTVC | 12.77 | 100.00% | 2.18 | 2.12% |
| Round Robin | MPEG | 18.44 | 99.65% | 1.89 | 2.12% |

the CPU in this case, we were able to realise these benefits in this batch of experiments. Consequently, management still offers improvements, even in cases when there is little hope of meeting quality under huge resource demands.

As another interesting note, we can see that deviation in delivered quality was lower when the CPU was loaded and management was in place, compared to the previous set of experiments when no load was applied. With CPU load applied, any spare CPU cycles that are leaked when a scheduled process with quality of service constraints blocks, are consumed by the CPU load generator. Without any load, these spare cycles fall to descheduled processes being managed; since these cycles are only left-overs, they come at irregular intervals, resulting in higher deviations. (Using a leakage protection thread in the scheduler, as discussed in Section 6.4.4, would reduce deviations by preventing such cycle leakage, but would waste these leaked cycles in the process.)

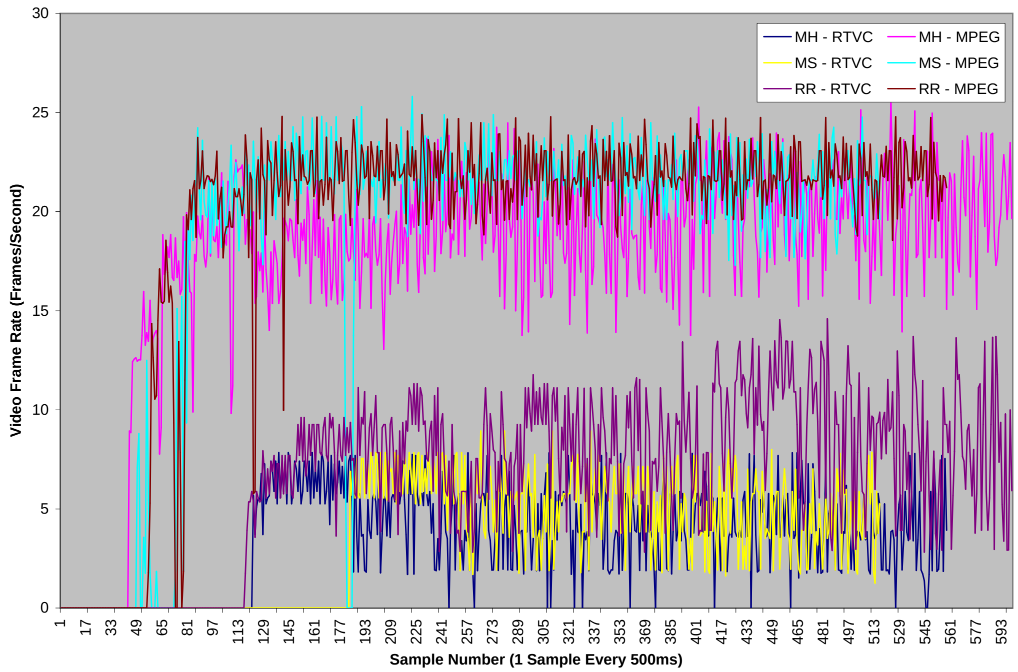
#### 7.4.1.4 Multiple Resource Load Comparison Experiments

The previous set of experiments showed that neither the MPEG player nor RTVC applications can meet quality requirements when executed together under a heavy CPU load. Once again, it is expected that adding memory load will not improve the situation, but will likely worsen things further. The results should, nevertheless, prove interesting.

To investigate this, we have again chosen to load the CPU resource with our load generator set to induce a CPU load of 8, and to use our memory load generator to induce the same heavy paging load that was produced in earlier experiments. Each experiment was also replicated.

Figure 7.12 presents a representative run from each application managed by each resource management algorithm when load has been applied to both the CPU and memory, with frame rate sampled every 500 milliseconds. Key results are summarized in Table 7.8, including the mean frame rate (in frames per second), percentage of quality samples violating

**Figure 7.12** Multiple Application Experiments with Both Resources Loaded




quality of service requirements, standard deviation of quality samples, and CPU overhead. As had been thought, neither of the applications could meet quality requirements when managed by any of the algorithms.

Once again, on average, the metric-heuristic approach fared worse than the other two algorithms. The round robin approach again did better than the metric-heuristic approach, and at times better than the multidimensional search approach in some respects. The multidimensional search approach, however, again fared better on average than the other two approaches in most categories of results.

It is interesting to note that, for all three algorithms, the number of policy violations for the MPEG player application decreased when memory load was induced in addition to CPU load. At the same time, however, the average distance between delivered and expected quality increased. It is believed that the addition of memory load increased the

**Table 7.8** Summary of Multiple Application Experiments with Both Resources Loaded

| Algorithm | Application | Mean Frame Rate (FPS) | Policy Violations (%) | Standard Deviation (FPS) | CPU Overhead (%) |
|---|---|---|---|---|---|
| Metric-Heuristic | RTVC | 2.36 | 100.00% | 3.05 | 3.72% |
| Metric-Heuristic | MPEG | 14.97 | 90.81% | 6.67 | 3.72% |
| Multidimensional Search | RTVC | 3.41 | 100.00% | 2.81 | 1.96% |
| Multidimensional Search | MPEG | 17.93 | 72.79% | 5.49 | 1.96% |
| Round Robin | RTVC | 6.60 | 100.00% | 2.86 | 1.76% |
| Round Robin | MPEG | 17.57 | 88.98% | 3.99 | 1.76% |

variability in the MPEG player's performance while still driving the average frame rate lower (this is also indicated by increased standard deviations observed during the experiments). By increasing the variability, the chances of periodically hitting the target quality range increased, allowing for fewer quality violations. It is also interesting to note that all three algorithms provided fewer violations and improved quality on average substantially than when no management was in place.

### 7.4.2 Multiple Stand-Alone MPEG Players Experiments

In the previous collection of experiments, we experimented with both the RTVC and MPEG player applications together. It did not take long to realise, however, that there were simply insufficient resources to support the quality of service expectation policies for these applications at the same time. The experiments still proved interesting, shedding light on how the various resource management algorithms being studied behaved under such scenarios.

To investigate the management of multiple applications, when quality of service can, in fact, be delivered to both applications at the same time, we chose to execute two copies of the MPEG player application instead. Each would have its own private movie to display, to prevent the effects of disk caching interfering with the results.

The target workstation was again `farquarson`. Both instances of the MPEG player application were instructed to display a copy of the same 160×120 8-bit video from a file. Both instances of this application were given a policy dictating that they must maintain a frame rate of 24 frames per second, plus or minus 2 frames (giving an acceptable range of 22 to 26 frames per second).

**Table 7.9** Summary of Multiple MPEG Application Base Case Experiments

| Load Applied | Application | Mean Frame Rate (FPS) | Policy Violations (%) |
|---|---|---|---|
| None | MPEG 1 | 21.56 | 23.85% |
| None | MPEG 2 | 21.65 | 26.04% |
| CPU only | MPEG 1 | 7.12 | 86.90% |
| CPU only | MPEG 2 | 7.01 | 87.46% |
| Both CPU and memory | MPEG 1 | 2.59 | 96.52% |
| Both CPU and memory | MPEG 2 | 2.45 | 97.28% |

#### 7.4.2.1 Base Case Experiments

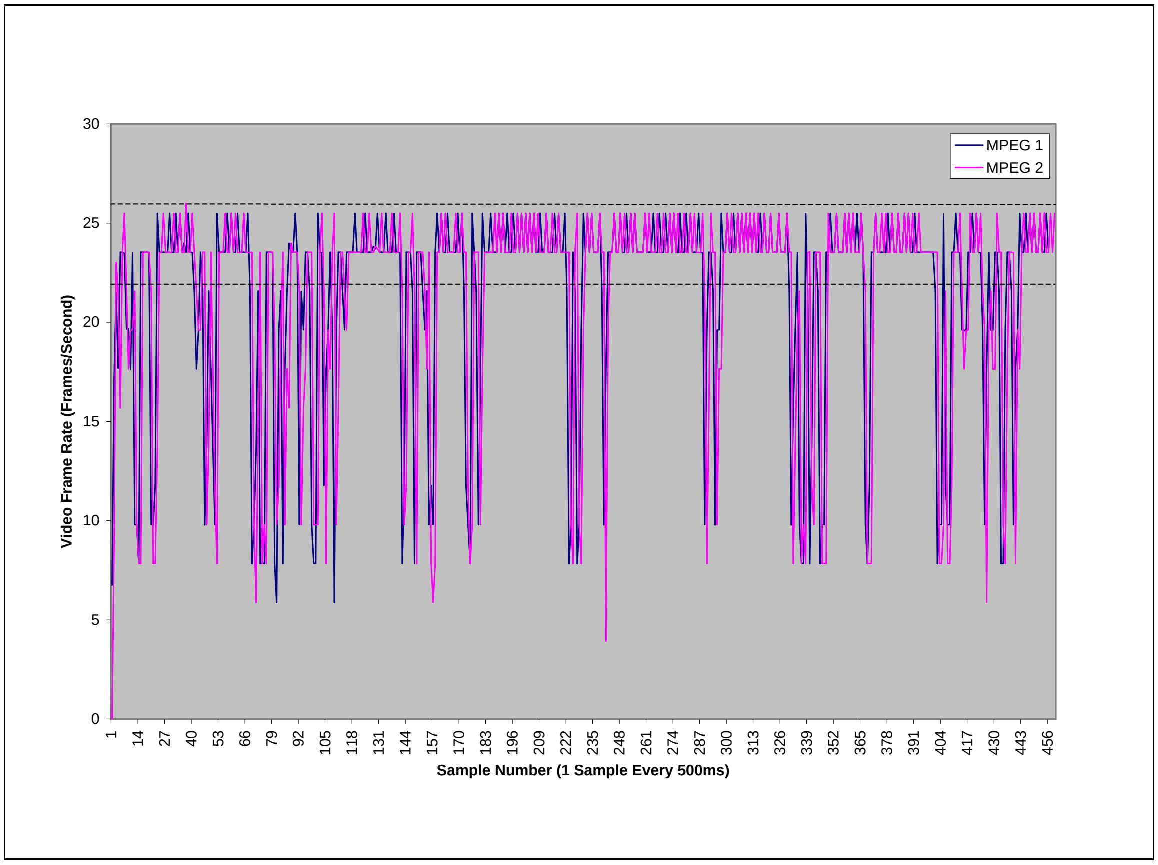
The first set of experiments executed was again a set of experiments to determine the behaviour of the MPEG player applications under different conditions without the benefit of our quality of service management. These experiments could then be used both as a base line for comparison, to determine the improvements brought by our management, and for a simple regression analysis to determine the initial resource split for the multidimensional search algorithm as discussed in Section 5.3.

For these experiments, we again chose two levels of load to be applied to each of two resources, CPU cycles and memory, using our load generators. For the CPU, we chose loads of 0 (no load induced by our generator) and 8. For memory, we chose no load and the same very high paging load as before. Each of the experiments was again replicated.

Figure 7.13, Figure 7.14, and Figure 7.15, present a representative run from each application in the no load, CPU load, and both load configurations. Key results are summarized in Table 7.9, including the mean frame rate delivered (in frames per second), and the percentage of quality samples that violated quality of service requirements. Once again, frame rates were sampled every 500 milliseconds.

With no load applied, quality of service was generally within requirements most of the time. However, violations of those requirements produced enough variation in results to lower the mean delivered quality for both instances of the MPEG application just below requirements. Variation only increased when CPU load was applied, and performance dropped substantially. With memory load added, performance dropped further to the point where both instances of the applications made virtually no progress after the load generators were ramped up to full load. Nevertheless, experiment results were more promising than in the previous sets of experiments with both the RTVC and MPEG application together. Hopefully, this time, the presence of our quality of service management system will result in acceptable quality meeting quality of service requirements.

**Figure 7.13** Multiple MPEG Application Base Case Experiments with No Resources Loaded

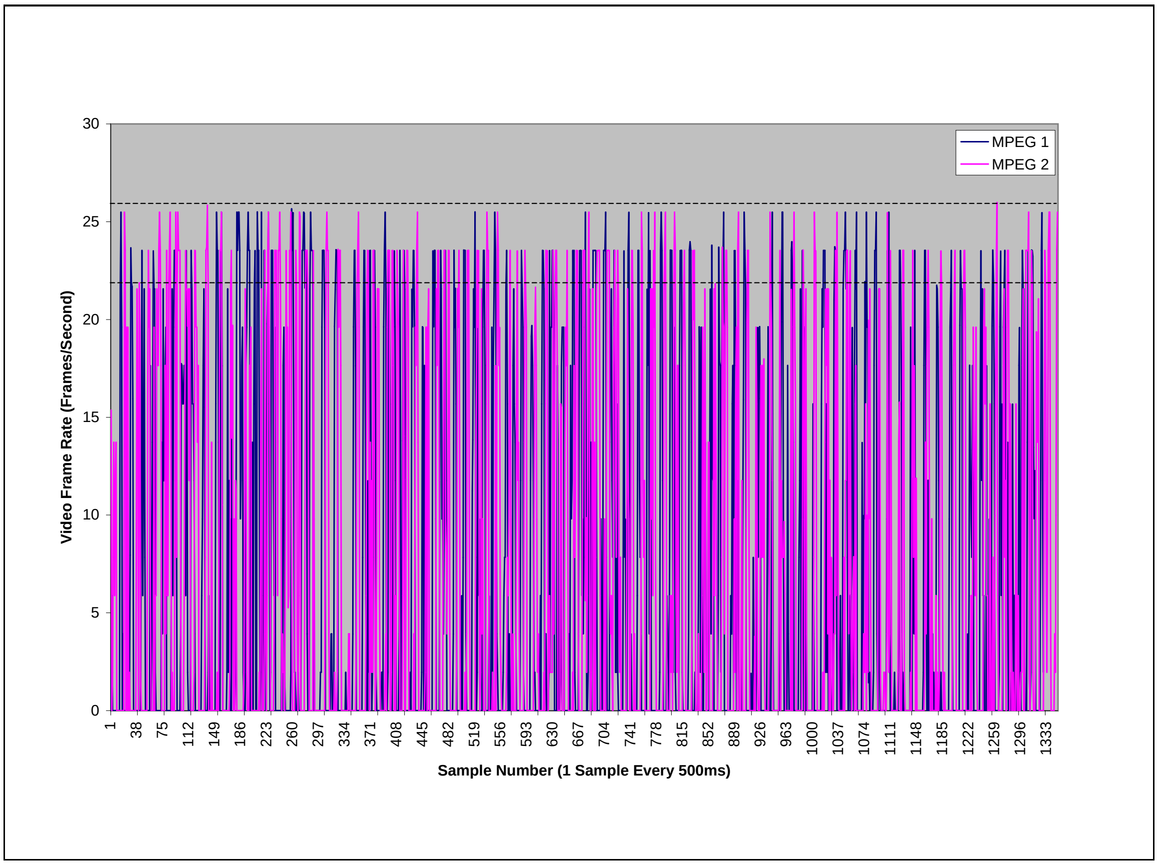



It is important to note that the difference in performance of the two instances of the application are due to the fact that instance 1 was always executed before instance 2. Consequently, instance 1 has first chance to access resources, potentially preventing instance 2 from obtaining them. As a result, the performance of instance 2 is usually below that of instance 1. This phenomenon will occur in later multiple application experiments, particularly when resources are managed by our quality of service management system on a first-come, first-serve basis.

#### 7.4.2.2 No Resource Load Comparison Experiments

Experiments conducted in the previous section have demonstrated that, at no resource load, it is nearly possible for two instances of the MPEG player application to converge and

**Figure 7.14** Multiple MPEG Application Base Case Experiments with CPU Loaded

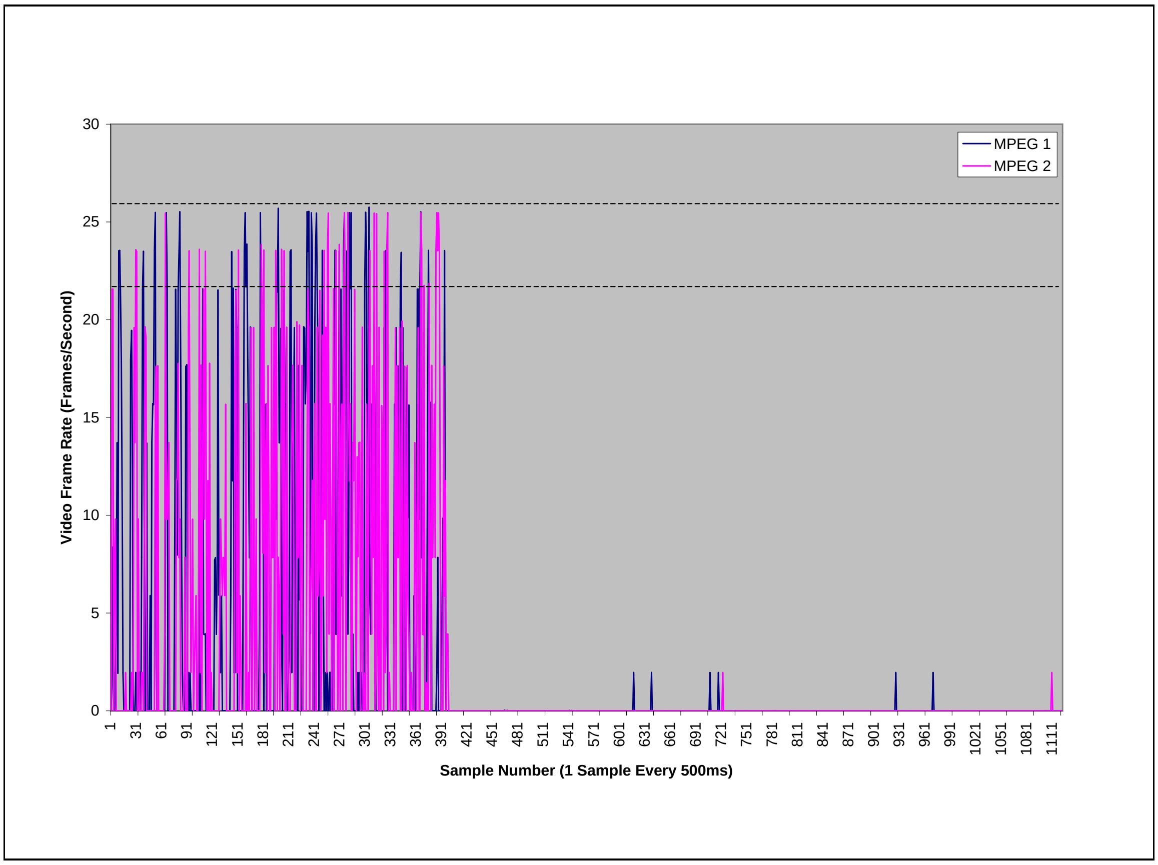



produce stable quality at the same time. With that in mind, however, we noted that there was still a great deal of policy violations and variability in results. It would be interesting to see if any of the management algorithms could reduce these effects and improve quality of service.

Figure 7.16 presents a representative run from each MPEG application managed by each resource management algorithm when no load has been applied, with frame rate sampled every 500 milliseconds. Key results are summarized in Table 7.10, including the convergence time, mean frame rate (in frames per second), percentage of quality samples violating quality of service requirements, standard deviation of quality samples, and CPU overhead. For all cases, the mean frame rate, policy violations, and standard deviation are listed only for samples after convergence.

In this case, both instances could converge and were stable under each resource management

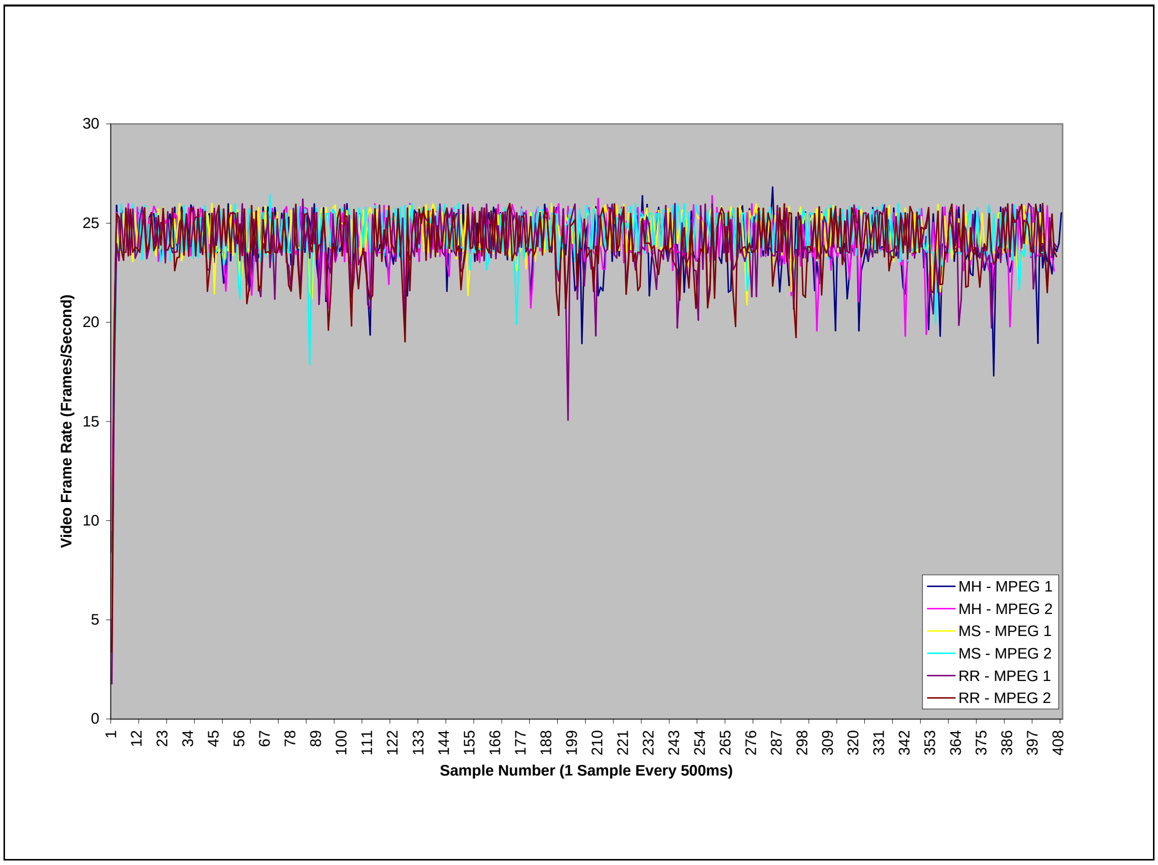
**Figure 7.15** Multiple MPEG Application Base Case Experiments with Both Resources Loaded




algorithm. The metric-heuristic algorithm tended to out-perform the other algorithms in every category, on average. The round robin approach fared next best in most respects, with the multidimensional search approach following close behind. It is important to note, however, that in some categories, the difference between the best result and worst result were quite minimal.

It is interesting to note that, in this set of experiments, all resource management algorithms provided converged quality quicker than when management was absent. Furthermore, this quality was also significantly more stable, and on average higher, than when management was not present.

**Figure 7.16** Multiple MPEG Application Experiments with No Resources Loaded




#### 7.4.2.3 Single Resource Load Comparison Experiments

When load was applied to the CPU and our quality of service management system was not in place, neither MPEG player could converge or provide stable quality. This set of experiments replicates that scenario, except that management is provided by our three selected algorithms.

Figure 7.17 presents a representative run from each instance of the MPEG application managed by each resource management algorithm when load has been applied to the CPU, with frame rate sampled every 500 milliseconds. Key results are summarized in Table 7.11, including the convergence time, mean frame rate (in frames per second), percentage of quality samples violating quality of service requirements, standard deviation of quality samples, and CPU overhead. For all cases, the mean frame rate, policy violations, and standard deviation are listed only for samples after convergence.

**Table 7.10** Summary of Multiple MPEG Application Experiments with No Resources Loaded

| Algorithm | Application | Convergence Time (%) | Mean Frame Rate (FPS) | Policy Violations (%) | Standard Deviation (FPS) | CPU Overhead (%) |
|---|---|---|---|---|---|---|
| Metric-Heuristic | MPEG 1 | 0.73% | 24.45 | 1.98% | 0.77 | 0.97% |
| Metric-Heuristic | MPEG 2 | 0.74% | 24.45 | 2.70% | 0.81 | 0.97% |
| Multidimensional Search | MPEG 1 | 0.75% | 23.98 | 8.64% | 1.01 | 1.04% |
| Multidimensional Search | MPEG 2 | 0.75% | 23.97 | 12.28% | 1.14 | 1.04% |
| Round Robin | MPEG 1 | 0.74% | 24.18 | 5.19% | 0.89 | 0.99% |
| Round Robin | MPEG 2 | 0.74% | 24.00 | 8.64% | 1.00 | 0.99% |

**Table 7.11** Summary of Multiple MPEG Application Experiments with CPU Loaded

| Algorithm | Application | Convergence Time (%) | Mean Frame Rate (FPS) | Policy Violations (%) | Standard Deviation (FPS) | CPU Overhead (%) |
|---|---|---|---|---|---|---|
| Metric-Heuristic | MPEG 1 | 11.11% | 24.28 | 8.30% | 0.82 | 1.81% |
| Metric-Heuristic | MPEG 2 | 13.95% | 24.04 | 10.41% | 1.09 | 1.81% |
| Multidimensional Search | MPEG 1 | 31.03% | 23.94 | 28.69% | 1.39 | 1.95% |
| Multidimensional Search | MPEG 2 | 33.06% | 23.89 | 32.83% | 1.74 | 1.95% |
| Round Robin | MPEG 1 | 17.83% | 24.02 | 19.00% | 1.33 | 2.06% |
| Round Robin | MPEG 2 | 20.22% | 23.98 | 21.22% | 1.61 | 2.06% |

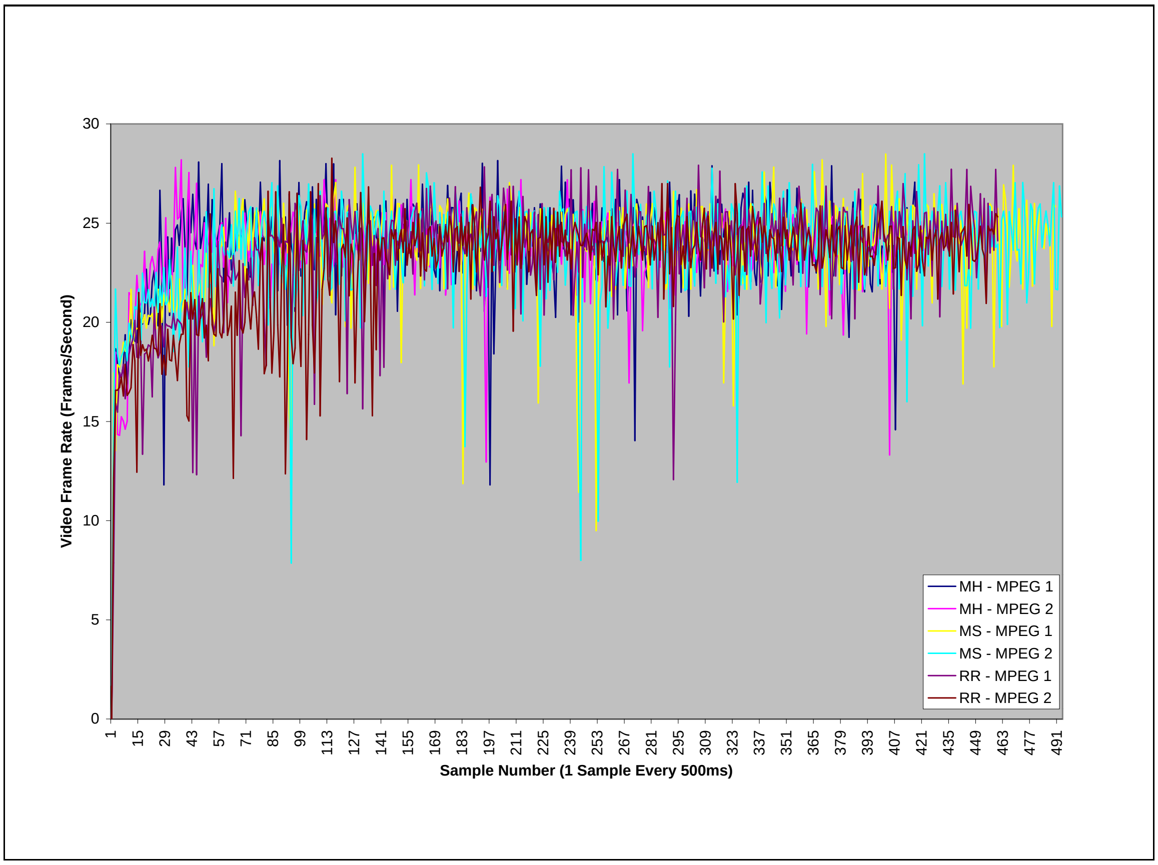
With the presence of management, both instances of the MPEG player could converge and provide stability under each resource management algorithm. The metric-heuristic algorithm again fared better than the other two algorithms in all categories. The round robin approach was again better than the multidimensional search approach in most respects on average. With load applied, differences in the performance of each algorithm grew larger, particularly in terms of convergence time and stability. Mean quality delivered after convergence, however, remained reasonably consistent across the algorithms.

#### 7.4.2.4 Multiple Resource Load Comparison Experiments

The experiments discussed in the previous experiment showed that management could overcome the impact of loading a single resource and provide quality meeting quality of service requirements. In this set of experiments, we investigate whether we can see similar benefits when multiple resources are loaded.

Figure 7.18 presents a representative run from each instance of the MPEG application managed by each resource management algorithm when load has been applied to both the CPU and memory, with frame rate sampled every 500 milliseconds. Key results are

**Figure 7.17** Multiple MPEG Application Experiments with CPU Loaded

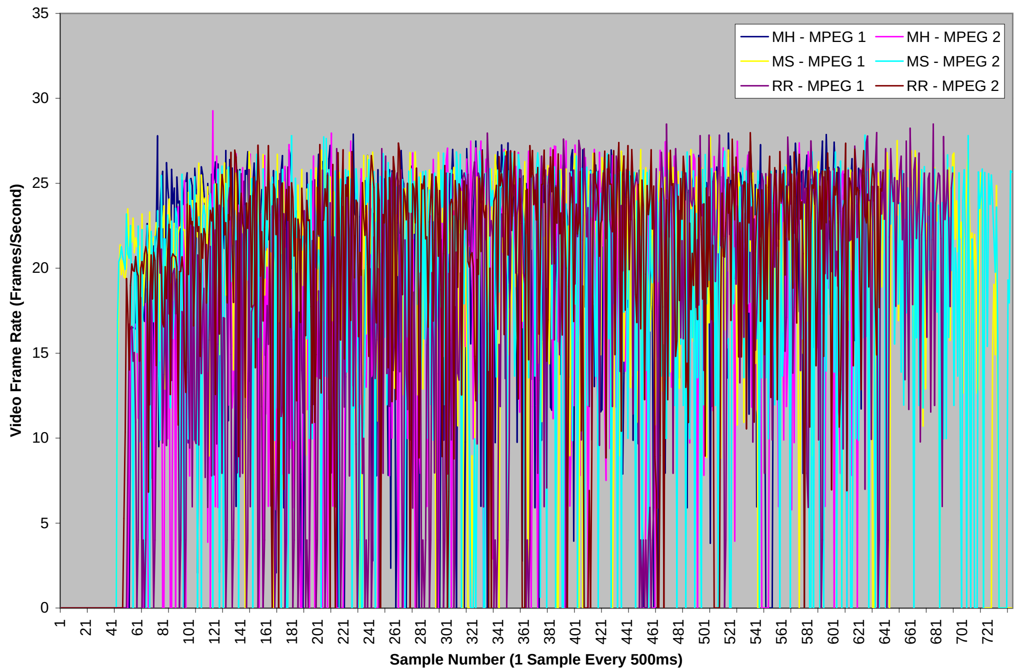



summarized in Table 7.12, including the mean frame rate (in frames per second), percentage of quality samples violating quality of service requirements, standard deviation of quality samples, and CPU overhead.

In these experiments, the presence of management, while improving quality of service substantially, could not make either instance of the application converge or provide stable quality. In an interesting turn of events, the metric-heuristic algorithm fared the worse of three algorithms in most respects where it proved better in earlier experiments. The multidimensional search performed somewhat better than the metric-heuristic algorithm, while the round robin approach fared the best of the three algorithms in most respects.

It is interesting to consider why the algorithms failed to meet quality requirements in this case. A quick analysis indicated that not only did each instance of the MPEG player application receive more CPU cycles than the previous set of experiments when only the

**Figure 7.18** Multiple MPEG Application Experiments with Both Resources Loaded




CPU was loaded (because continued quality complaints resulted in more allocation than before), each MPEG player application was locked entirely resident in physical memory by our memory manager, so lack of memory resources should not hinder performance either. Despite both of these facts, quality curiously still failed to meet requirements.

The key to this mystery lies in the memory load applied to the system. With the memory load in place, there was a high volume of page faulting in the system; our memory manager insulated the managed MPEG players from sharing in the page faulting. However, the other page faults resulted in heavy traffic over the SCSI channel to the system disks where paging areas were located. This traffic interfered with the MPEG players' abilities to access their video streams in a timely and consistent fashion from disks, resulting in missed quality objectives. In effect, the MPEG players had to contend with the memory subsystem, not for physical memory, but rather disk I/O bandwidth. This is an interesting dependency,

**Table 7.12** Summary of Multiple MPEG Application Experiments with Both Resources Loaded

| Algorithm | Application | Mean Frame Rate (FPS) | Policy Violations (%) | Standard Deviation (FPS) | CPU Overhead (%) |
|---|---|---|---|---|---|
| Metric-Heuristic | MPEG 1 | 15.00 | 73.80% | 5.95 | 2.59% |
| Metric-Heuristic | MPEG 2 | 14.84 | 76.20% | 6.07 | 2.59% |
| Multidimensional Search | MPEG 1 | 16.66 | 59.00% | 5.24 | 2.03% |
| Multidimensional Search | MPEG 2 | 15.78 | 61.57% | 5.37 | 2.03% |
| Round Robin | MPEG 1 | 17.53 | 46.34% | 4.54 | 2.24% |
| Round Robin | MPEG 2 | 17.39 | 48.92% | 4.62 | 2.24% |

and highlights the need for the management of all computing resources, even if only a few are apparently loaded and scarce.

It is important to note that while quality did not meet targets for any algorithm, each algorithm provided substantial improvements over the no management scenario, despite the above difficulties. This, in itself, provides some consolation.

## 7.5 Distributed Applications

The previous experiments focussed on the performance of a single or multiple applications on a single host. In the remaining experiments we examine several scenarios involving the distributed versions of our applications across multiple hosts. In the first batch of experiments, discussed in this section, we examine the quality delivered by a single distributed application. In later sections, we will examine the management of multiple distributed applications at once.

### 7.5.1 Distributed RTVC Experiments

To examine the effects of management on a distributed application, we chose the RTVC video conferencing application as our target for experimentation. The video transmitter was executed on `farquarson`, while the receiver to display the transmitted video was executed on `strawberry`.

In all cases, the RTVC application was instructed to capture, transmit, and display 18000 frames of a 320×240 8-bit video stream from a camera at the department's operations counter, connected to the SunVideo board. Also in all cases, this application was given its usual quality of service expectation policy dictating that it must maintain a frame rate of

26 frames per second, plus or minus 2 frames (giving an acceptable range of 24 to 28 frames per second).

Previously, in single host experiments, only one quality of service manager was necessary—a host manager on the machine executing the applications. (A domain manager was also executed for completeness, but was not active during the experiments.) With distributed applications, we now need a host manager on each host involved, as well as the domain manager. In this case, we executed the domain manager on the same host as the RTVC transmitter, `farquarson`.

#### 7.5.1.1 Base Case Experiments

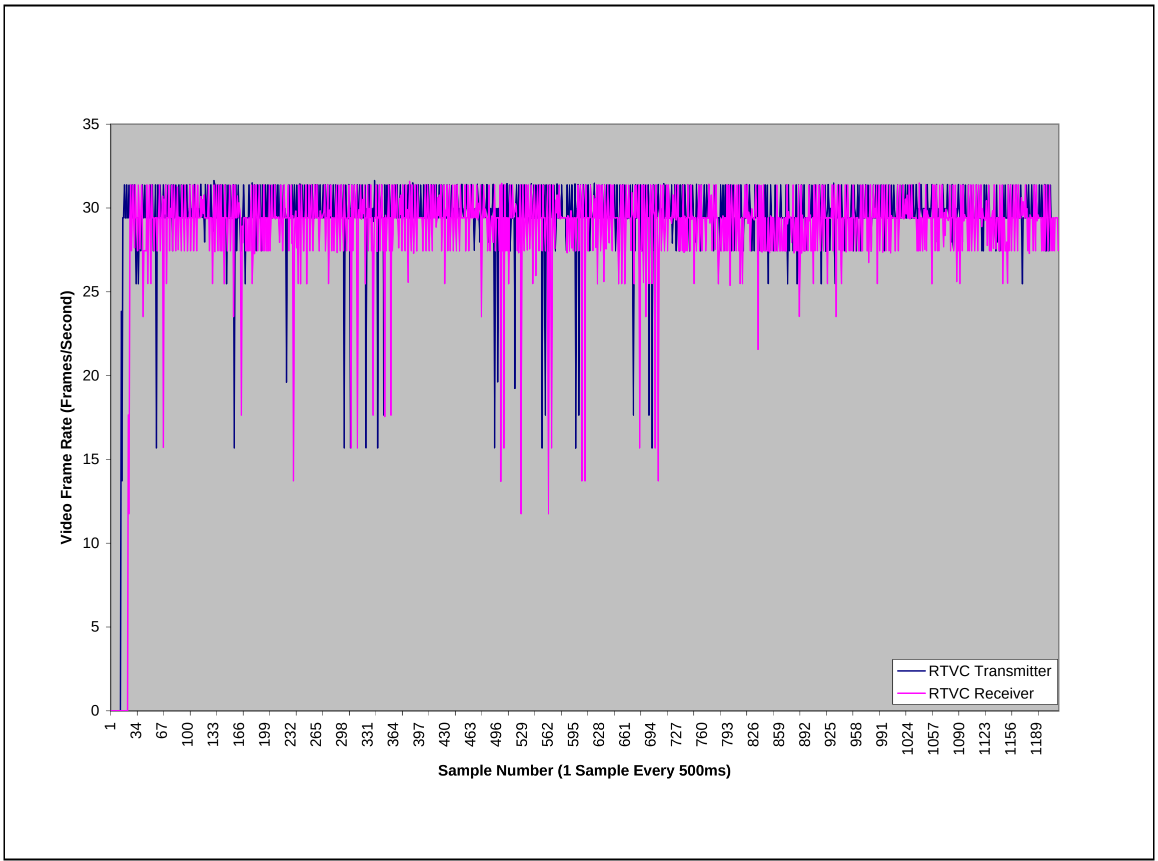
The first set of experiments executed was again a set of experiments to determine the behaviour of the distributed RTVC application under different conditions without the benefit of our quality of service management. These experiments could then be used both as a base line for comparison, to determine the improvements brought by our management, and for a simple regression analysis to determine the initial resource split for the multidimensional search algorithm as discussed in Section 5.3.

For these experiments, we again chose two levels of load to be applied to each of two resources, CPU cycles and memory, using our load generators. For the CPU, we chose loads of 0 (no load induced by our generator) and 8. For memory, we chose no load and the same very high paging load as before. By combining these loads, we have four possible load combinations: no load, CPU load only, memory load only, and both CPU and memory loads. For simplicity, we decided to apply the same load to both hosts involved in the experiments at the same time. This reduced the number of experiment configurations from 16 down to 4, making analyses more manageable. Each of these experiments was also replicated.

Figure 7.19, Figure 7.20, Figure 7.21, and Figure 7.22 present representative runs from the application in the no load, CPU load, memory load, and both load configurations, with frame rate sampled every 500 milliseconds. Key results are summarized in Table 7.13, including the mean frame rate delivered (in frames per second), and the percentage of quality samples that violated quality of service requirements. Although the quality measure of prime importance is the frame rate delivered by the receiver, as this is what the end user experiences, we consider the quality measured by both the transmitter and receiver in these figures and this table for comparison purposes.

The distributed RTVC application was unable to converge to meet quality requirements in any load condition. When no load was applied, quality exceeded requirements (this leads to an interesting scenario in which our management system will need to try to reduce quality

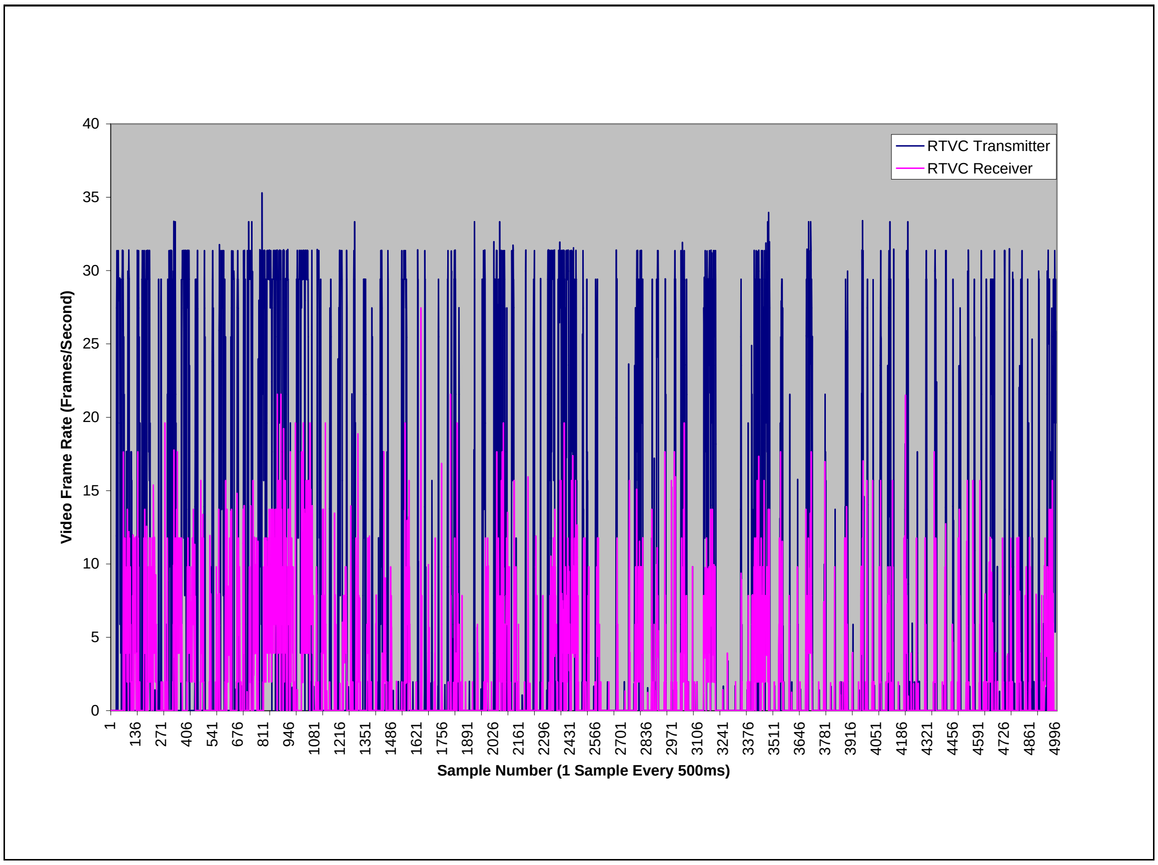
**Figure 7.19** Distributed RTVC Application Base Case Experiments with No Resources Loaded




delivered instead of increase it). When any load was applied, whether CPU or memory alone, or both together, quality of service was abysmal. When both resources were loaded, the RTVC receiver was essentially at a stand-still, and was only able to process any video frames during the initial period in which the load generators were not operating at full load.

It is interesting to note that with the capture and display functionality on separate hosts, the RTVC application was able to increase its performance, but at a cost. Because the transmitter and receiver acted independently with no synchronization, some video frames were lost in transit, since unreliable datagrams were used for video transmission. Frames were lost even in the unloaded scenario. When CPU load was added, the difference in frame rates between the transmitter and receiver resulted in a substantially larger loss of video frames. Frame loss was less severe under a memory load, only because the transmitter was impacted severely by the memory load as well, and could barely manage to transmit

**Figure 7.20** Distributed RTVC Application Base Case Experiments with CPU Loaded

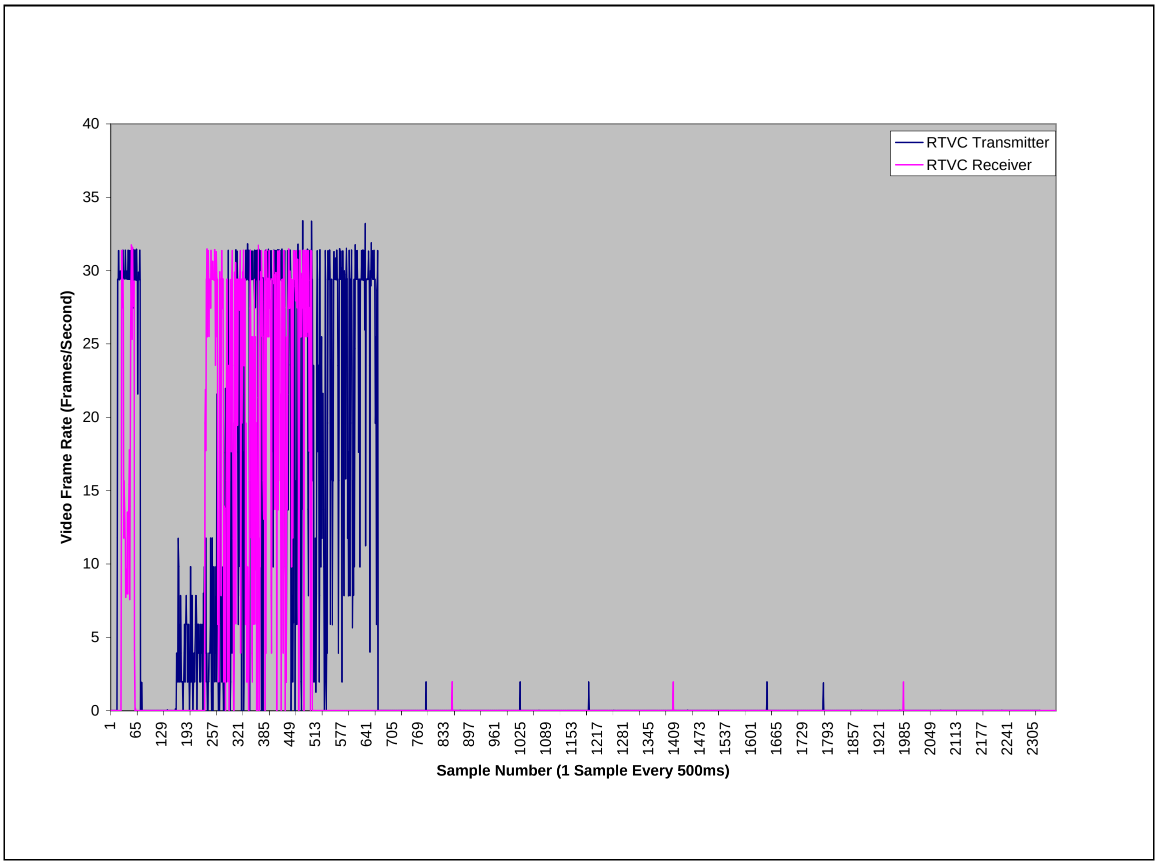



frames itself. When both resources were loaded, nearly all frames transmitted were lost, particularly when the load generators were operating at full load—since the receiver could make virtually no progress, all transmitted frames were dropped and lost.

#### 7.5.1.2 No Resource Load Comparison Experiments

In the previous section, experiments found that the distributed RTVC application actually exceeded quality of service requirements when no load was applied to the hosts the transmitter and receiver were executing on. For a change, this means that our resource management algorithms will have to attempt to provide a reduced level quality compared to what the application can actually produce. It is important to note that this kind of circumstance would rarely occur in practice, as most users do not typically place upper limits on application performance, and in most cases, contention for resources limits actual

**Figure 7.21** Distributed RTVC Application Base Case Experiments with Memory Loaded

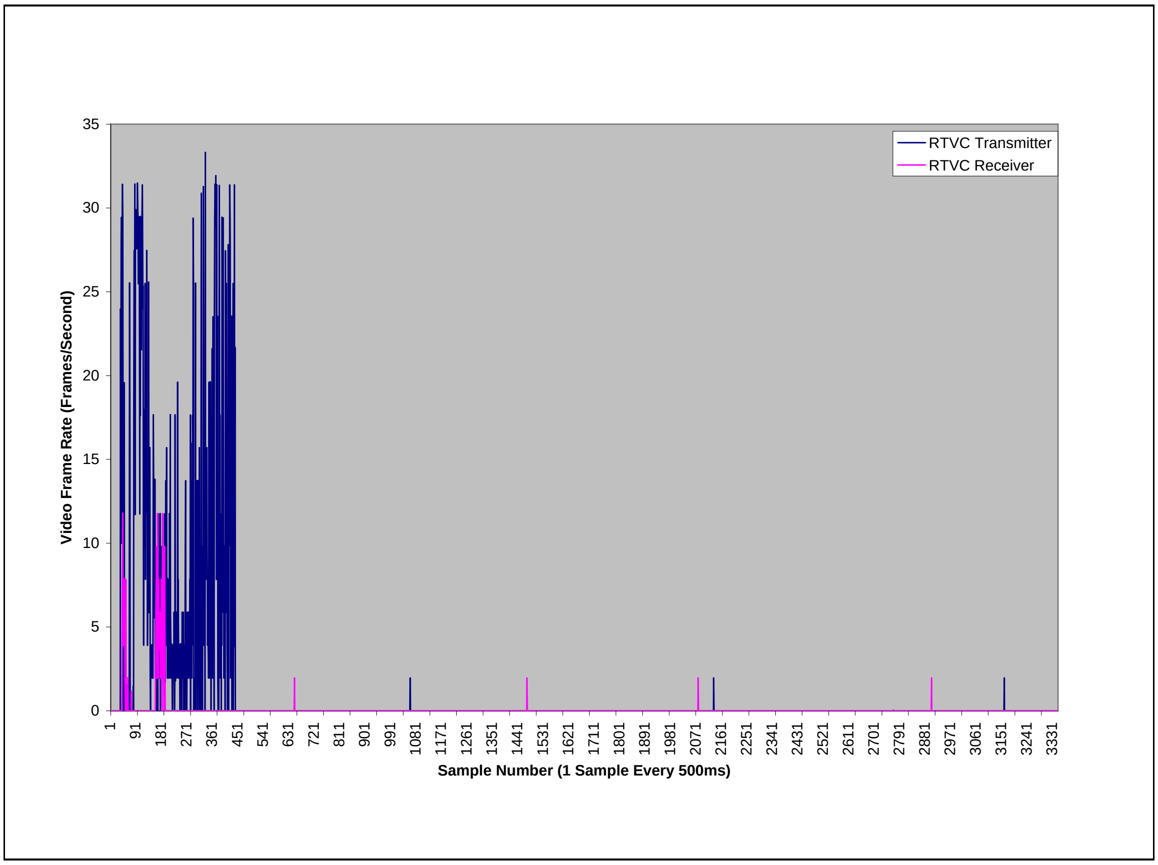



application performance compared to what could theoretically be achieved. Nevertheless, the results should prove interesting.

Figure 7.23 and Figure 7.24 present representative runs from the RTVC application receiver and transmitter respectively, when managed by each resource management algorithm when no load has been applied, with frame rate sampled every 500 milliseconds. Key results are summarized in Table 7.14, including the mean frame rate (in frames per second), percentage of quality samples violating quality of service requirements, standard deviation of quality samples, and CPU overhead. To simplify matters and to highlight the differences between algorithms, transmitter frame rates and other statistics have not been included in this table. (As can be seen from Figure 7.23 and Figure 7.24, the results from the transmitter correspond similarly to the receiver, but are less dramatic.)

As can be seen, the algorithms had difficulty in decreasing quality in a stable fashion, and

**Figure 7.22** Distributed RTVC Application Base Case Experiments with Both Resources Loaded




quality seemed to alternate between too high and too low. Consequently, none of the algorithms were able to converge, even though mean quality delivered was well with policy bounds on average. The round robin approach appeared to perform better than the other algorithms, providing fewer violations, lower deviation, and lower overhead. The multidimensional search approach fared next best, followed by the metric-heuristic approach.

This did indeed prove to be an interesting set of experiments. Each algorithm was able to deliver average quality that met requirements, but individual samples were unable to do so. Quality was simply too unstable to converge. In all fairness to the algorithms, however, they were designed and implemented primarily to increase the quality of service of applications, rather than decrease it. The algorithms, and quality of service managers in general, may require tuning somewhat, if reducing quality becomes a more prevalent task for them.

**Table 7.13** Summary of Distributed RTVC Application Base Case Experiments

| Load Applied | Application | Mean Frame Rate (FPS) | Policy Violations (%) |
|---|---|---|---|
| None | Transmitter | 28.24 | 92.22% |
| None | Receiver | 29.25 | 75.90% |
| CPU only | Transmitter | 7.03 | 96.12% |
| CPU only | Receiver | 2.04 | 99.98% |
| Memory only | Transmitter | 3.75 | 98.73% |
| Memory only | Receiver | 1.20 | 98.84% |
| Both CPU and memory | Transmitter | 1.14 | 99.19% |
| Both CPU and memory | Receiver | 0.03 | 100.00% |

**Table 7.14** Summary of Distributed RTVC Application Experiments with No Resources Loaded

| Algorithm | Mean Frame Rate (FPS) | Policy Violations (%) | Standard Deviation (FPS) | CPU Overhead (%) |
|---|---|---|---|---|
| Metric-Heuristic | 25.74 | 91.65% | 4.71 | 7.39% |
| Multidimensional Search | 26.36 | 88.89% | 4.60 | 6.32% |
| Round Robin | 26.93 | 81.58% | 4.08 | 5.81% |

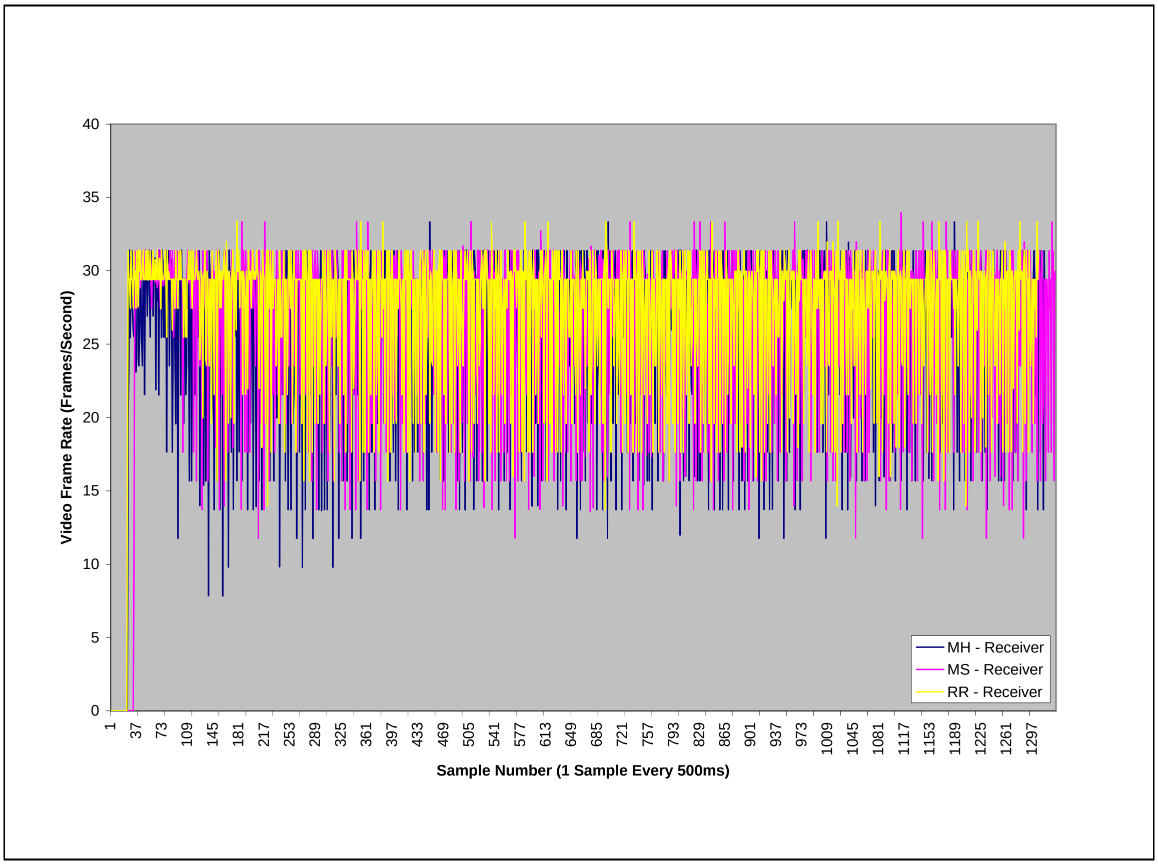
#### 7.5.1.3 Single Resource Load Comparison Experiments

In previous experiments, we found that the addition of CPU load to the distributed RTVC application caused a substantial drop in quality and an increase in dropped video frames. In such a case, quality of service requirements could not be met. We have subsequently repeated those experiments; this time, with the presence of our resource management algorithms. All experiments were once again replicated.

Figure 7.25 presents a representative run from the RTVC application managed by each resource management algorithm with load applied to the CPU, and with frame rate sampled every 500 milliseconds. Key results are summarized in Table 7.15, including the mean frame rate (in frames per second), percentage of quality samples violating quality of service requirements, standard deviation of quality samples, and CPU overhead. To simplify matters to highlight the differences between algorithms, transmitter frame rates and other statistics have not been included this time.

In this case, the resource management algorithms needed to improve quality of service, instead of reducing it, and should not have run into the same problems as in the previous case. Nevertheless, all algorithms ran into difficulties, and were unable to converge or deliver stable quality. All three algorithms performed similarly, with relatively little difference in results. Generally, the round robin approach fared best, followed by the multidimensional

**Figure 7.23** Distributed RTVC Application Receiver Experiments with No Resources Loaded

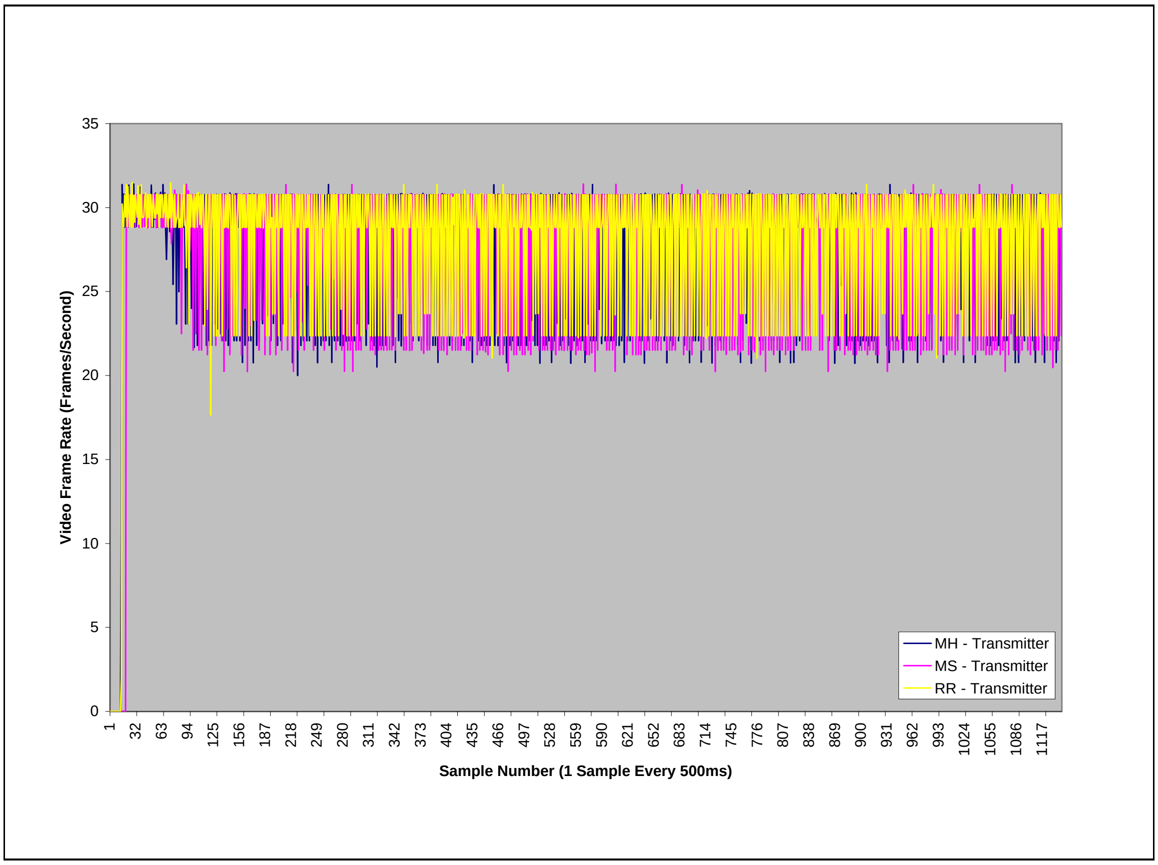



search approach, with the metric-heuristic approach trailing behind the other two.

In this case, it is interesting to examine why quality was not met. We know that the distributed RTVC application is not only capable of meeting the quality policy we have given it, but also exceeding it. Yet, in this case, we cannot even achieve quality. If we examine the performance of the video transmitters (a sample run of which is given in Figure 7.26), we see that the video transmitters are able to converge and deliver stable quality, with a certain degree of variation. So, the RTVC receivers were sent a sufficient video stream to satisfy quality needs, yet quality was still not met. Since we were working with an unloaded, isolated network segment, we know that the network was not the cause of the problem.

Each resource management algorithm gave the RTVC receiver substantially more CPU

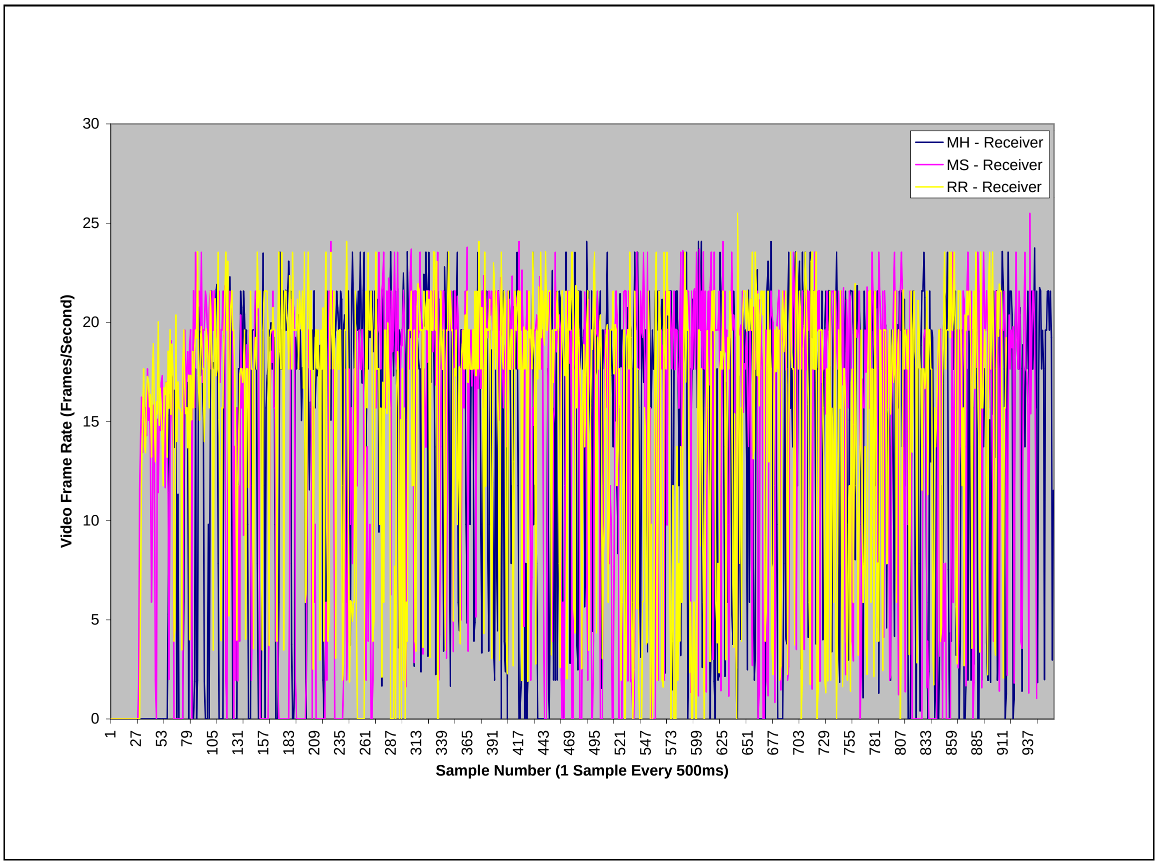
**Figure 7.24** Distributed RTVC Application Transmitter Experiments with No Resources Loaded




cycles than what was consumed by the receiver during the unloaded base case, when quality exceeded targets. Consequently, the receiver *should* have been able to process and display the video quickly enough to meet targets. Further analyses of video frame sequence numbers indicated that a very large number of frames were being lost before arriving at the receiver; with sufficient frames lost, there would not be enough left to achieve a reasonable frame rate.

This leads to two possible explanations for the poor performance. One is that, while the transmitter process could capture frames and issue `sendto` requests to transmit them quickly enough, Solaris does not have enough free cycles left at high CPU loads to actually put them on the network. Consequently, subsequent `sendto` requests by the transmitter will overwrite the queued frames waiting to be sent, causing the frames to be lost. The other possibility is that the frames are actually sent to the receiver's host, but the high CPU load makes Solaris

**Figure 7.25** Distributed RTVC Application Receiver Experiments with CPU Loaded




unable feed them to the receiver's `read` requests quickly enough before they are overwritten by subsequent incoming frames. In either case, since the RTVC application uses UDP as a data transport and delivery of the data is not guaranteed, the system may consider its packets as unimportant and not dedicate resources to delivering them, particularly when resources are scarce. It was found, however, that adjusting outgoing and incoming socket buffer sizes can alleviate these problems somewhat, but the flow of packets still remains irregular and unstable.

As a result, it appears that despite the fact that the RTVC processes received sufficient resources, quality was not delivered because of the impact CPU load had on network subsystem in buffering, sending, and receiving data. Once again, it would appear that the loading of one resource affected the availability and performance of another computing resource. These dependencies indicate that it is vital to manage all computing resources, not

**Table 7.15** Summary of Distributed RTVC Application Experiments with CPU Loaded

| Algorithm | Mean Frame Rate (FPS) | Policy Violations (%) | Standard Deviation (FPS) | CPU Overhead (%) |
|---|---|---|---|---|
| Metric-Heuristic | 13.56 | 99.67% | 5.74 | 7.58% |
| Multidimensional Search | 13.89 | 99.58% | 5.27 | 6.50% |
| Round Robin | 14.56 | 99.58% | 5.05 | 6.29% |

just those which appear to be the most scarce.

As a result of these findings, we felt that continued experimentation with this application under multiple resource loads would reveal no further insights. Consequently, we chose to change applications, and repeat this line of experimentation again and see what else we could find.

### 7.5.2 Distributed Video-on-Demand Experiments

Because of the results of previous experiments with the distributed RTVC application, we switched to the distributed video-on-demand application, to further our investigations. The video-on-demand server was executed on `farquarson`, while the client to request and display the transmitted video was executed on `strawberry`.

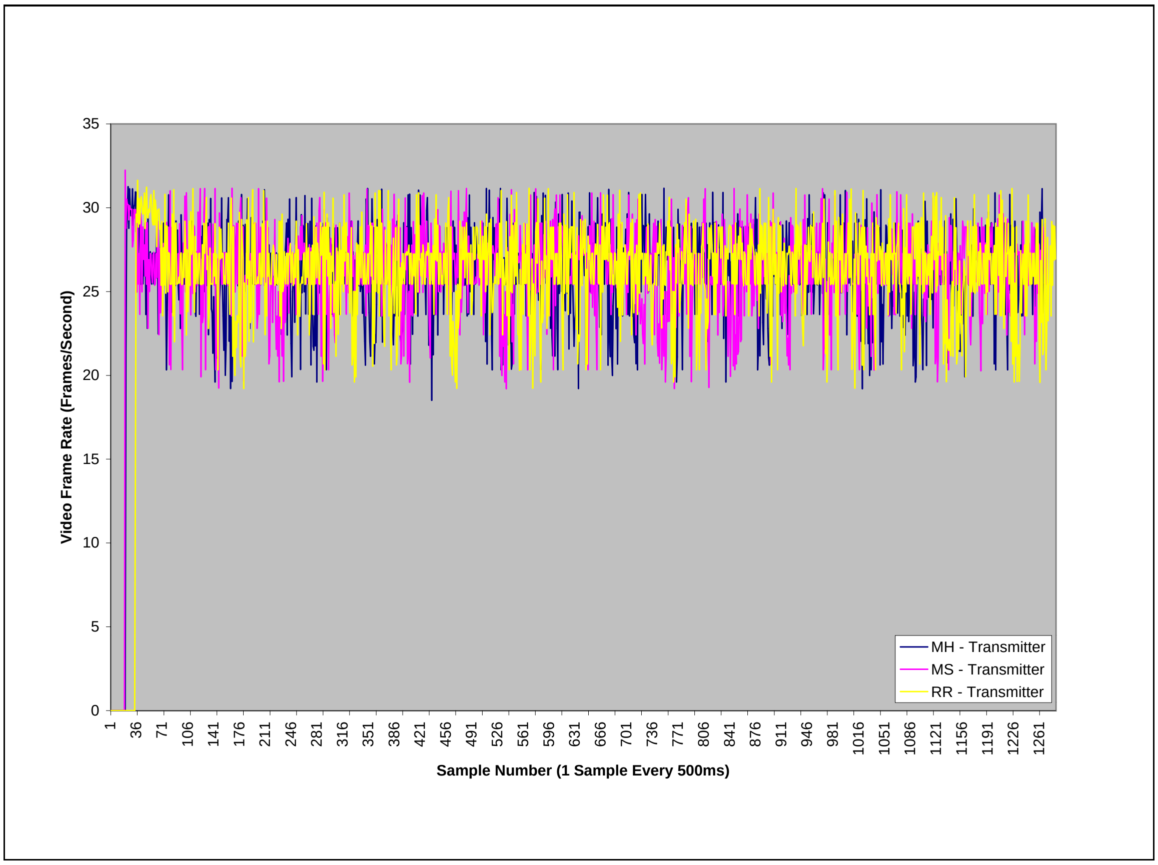
In all cases, the application was instructed to load, transmit, and display a 160×120 8-bit video stream from a file stored on the server's host. Also in all cases, this application was given a quality of service expectation policy dictating that it must maintain a frame rate of 24 frames per second, plus or minus 2 frames (giving an acceptable range of 22 to 26 frames per second).

As with the previous set of distributed experiments, we need a host manager on each host involved in these experiments, as well as a domain manager. In this case, we executed the domain manager on the same host as the video-on-demand server, `farquarson`.

#### 7.5.2.1 Base Case Experiments

The first set of experiments executed was again a set of experiments to determine the behaviour of the video-on-demand application under different conditions without the benefit of our quality of service management. These experiments could then be used both as a base line for comparison, to determine the improvements brought by our management, and for a simple regression analysis to determine the initial resource split for the multidimensional search algorithm as discussed in Section 5.3.

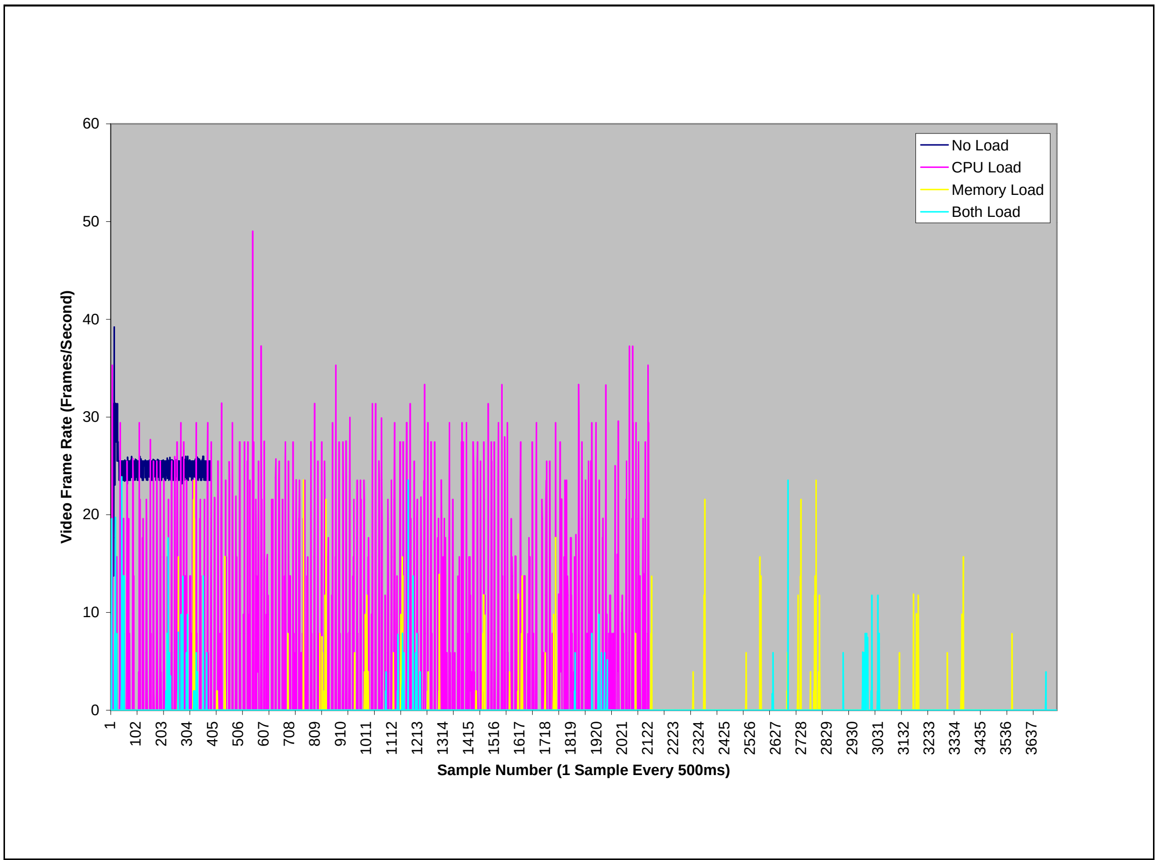
**Figure 7.26** Distributed RTVC Application Transmitter Experiments with CPU Loaded




For these experiments, we again chose two levels of load to be applied to each of two resources, CPU cycles and memory, using our load generators. For the CPU, we chose loads of 0 (no load induced by our generator) and 8. For memory, we chose no load and the same very high paging load as before. By combining these loads, we have four possible load combinations: no load, CPU load only, memory load only, and both CPU and memory loads. For simplicity, we decided to apply the same load to both hosts involved in the experiments at the same time. This reduced the number of experiment configurations from 16 down to 4, making analyses more manageable. Each of these experiments was again replicated.

Figure 7.27 presents a representative run from each application in the different load configurations, with frame rate sampled every 500 milliseconds. Key results are summarized in Table 7.16, including the convergence time, mean frame rate delivered (in frames per sec-

**Figure 7.27** Video-on-Demand Application Base Case Experiments




ond), and the percentage of quality samples that violated quality of service requirements. When quality was able to converge and meet requirements, values reported apply to quality samples after convergence. Unlike the RTVC application, the video-on-demand application can only report quality metrics from the client side (the server only deals with a compressed MPEG bit stream from its files, and does not recognize frame boundaries necessary to determine and report a frame rate).

The video-on-demand application could only converge and be stable when no load was applied to either host it was executing on. With no load applied, quality delivered was practically flawless, especially after convergence. Quality degraded substantially when CPU load was applied, and worsened when memory load was applied. quality of service was at its worst with both CPU and memory loaded; at such a high load, the application was virtually unable to make any progress at all.

**Table 7.16** Summary of Video-on-Demand Application Base Case Experiments

| Load Applied | Convergence Time (%) | Mean Frame Rate (FPS) | Policy Violations (%) |
|---|---|---|---|
| None | 7.58% | 25.00 | 0.00% |
| CPU only | N/A | 4.05 | 96.48% |
| Memory only | N/A | 0.27 | 99.91% |
| Both CPU and memory | N/A | 0.16 | 99.92% |

**Table 7.17** Summary of Distributed Video-on-Demand Application Experiments with CPU Loaded

| Algorithm | Convergence Time (%) | Mean Frame Rate (FPS) | Policy Violations (%) | Standard Deviation (FPS) | CPU Overhead (%) |
|---|---|---|---|---|---|
| Metric-Heuristic | 6.41% | 24.54 | 21.86% | 1.23 | 0.89% |
| Multidimensional Search | 90.93% | 23.91 | 37.30% | 0.67 | 5.97% |
| Round Robin | 52.20% | 24.19 | 27.78% | 2.77 | 3.66% |

#### 7.5.2.2 Single Resource Load Comparison Experiments

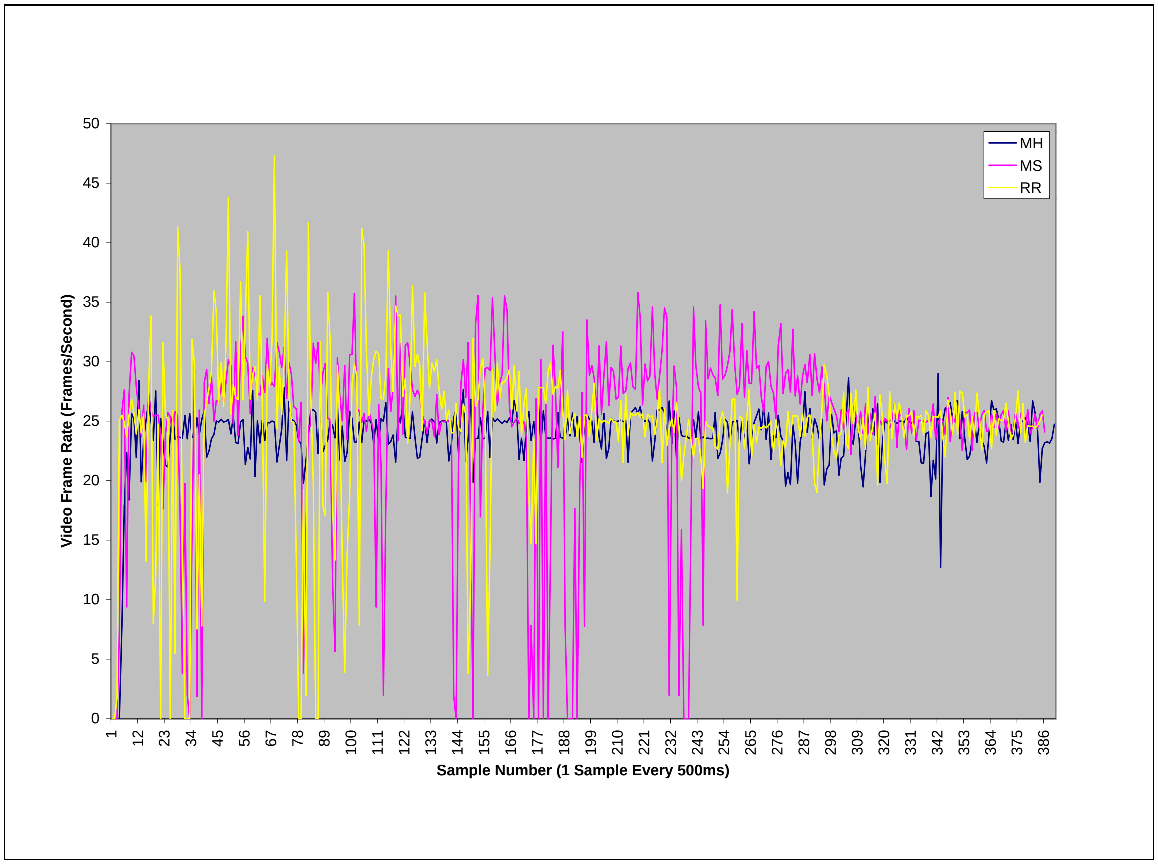
Observing that quality was met nicely in unloaded circumstances, we decided to not repeat this scenario with management present. In such a case, the host and domain managers in place would simply sit idle, not generating any overhead, as the absence of policy violations would not trigger the management into action. Consequently, the results would be precisely the same as before, as previous experiments have indicated.

Instead, we decided to directly proceed with more interesting experiments where the algorithms would need to work to provide quality. Such a case would be managing resources in face of a single loaded resource–the CPU. We repeated the same high CPU load as in the base case experiments, and again replicated the experiments.

Figure 7.28 presents a representative run from the video-on-demand application managed by each resource management algorithm with load applied to the CPU, and with frame rate sampled every 500 milliseconds. Key results are summarized in Table 7.17, including the convergence time, mean frame rate (in frames per second), percentage of quality samples violating quality of service requirements, standard deviation of quality samples, and CPU overhead. The mean frame rate, policy violations, and standard deviation are listed for all samples after convergence.

All three algorithms were able to converge and deliver stable quality, although some barely managed to do so. The metric-heuristic approach fared the best of the three algorithms, in most respects. The round robin approach took considerably longer to converge than the metric-heuristic approach, but still managed to out-perform the multidimensional search

**Figure 7.28** Distributed Video-on-Demand Application Experiments with CPU Loaded




approach. The multidimensional search approach performed the worst of the three algorithms, failing to converge at all in nearly half of replications, and only barely managing to converge at all in the others. While it posted the lowest post-convergence standard deviation of quality samples, we must remember that this deviation covers a much smaller number of samples than the corresponding results for the other algorithms, and is not a fair basis of comparison.

#### 7.5.2.3 Multiple Resource Load Comparison Experiments

In the previous section, we saw the improvements management brought to the video-on-demand application when under a single resource load. In this section, we investigate the effectiveness of managing this distributed application when under multiple resource loads. Once again, all experiments are replicated.

**Figure 7.29** Distributed Video-on-Demand Application Experiments with Both Resources Loaded

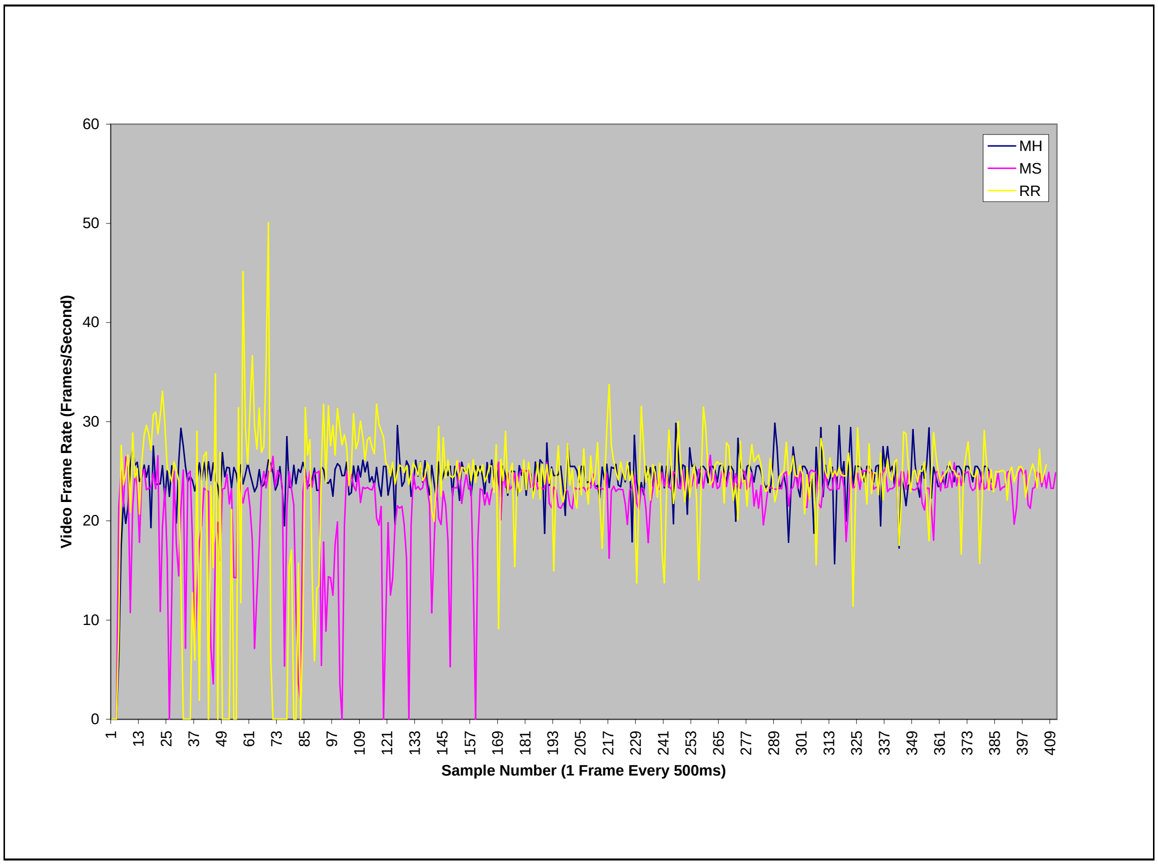



Figure 7.29 presents a representative run from the video-on-demand application managed by each resource management algorithm with load applied to both the CPU and memory, and with frame rate sampled every 500 milliseconds. Key results are summarized in Table 7.18, including the convergence time, mean frame rate (in frames per second), percentage of quality samples violating quality of service requirements, standard deviation of quality samples, and CPU overhead. The mean frame rate, policy violations, and standard deviation are listed for all samples after convergence.

Once again, all three algorithms were able to converge and deliver stable quality. The metric-heuristic approach again provided the best overall quality of service in most respects. The round robin approach fared next best, followed by the multidimensional search approach.

It is interesting to observe that quality of service for both the round robin and multidimen-

**Table 7.18** Summary of Distributed Video-on-Demand Application Experiments with Both Resources Loaded

| Algorithm | Convergence Time (%) | Mean Frame Rate (FPS) | Policy Violations (%) | Standard Deviation (FPS) | CPU Overhead (%) |
|---|---|---|---|---|---|
| Metric-Heuristic | 8.62% | 24.53 | 23.11% | 1.28 | 0.94% |
| Multidimensional Search | 41.61% | 24.06 | 27.37% | 1.03 | 3.52% |
| Round Robin | 29.48% | 24.42 | 28.13% | 2.04 | 2.01% |

sional search approaches improved substantially in these experiments, even though memory load was added to the systems under experimentation. This is likely because algorithm iterations spent on allocating client and server memory are now effective in improving quality of service delivery. When load was only applied to the CPU, both of these algorithms spent additional time in allocating memory resources when they were not scarce. This is similar to the effects noted in Section 7.3.2.2. The metric-heuristic approach is not affected in the same way, since its heuristics guide resource selection, and are less likely to choose resources not needing adjustment, even though each adjustment tends to take slightly longer.

As an important note, we found another interesting aberration that comes from managing distributed applications. In some cases, it is possible under heavy loading for quality of service to develop an increasing oscillation and destabilize, as shown in the sample run in Figure 7.30. This sample came from the metric-heuristic algorithm when both resources were loaded. This problem is algorithm independent, and has been reproduced with the other algorithms. This problem seems to arise out of the following sequence of events:

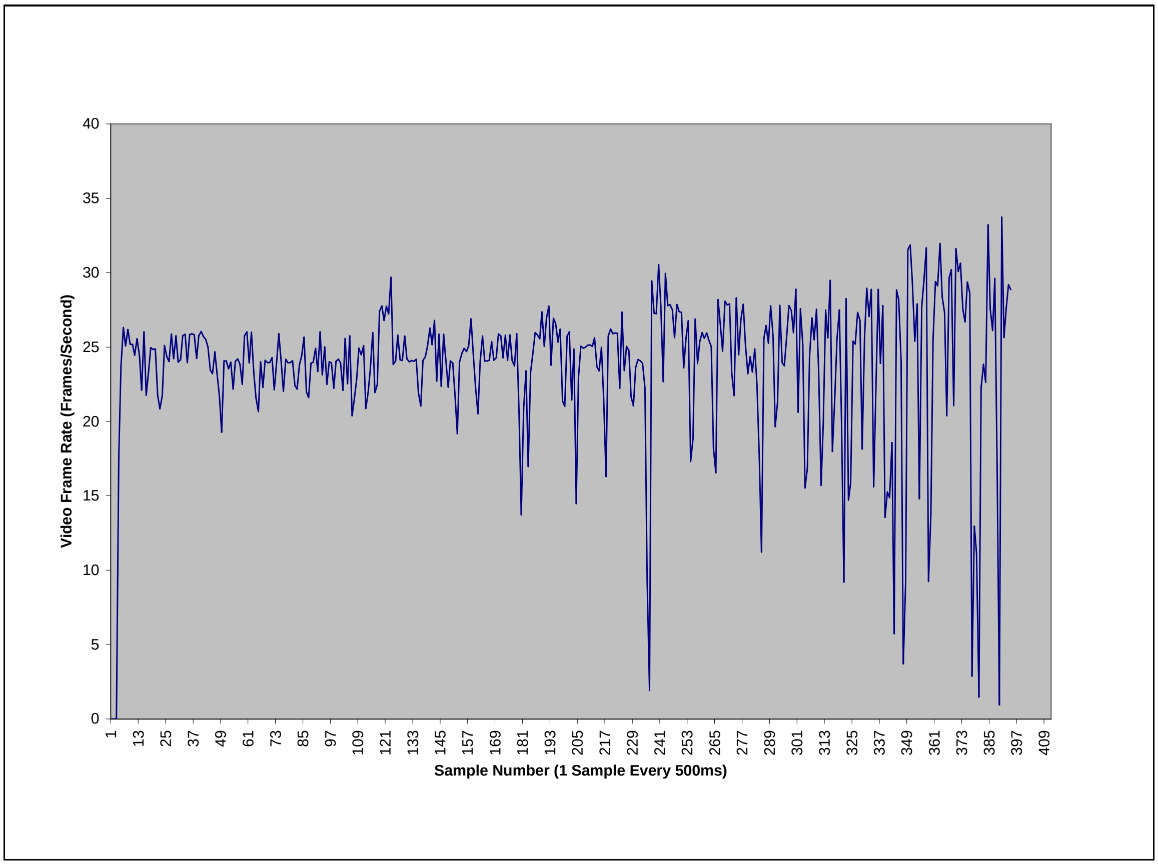
1. Because of load changes and disturbances on the system, we see a downward spike in quality delivered at the client end receiving data.

2. Since the client is unable to process and display the data, the excess data is buffered. Since this application uses TCP, the data is not lost, as was the case with the RTVC application.

3. Once the problem is corrected by the host manager, the client can proceed and resumes processing the data. The client, however, now has an excess of data readily at its disposal, and is consequently able to process the data very quickly. This leads to a sharp jump in delivered frame rate, pushing the frame rate above the upper quality threshold, possibly for several samples in a row.

4. Instrumentation in the client detects this situation, and reports this as a policy violation. The host manager consequently takes resources away from the application accordingly, in an attempt to bring quality back in line with requirements.

**Figure 7.30** Distributed Video-on-Demand Application Observed Oscillations




5. When the next disturbance to quality comes, delivered quality dips even further than before, because the client now has fewer resources to weather the disturbance. As a result, even more data is buffered than before.

6. Once the client executes again, there is an even larger amount of data at its ready disposal. Consequently, the client proceeds to drive the frame rate of the application above the upper quality threshold once again, possibly even higher than before.

7. This violation of quality policy is detected, and the host manager strips resources away from the application client once again.

8. This process repeats, and delivered quality destabilizes and oscillates.

There are several ways to help prevent or alleviate this scenario. These include the following:

- One approach is to remove upper bounds on quality expectation policies. Without an upper bound, the host manager will not take away resources on the high quality spike, thereby preventing or limiting future oscillations (preliminary testing shows this approach to be quite effective).
- Another approach would be to change the way quality measurements are taken. By smoothing or filtering collected data appropriately, spikes in measurements would be removed or reduced, which could prevent oscillations from occurring. While this approach would result in increased stability, we may lose responsiveness and increase convergence time as we no longer getting as sharp a picture of quality of service.
- A more complex approach would be to build logic into the host managers to detect oscillations in quality of service and take steps to stop them. This would require all algorithms to maintain more state information and to perform analyses. While this approach could be quite effective, it could also impose significant overhead.
- These problems could also be solved through management of network resources—including the end-system buffers used for transmitting and receiving data. It is becoming more apparent that every resource must be managed, and that managed network resources must include end-system buffering.

## 7.6 Multiple Distributed Applications

Having examined scenarios involving both multiple applications and distributed applications, it is now time to investigate the management of multiple distributed applications. This is, in effect, the ultimate instance of the quality of service management problem—multiple applications, multiple hosts, and multiple resources.

From previous experiments, we know that attempting to execute two RTVC transmitters from the same host will not work, as two video sources would be required. Even if a third host was used to act as a second transmitter, we know that a single distributed RTVC application will ultimately fail from lost frames under heavy loads. For that reason, as well, it would make little sense to attempt a distributed run of a one RTVC application and one video-on-demand application scenario. Consequently, we will proceed with multiple video-on-demand applications as our experiment.

### 7.6.1 Multiple Distributed Video-on-Demand Experiments Using Two Hosts

Our first set of experiments involved executing two instances of the video-on-demand application, with both servers residing on a single host, and both clients residing on another. The video-on-demand servers were executed on `farquarson`, while the clients to request and display the transmitted videos were executed on `strawberry`.

In all cases, both instances of the application were instructed to load, transmit, and display a copy of the same 160×120 8-bit video stream from files stored on different disks in the server's host. Also in all cases, the applications were given a quality of service expectation policy dictating that they must maintain a frame rate of 24 frames per second, plus or minus 2 frames (giving an acceptable range of 22 to 26 frames per second).

As with the previous set of distributed experiments, we need a host manager on each host involved in these experiments, as well as a domain manager. In this case, we executed the domain manager on the same host as the video-on-demand servers, `farquarson`.

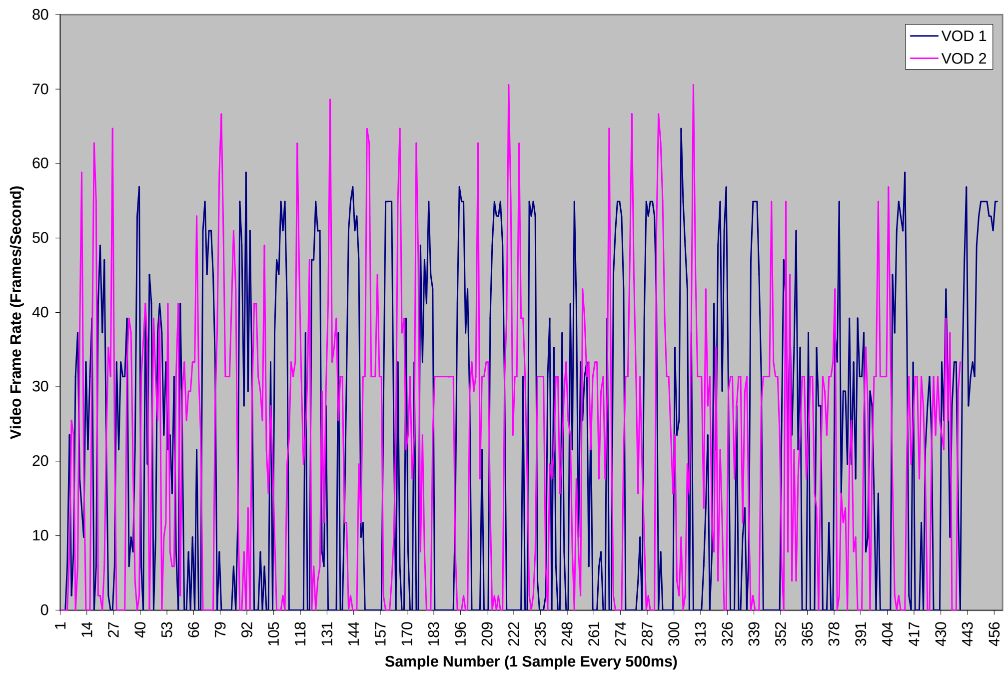
We began experimentation by once again collecting base case results. At first, we executed both applications in a completely unloaded environment, with plans to follow with loaded configurations. However, every repetition of this experiment failed to converge, but rather produced some interesting results.

Figure 7.31 presents a representative run from both instances of the video-on-demand application without management and without any applied load, with frame rate sampled every 500 milliseconds. From this chart, one can easily see the interleaving of execution of the two applications. This strongly suggests the contention for some resource on one of the hosts by either the two client or two server processes.

To uncover the source of this contention, we first sought to isolate the problem to either the client or server host. We repeated the above experiments, this time running clients on separate hosts (using `vanilla` as the host for the second client). There were no changes in observed performance. However, when we tried again with clients on the same original host, and the servers separated (using `vanilla` as described above), quality of service converged and stability was easily achieved. These results indicate that resource contention lies on the server side, and not on the client.

Observing the performance of the servers during experiments, we found that both servers consumed a low percentage of both CPU time and memory; there was a considerable amount of both resources sitting idle and ready for use. This suggests that neither of these resources were the problem. We found that our quality of service managers had no effect on quality of service either; giving the servers additional CPU cycles and memory made no difference

**Figure 7.31** Multiple Distributed Video-on-Demand Application Experiments with Two Hosts




whatsoever. Consequently, we must conclude that with both servers on the same host, there is a high contention for either disk or network I/O resources. A quick examination of I/O wait times suggests the contention is, in fact, for disk I/O.

These experiments again highlight the need for management of all computing resources. Unfortunately, since we do not currently have resource managers for these resources, there is little more we can do at this point. Consequently, we decided to pursue further experimentation using a third host to permit separation of the video-on-demand servers.

### 7.6.2 Multiple Distributed Video-on-Demand Experiments Using Three Hosts

Based on our experiences from experiments discussed in the previous section, we decided to carry out a new set of experiments involving two instances of the video on demand application. This time, the video-on-demand servers executed on separate hosts. One server was executed on `farquarson`, while the other was executed on `vanilla`. Both clients were executed on `strawberry`.

In all cases, both instances of the application were instructed to load, transmit, and display a copy of the same 160×120 8-bit video stream from files stored on each host. Also in all cases, the applications were given a quality of service expectation policy dictating that they must maintain a frame rate of 24 frames per second, plus or minus 2 frames (giving an acceptable range of 22 to 26 frames per second).

As with the previous set of distributed experiments, we need a host manager on each host involved in these experiments, as well as a domain manager. In this case, we executed the domain manager on `farquarson`, as we had done in previous experiments.

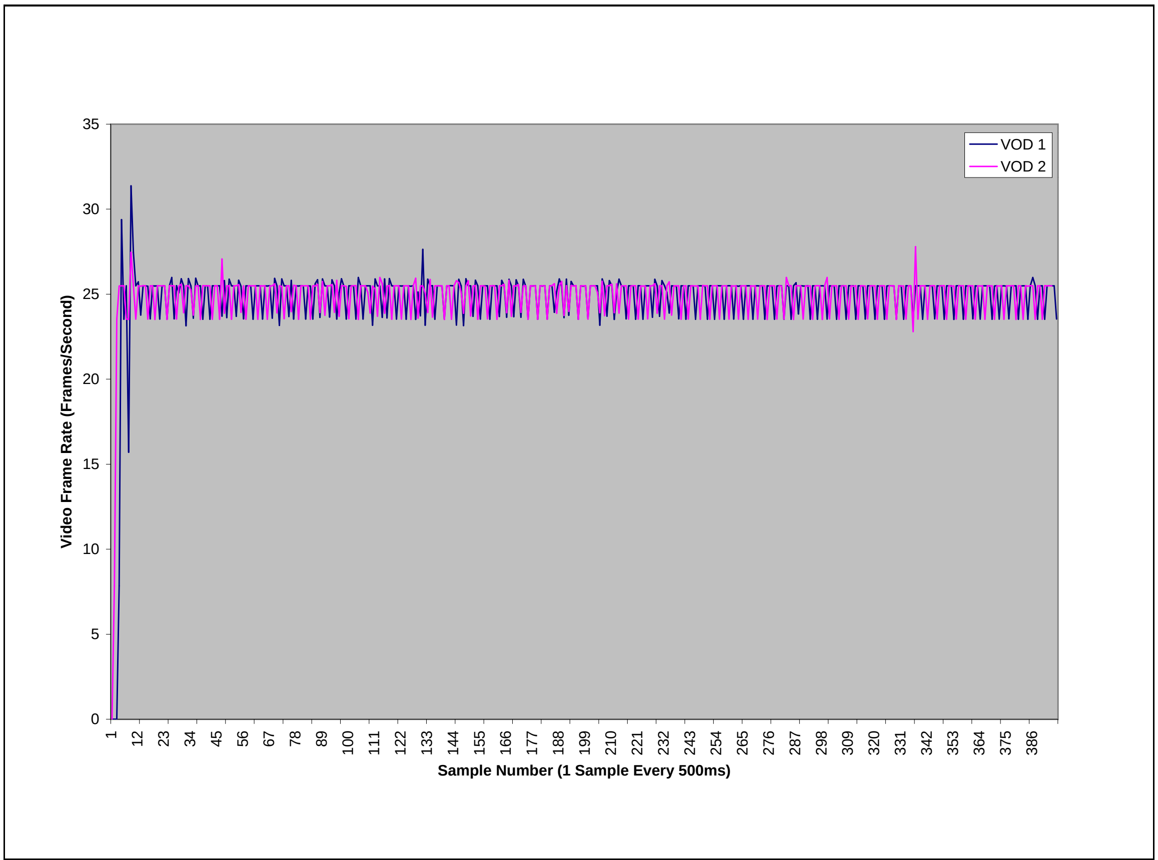
#### 7.6.2.1 Base Case Experiments

The first set of experiments executed was again a set of experiments to determine the behaviour of the two instances of the video-on-demand application under different conditions without the benefit of our quality of service management. These experiments could then be used both as a base line for comparison, to determine the improvements brought by our management, and for a simple regression analysis to determine the initial resource split for the multidimensional search algorithm as discussed in Section 5.3.

For these experiments, we again chose two levels of load to be applied to each of two resources, CPU cycles and memory, using our load generators. For the CPU, we chose loads of 0 (no load induced by our generator) and 8. For memory, we chose no load and the same very high paging load as before. By combining these loads, we have four possible load combinations: no load, CPU load only, memory load only, and both CPU and memory loads. For simplicity, we decided to apply the same load to all three hosts involved in the experiments at the same time. This reduced the number of experiment configurations from 64 down to 4, making analyses more manageable. Each of these experiments was again replicated.

Figure 7.32, Figure 7.33, Figure 7.34, and Figure 7.35 present representative runs from each instance of the the video-on-demand application in the no load, CPU load, memory

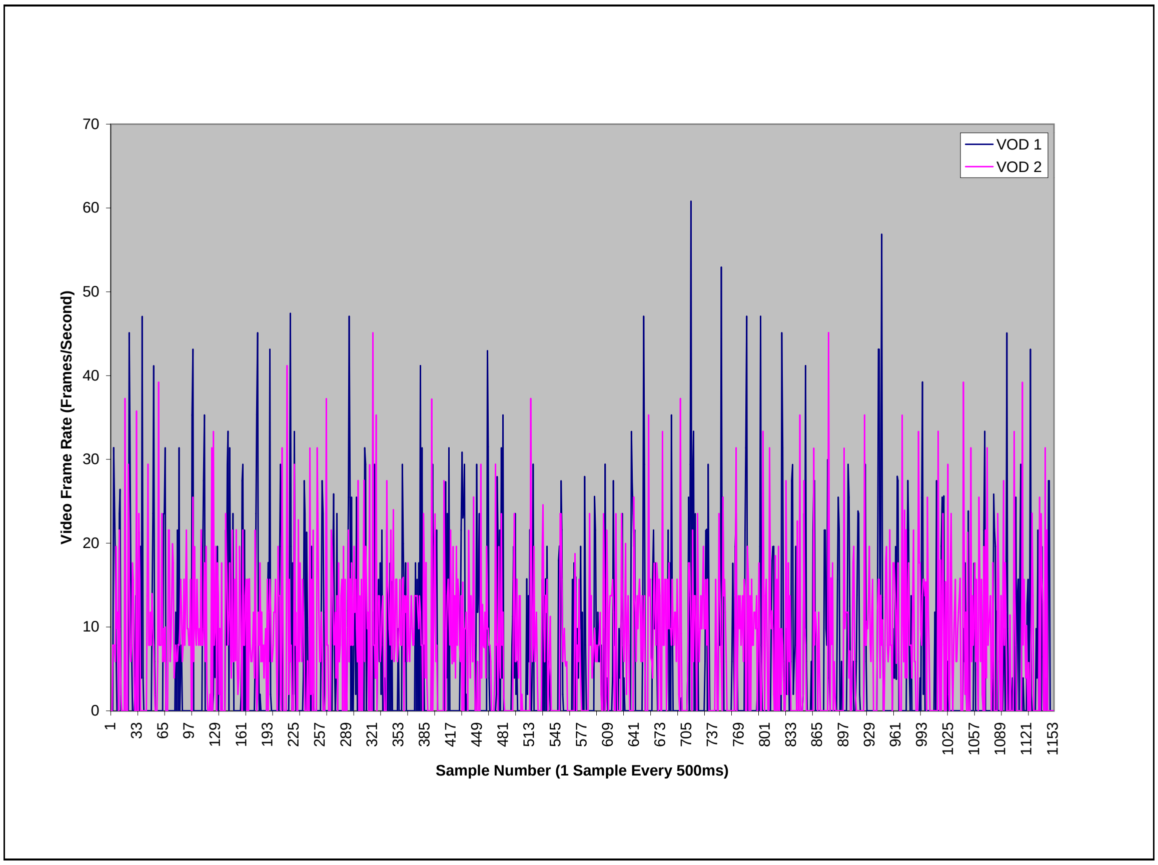
**Figure 7.32** Multiple Video-on-Demand Application Base Case Experiments with No Resources Loaded




load, and both load configurations, respectively. Key results are summarized in Table 7.19, including the convergence time, mean frame rate delivered (in frames per second), and the percentage of quality samples that violated quality of service requirements. When quality was able to converge and meet requirements, values reported apply to quality samples after convergence. Once again, frame rates were sampled every 500 milliseconds.

As can be seen, there was little difficulty in converging and delivering stable quality when no load was applied to any of the systems involved. When load was applied, quality dropped drastically, and could not converge to meet requirements. When applied individually, memory load caused more performance problems than CPU load; when applied together, quality of service delivery was at its worst, and neither instance of the video-on-demand application could make much progress.

**Figure 7.33** Multiple Video-on-Demand Application Base Case Experiments with CPU Loaded

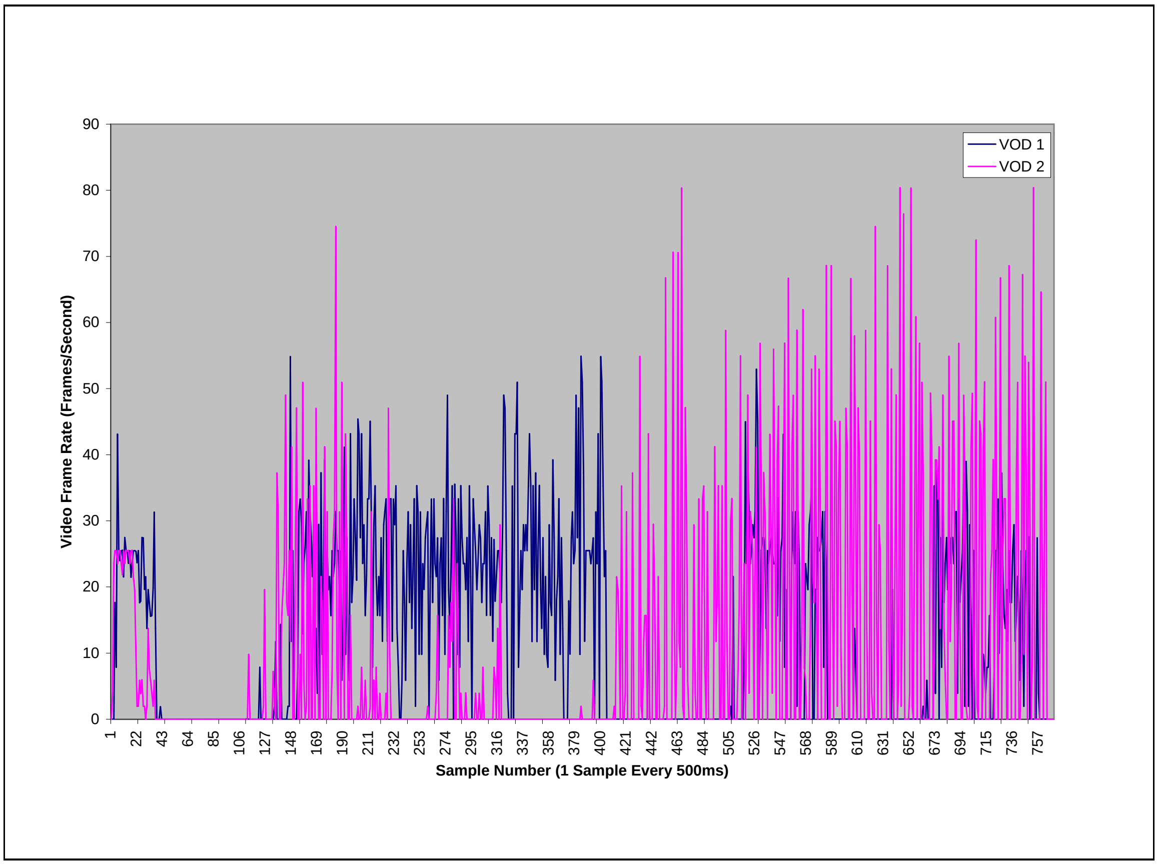



#### 7.6.2.2 No Resource Load Comparison Experiments

The base case experiments discussed in the previous section indicated that quality requirements could only be met when no load was applied to the application's environment. Since there were so few policy violations in this case, adding our quality of service managers to this configuration would produce the same results; our managers would simply sit idle, since there are no quality complaints to respond to.

Figure 7.36 presents a representative run from each instance of the the video-on-demand application when no load has been applied to each of the involved hosts under each resource management algorithm, with frame rate sampled every 500 milliseconds. As was expected, the quality of service management system sat idle throughout the entire experiment, as quality of service delivered met requirements except in only two or three samples in each

**Figure 7.34** Multiple Video-on-Demand Application Base Case Experiments with Memory Loaded



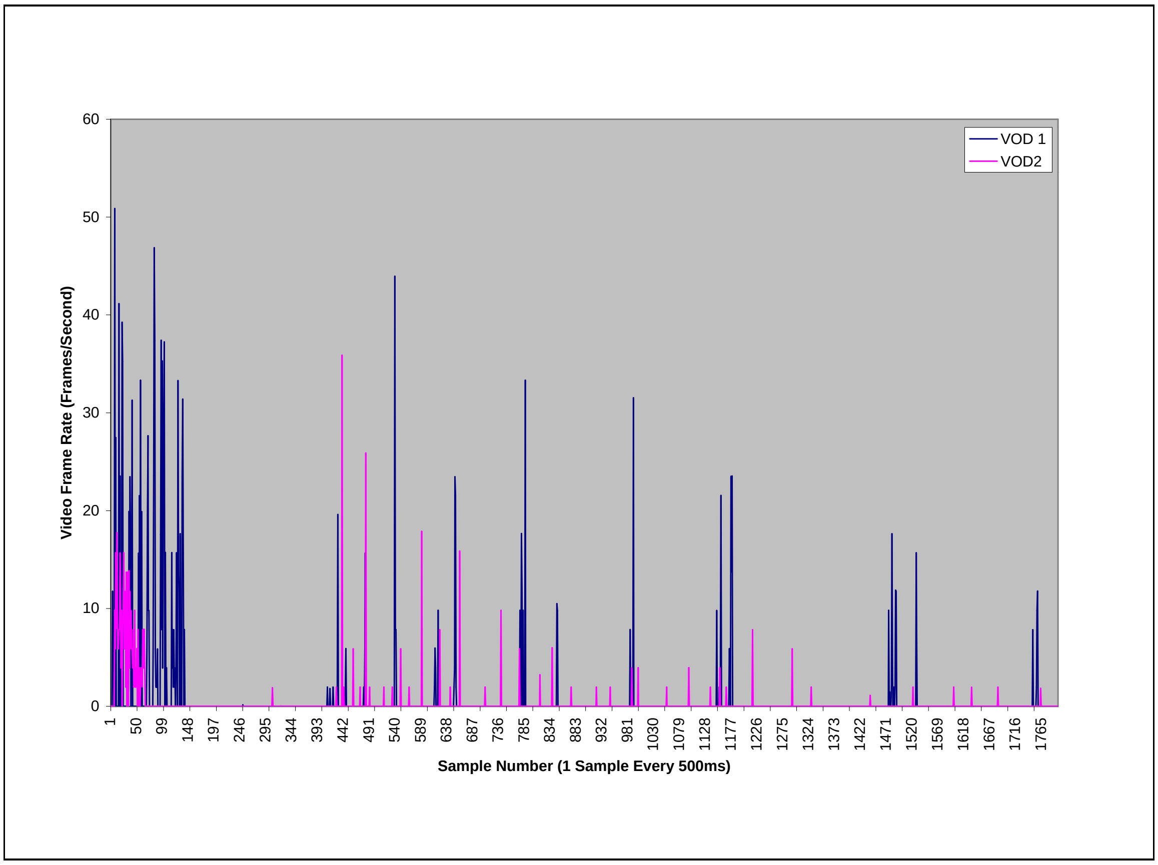

case. Consequently, we will move on to more interesting experiments without the need for further discussion.

#### 7.6.2.3 Single Resource Load Comparison Experiments

As shown earlier, in the presence of any load, neither instance of the video-on-demand application could meet quality objectives. In this section, we determine the impact our resource management algorithms have on quality of service delivery when a single resource is loaded. Once again, we have chosen the CPU as our resource, with its usual high load of 8. Experiments were again replicated.

Figure 7.37 presents a representative run from each instance of the the video-on-demand application when CPU load has been applied to each of the involved hosts under each

**Figure 7.35** Multiple Video-on-Demand Application Base Case Experiments with Both Resources Loaded




resource management algorithm, with frame rate sampled every 500 milliseconds. Key results are summarized in Table 7.20, including the convergence time, mean frame rate (in frames per second), percentage of quality samples violating quality of service requirements, standard deviation of quality samples, and CPU overhead. The mean frame rate, policy violations, and standard deviation are listed for all samples after convergence.

Each algorithm was able to provide converged and stable quality delivery for both instances of the algorithms. There was, however, a great deal of difference between the performance of the algorithms, with the metric-heuristic approach faring significantly better than the other two algorithms, particularly convergence time. The round robin approach trailed the metric-heuristic approach in performance, but still delivered better quality than the multidimensional search algorithm in most respects. The multidimensional search approach fared the worst of the three algorithms, barely managing to converge and deliver stable

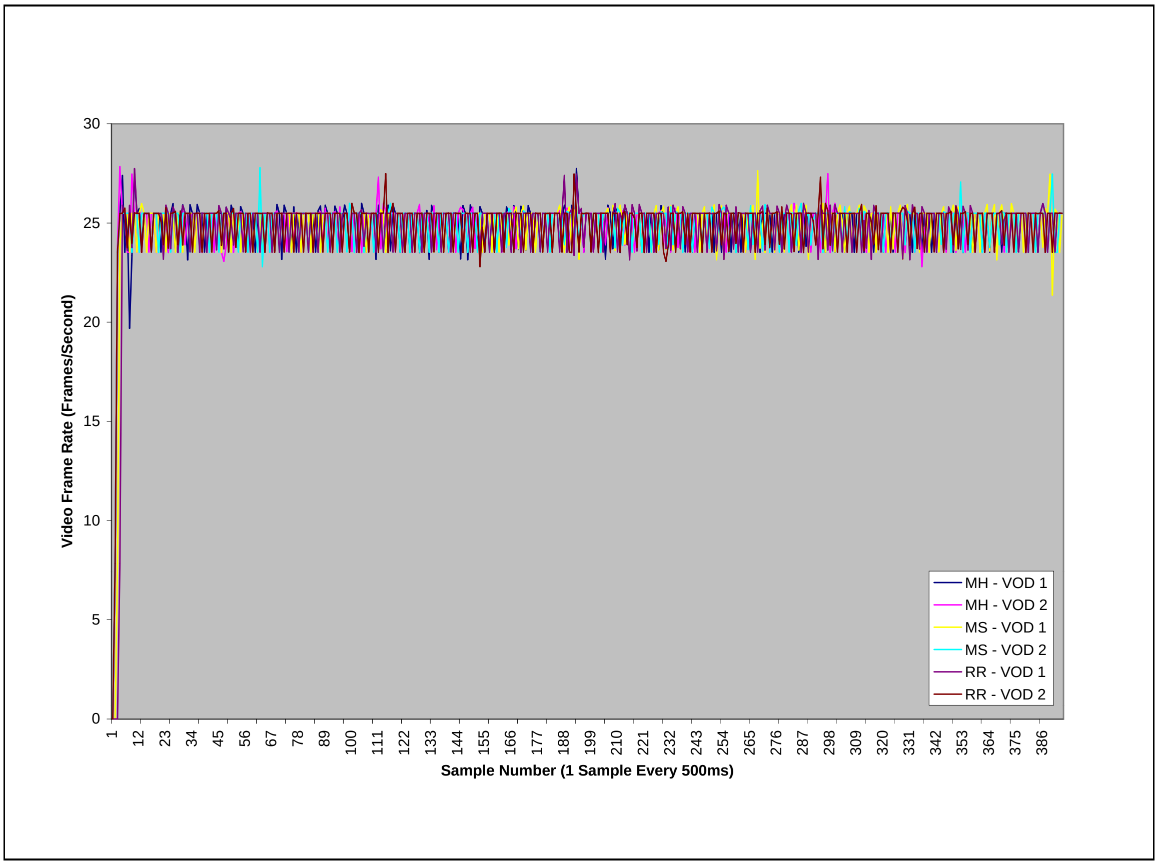
**Table 7.19** Summary of Multiple Video-on-Demand Application Base Case Experiments

| Load Applied | Application | Convergence Time (%) | Mean Frame Rate (FPS) | Policy Violations (%) |
|---|---|---|---|---|
| None | MPEG 1 | 2.54% | 25.00 | 0.25% |
| None | MPEG 2 | 2.78% | 25.00 | 0.26% |
| CPU only | MPEG 1 | N/A | 8.36 | 96.45% |
| CPU only | MPEG 2 | N/A | 7.63 | 97.22% |
| Memory only | MPEG 1 | N/A | 4.56 | 96.64% |
| Memory only | MPEG 2 | N/A | 4.52 | 96.94% |
| Both CPU and memory | MPEG 1 | N/A | 0.65 | 99.61% |
| Both CPU and memory | MPEG 2 | N/A | 0.32 | 99.94% |

**Table 7.20** Summary of Multiple Distributed Video-on-Demand Application Experiments with CPU Loaded

| Algorithm | Application | Convergence Time (%) | Mean Frame Rate (FPS) | Policy Violations (%) | Standard Deviation (FPS) | CPU Overhead (%) |
|---|---|---|---|---|---|---|
| Metric-Heuristic | MPEG 1 | 14.36% | 23.88 | 21.67% | 1.47 | 1.24% |
| Metric-Heuristic | MPEG 2 | 14.97% | 23.83 | 30.09% | 1.73 | 1.24% |
| Multidimensional Search | MPEG 1 | 91.78% | 23.24 | 25.00% | 2.04 | 6.07% |
| Multidimensional Search | MPEG 2 | 92.72% | 22.88 | 35.14% | 2.42 | 6.07% |
| Round Robin | MPEG 1 | 70.75% | 23.68 | 37.60% | 2.53 | 3.64% |
| Round Robin | MPEG 2 | 71.67% | 23.64 | 52.89% | 3.25 | 3.64% |

**Figure 7.36** Multiple Video-on-Demand Application Experiments with No Resources Loaded



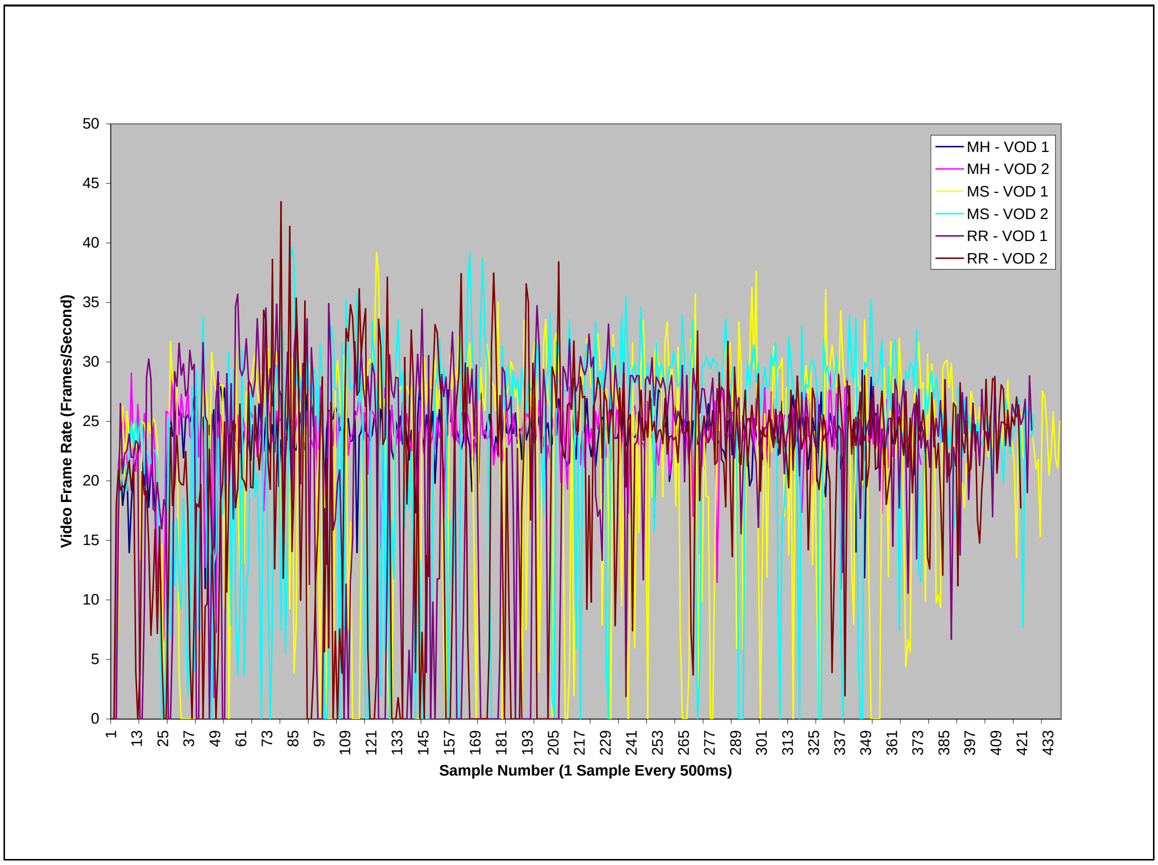

quality, and in some cases not even able to do that.

#### 7.6.2.4 Multiple Resource Load Comparison Experiments

In the previous section, we saw the improvements management brought to multiple video-on-demand applications when under a single resource load. In this section, we investigate the effectiveness of managing these applications when under multiple resource loads—both the CPU and memory. Once again, all experiments are replicated.

Figure 7.38 presents a representative run from both instances of the video-on-demand application managed by each resource management algorithm with load applied to both the CPU and memory, and with frame rate sampled every 500 milliseconds. Key results are summarized in Table 7.21, including the convergence time, mean frame rate (in frames per

**Figure 7.37** Multiple Video-on-Demand Application Experiments with CPU Loaded

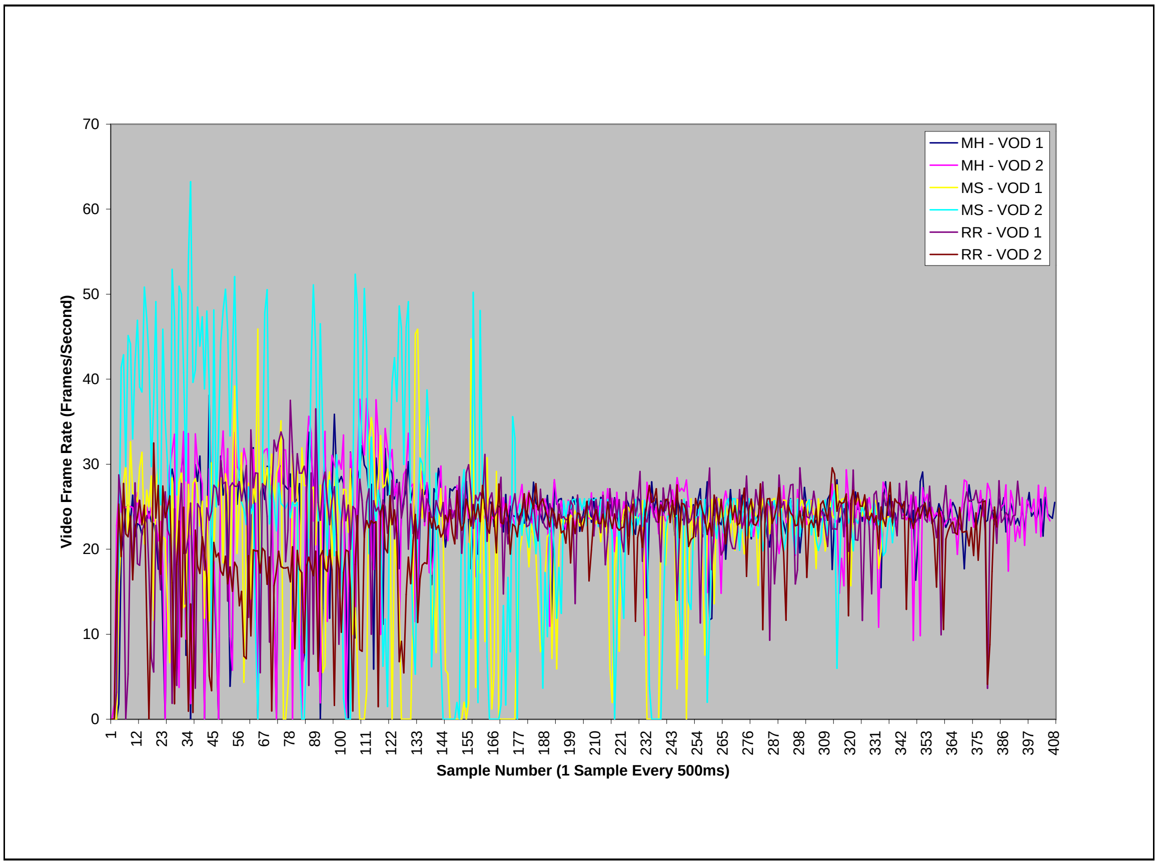



second), percentage of quality samples violating quality of service requirements, standard deviation of quality samples, and CPU overhead. The mean frame rate, policy violations, and standard deviation are listed for all samples after convergence.

Once again, all three algorithms were able to converge and deliver stable quality. This time, however, the differences between the various algorithms were not as large. The metric-heuristic approach again posted top numbers. The round robin approach improved substantially over results with CPU load alone, but still fell behind the metric-heuristic approach. The multidimensional search approach again fared worse than the other two algorithms, but also improved its quality of service substantially compared to when the CPU was loaded alone.

As was the case with only a single instance of the video-on-demand application, it is interesting to observe that quality improved dramatically for both the round robin and multidi-

**Figure 7.38** Multiple Video-on-Demand Application Experiments with Both Resources Loaded




mensional search algorithms when resource loads were applied to both resources, as opposed to only single load. Once again, this is likely because each iteration of the algorithms, including those adjusting memory resources, are now making progress towards quality goals. When load was only applied to the CPU, both of these algorithms still made adjustments to the memory allocations, and took longer to make the required adjustments to improve quality. The metric-heuristic approach, however, was able to make better resource management decisions based on the metric information collected.

**Table 7.21** Summary of Multiple Distributed Video-on-Demand Application Experiments with Both Resources Loaded

| Algorithm | Application | Convergence Time (%) | Mean Frame Rate (FPS) | Policy Violations (%) | Standard Deviation (FPS) | CPU Overhead (%) |
|---|---|---|---|---|---|---|
| Metric-Heuristic | MPEG 1 | 39.16% | 24.02 | 19.68% | 1.67 | 1.21% |
| Metric-Heuristic | MPEG 2 | 39.22% | 23.94 | 26.61% | 2.07 | 1.21% |
| Multidimensional Search | MPEG 1 | 57.52% | 22.61 | 31.03% | 4.42 | 4.54% |
| Multidimensional Search | MPEG 2 | 57.82% | 22.57 | 33.33% | 4.53 | 4.54% |
| Round Robin | MPEG 1 | 47.58% | 23.54 | 27.47% | 2.34 | 2.32% |
| Round Robin | MPEG 2 | 49.43% | 23.32 | 36.23% | 2.48 | 2.32% |

## 7.7 Enforcement Policy Experiments

Experiments in previous sections of this chapter focussed on evaluating and comparing the performance and capabilities of the various resource management algorithms under a variety of application configurations and operating conditions. In this section, we turn our focus to experiments that investigate the effects of enforcement policies on the delivery of quality of service. These policies are geared at providing more flexibility and control over the resource management process, particularly in overload conditions when there are insufficient resource supplies to keep up with resource demands.

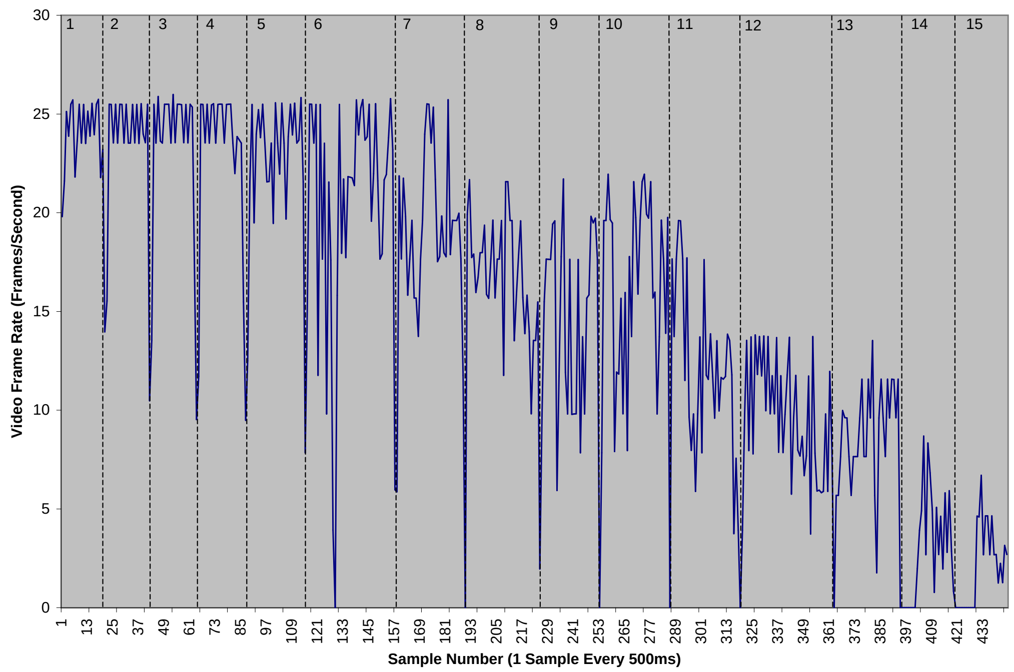
A preliminary analysis of previous experiments indicates that the leading resource management algorithms are the metric-heuristic and the round robin approaches, with the multidimensional search approach trailing in almost all experiments in most respects. Consequently, these policy experiments only deal with the metric-heuristic and round robin algorithms for further study.

### 7.7.1 Admission Control Experiments

As mentioned in Section 6.5.1, admission control policies were developed to prevent saturation of the computing environment. By restricting the number of applications with quality of service requirements, more resources are available to support the expectation policies of those applications that are still admitted. In this section, we examine experiments that make use of these policies.

For these experiments, the host `farquarson` was used. Every 10 seconds, a new instance of the MPEG player application was launched on this host, and instructed to play the same 80×60 8-bit video from a file at a mean frame rate of 25 frames per second, plus or minus 2 frames (yielding an acceptable quality range of 23 to 27 frames per second). No loads were

**Figure 7.39** Admission Control Experiment Results with No Management




applied to the computing environment during these experiments.

#### 7.7.1.1 Without Management

Figure 7.39 shows a representative run from the first instance of the MPEG player started during experimentation. As noted in the figure, additional instances of the application were started roughly 10 seconds apart (sometimes more, depending on the load on the system induced by other instances of the application). As can be seen from the figure, quality of service is quite good until the fifth instance of the player is added. Variation in quality deliver begins to increase at this point with each subsequent player added, and mean quality delivered drops.

Before the fifth video player was added, quality was delivered at a mean frame rate of

**Table 7.22** Summary of Admission Control Experiments with Management

| Algorithm | Admitted Instances | Mean Frame Rate (FPS) | Standard Deviation (FPS) |
|---|---|---|---|
| Metric-Heuristic | 4 | 24.55 | 0.78 |
| Round Robin | 5 | 21.97 | 3.08 |

24.52 frames per second, and a standard deviation of 0.79. When five video players were executing, mean quality dipped to 22.98 frames per second (now below requirements on average), while standard deviation increased to 2.00. After the fifth player, mean quality drops sharply, while deviation in results increases dramatically. After the eighth MPEG player is added to the system, quality requirements are missed 100% of the time for all players.

As an interesting note, a spike in quality was observed when each instance of the application was created. After a brief investigation, it was discovered that this disturbance is caused by the window manager temporarily blocking X Windows traffic during the creation and placement of the each video window. This temporary blockage causes a downward spike in quality of service, as video frames cannot be displayed, which results in the application waiting briefly until it can proceed.

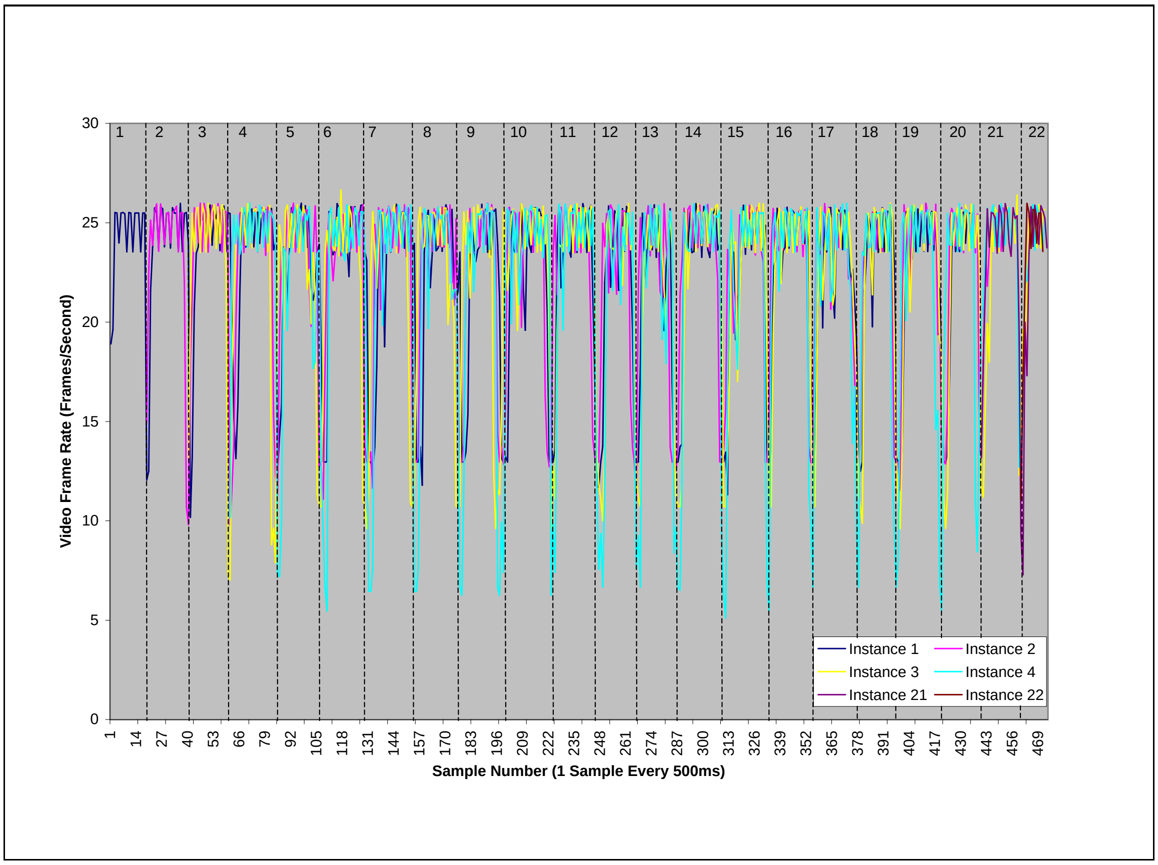
#### 7.7.1.2 With Management

From experimentation discussed in the previous section, it would appear that four MPEG players is the maximum number that should be allowed while maintaining quality. Five players results in nearly acceptable quality, but may be too much for the system to bear. In this section, we examine admission control experiments when our resource management algorithms (metric-heuristic and round robin) are in place. This time, an admission control policy is also added requiring that at least 5% of the CPU and 100 pages of free physical memory are available before admitting an instance of the MPEG player application.

Figure 7.40 and Figure 7.41 present representative runs from experiments with metric-heuristic and round robin management in place, respectively. In both cases, our admission control policy was also in effect. The metric-heuristic algorithm admits four instances of the MPEG player application, and proceeds to deliver quality within requirements. The round robin algorithm, on the other hand, admits five instances of the MPEG player application, and quality of service is substandard. When earlier instances of the MPEG player application complete, resources are freed, and new instances are admitted to the system.

The difference in the number of admissions is likely due to two factors. First, the metric-heuristic algorithm tends to converge faster than the round robin approach, as observed

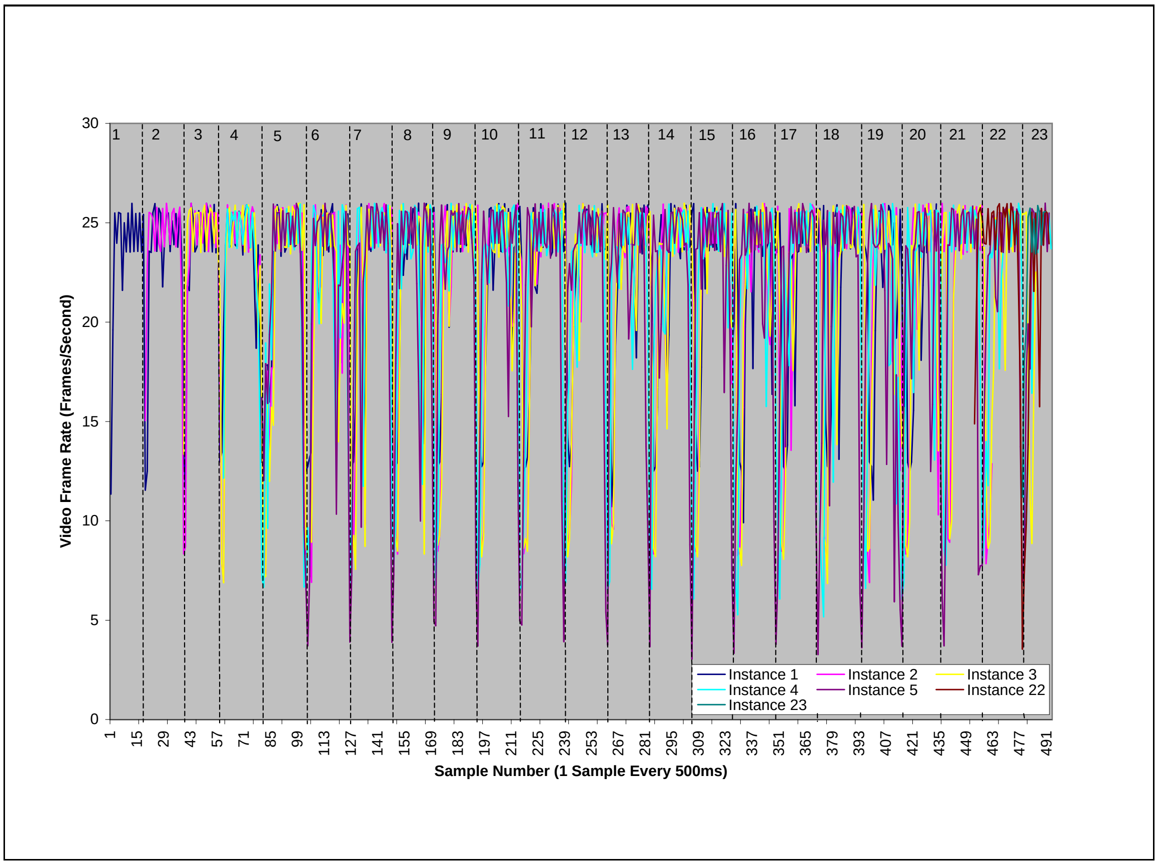
**Figure 7.40** Admission Control Experiment Results with Metric-Heuristic Management




in earlier experiments. Consequently, it may allocate all resources to support other applications, before the fifth application arrives, and denies it as a result. The round robin algorithm, in working slower, has resources to spare when the fifth instance arrives, and admits it as a result, only to run out later on. The second factor is the lower overhead of the round robin algorithm—because it consumes fewer resources itself, more are available to other applications, giving more room to admit the fifth instance than the metric-heuristic approach would have available.

It would appear that by the nature of the algorithms, the round robin approach may be more generous than the metric-heuristic approach in admitting applications to the system. At times this may be favourable; at other times, as was the case above, this may lead to slight system saturation.

**Figure 7.41** Admission Control Experiment Results with Round Robin Management

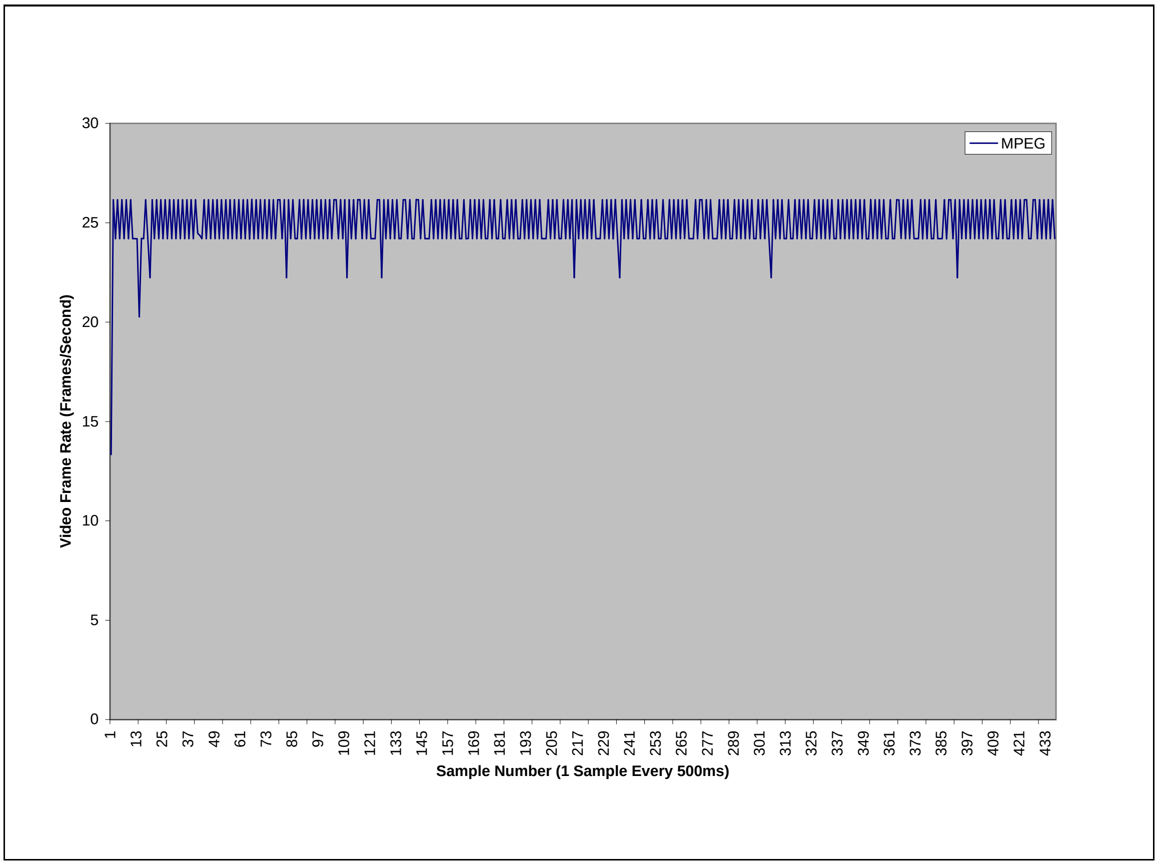



### 7.7.2 User Hint Experiments

As was discussed in earlier sections of this work, user hints present a mechanism that allows computing resources to be routed to applications that are the focus of user activity and interest, and away from those that are essentially ignored. Using the sensors discussed in Section 6.1.2.3, and the policies discussed in Section 6.5.2, we carried out experiments investigating the effectiveness of user hints. (Additional experimental results with user hints can be found in [90].)

We began by adding window system user hint sensors to our test applications discussed in Section 7.1.4.8 to detect user activity with the applications. We then defined policies for the applications to detect and report all changes in their visibility (such as minimizing and restoring, covering and uncovering, and so on).

**Figure 7.42** Single MPEG Player Experiment Results

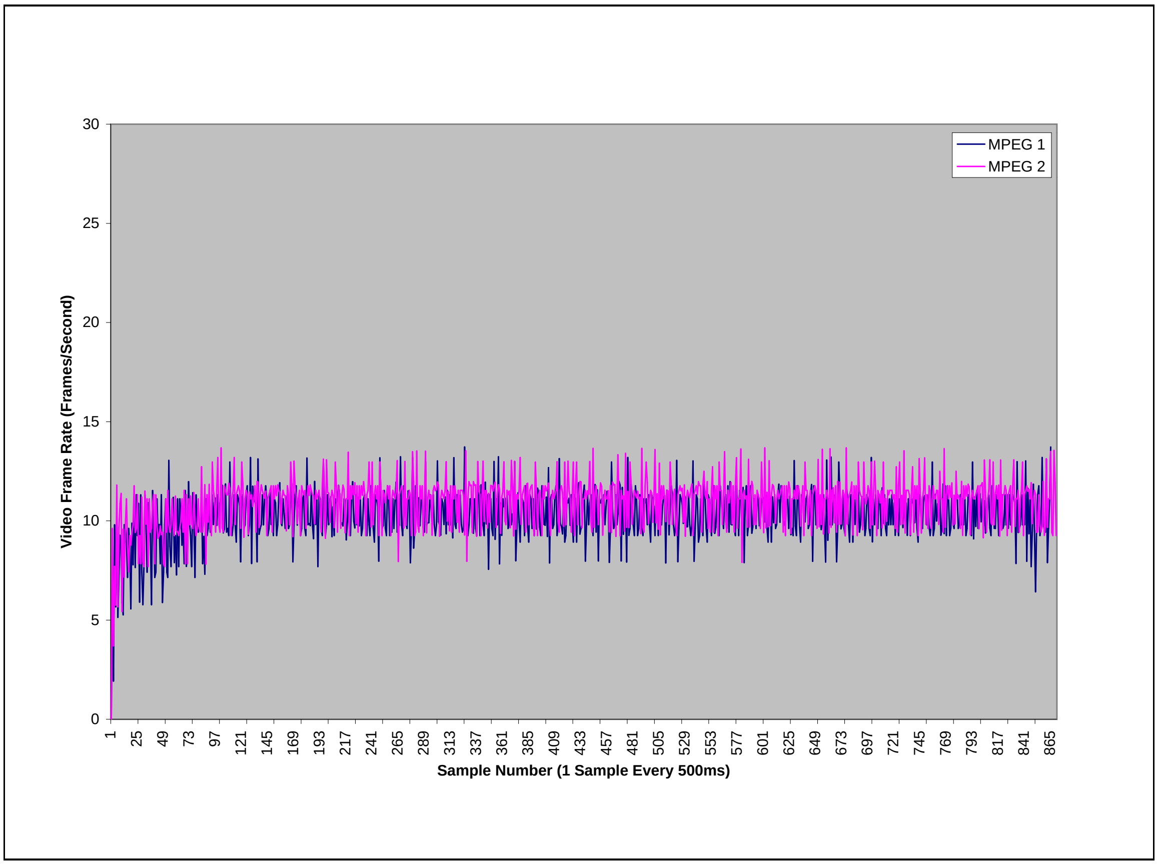



#### 7.7.2.1 Without User Hint Policies

For experimentation, we decided to use two instances of the stand-alone MPEG player application. As can be seen from Figure 7.42, giving a single instance of the MPEG player a 320×240 8-bit video from a file resulted in acceptable quality, averaging 25.01 frames per second, with little variation on `farquarson`. However, we noted that the application consumed roughly 85% of available CPU cycles to attain and maintain this level of quality. Consequently, one would predict that two instances of the application would fail.

Figure 7.43 and Figure 7.44 show representative runs from executing two instances of the MPEG player displaying the same video from different files on `farquarson`. In both cases, both instances of the application were instructed to display its video at a mean frame rate of 25 frames per second, plus or minus 2 frames (giving an acceptable quality range of 23 to 27 frames per second). In Figure 7.43, the applications were managed by our quality

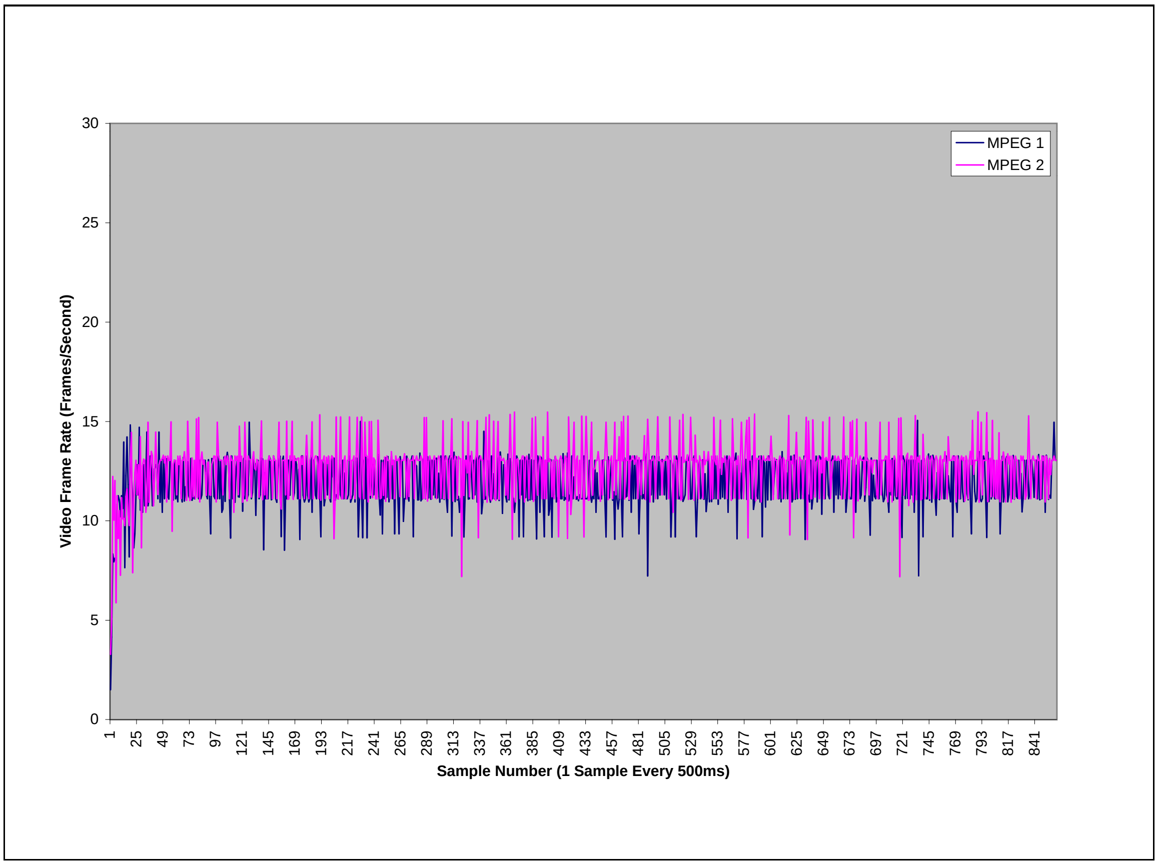
**Figure 7.43** Multiple MPEG Player Experiment Results with Metric-Heuristic Management




of service managers using the metric-heuristic algorithm; in Figure 7.44, the round robin algorithm was used instead. In both cases, a user hints policy was not deployed.

In both cases, neither application could converge and produce acceptable results; in fact, 100% of quality samples were below quality of service requirements. quality of service was limited to an average of 10.61 frames per second for the metric-heuristic approach, and 12.29 frames per second for the round robin approach. In both cases, higher variation of delivered quality was present when both applications were executed, as opposed to the single application case. The results were the same regardless of whether both applications were completely visible, both were completely obscured, or one was obscured and the other visible. Whether applications are actually usable by users or not had no effect on their performance or resource consumption.

**Figure 7.44** Multiple MPEG Player Experiment Results with Round Robin Management

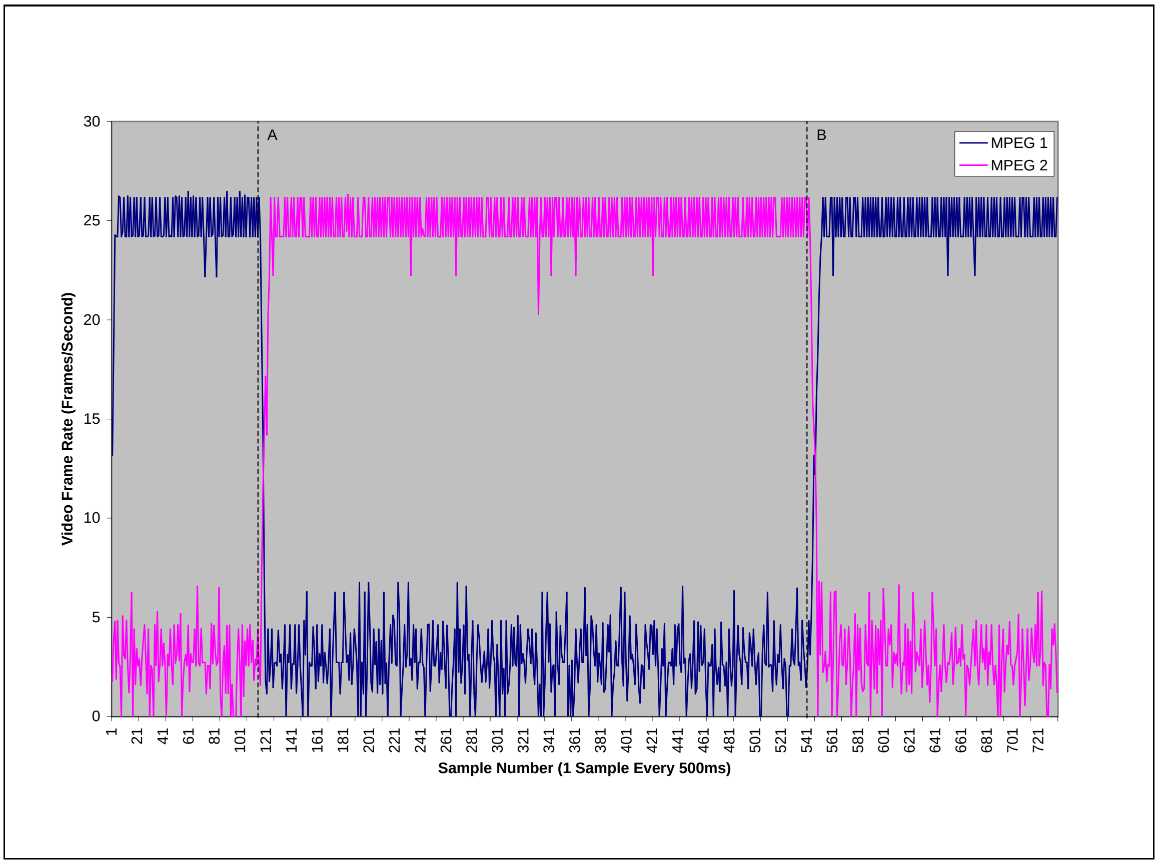



#### 7.7.2.2 With User Hint Policies

Figure 7.45 and Figure 7.46 show representative runs repeating the above scenarios, using the metric-heuristic and round robin algorithms respectively. This time, however, user hints were detected and considered in resource management decisions. In both cases, the two instances of the MPEG player were started at the same time, with the first instance's video window completely obscuring the second's. The user hint sensor in the covered application detected this, and reported that it was not visible to the user. Consequently, the quality of service expectation policies for the second instance were suspended, and it was not allocated resources, allowing the first instance to meet quality requirements.

Part way through execution, at time A in Figure 7.45 and Figure 7.46, the ordering of the windows were reversed. The first instance of the MPEG player was placed behind the second, making the first completely obscure and the second completely visible. At

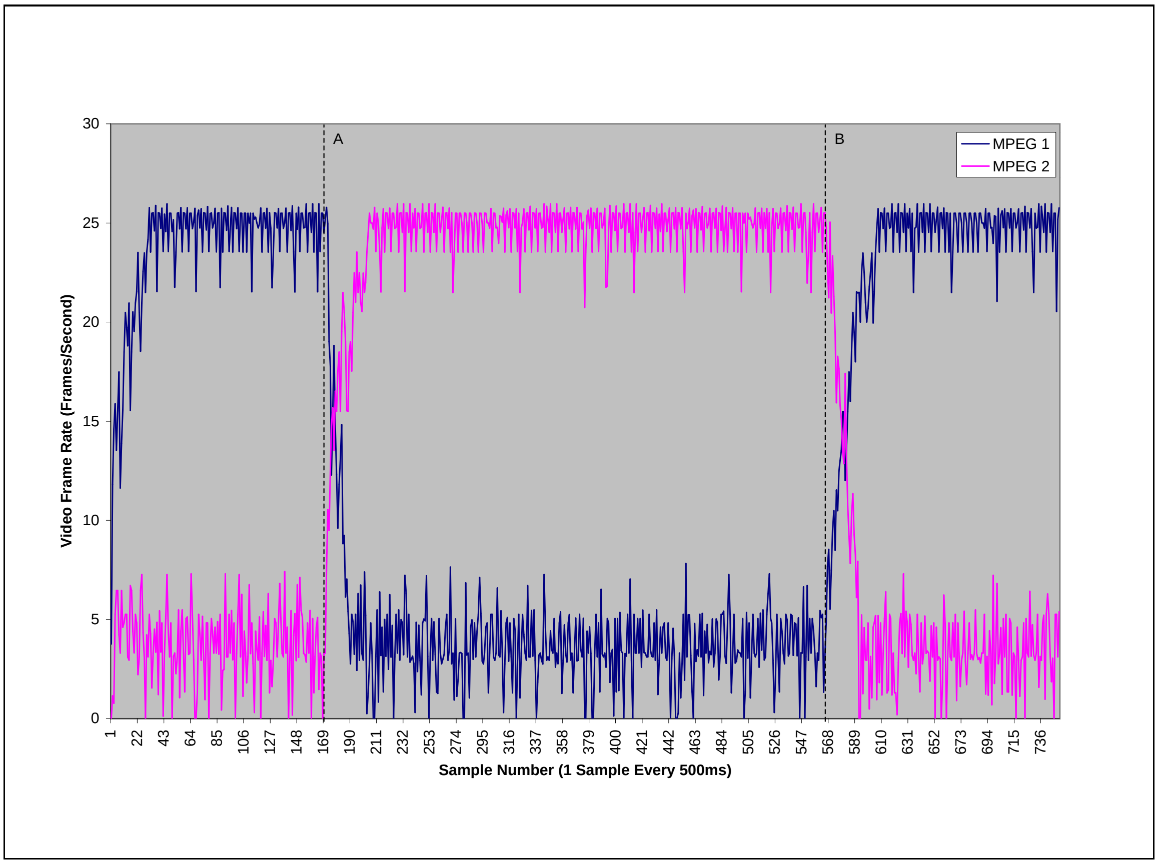
**Figure 7.45** Multiple MPEG Player Experiment Results with User Hints Handled with Metric-Heuristic Management




this point, the user hint sensors in each instance detect what has happened. The first instance, realizing it can no longer be seen, notifies the local host manager, which suspends its quality of service policies, and frees its resources for allocation elsewhere. The second instance, knowing that it is now completely visible, notifies the lost host manager, which reinstates its quality of service policies. The video frame rate sensor in the second instance detects that it is below its expected frame rate, and complains until it is given enough resources to meet its requirements. In a short amount of time, the quality output of the two applications have essentially reversed, with the second instance meeting requirements, and the first receiving resources leftover from the second.

Further along the execution, at time B in Figure 7.45 and Figure 7.46, the ordering of the windows were reversed once again. This time, the second instance of the MPEG player

**Figure 7.46** Multiple MPEG Player Experiment Results with User Hints Handled with Round Robin Management




was sent behind the first, once again making the second completely obscure and the first completely visible. The user hints sensors in the two instances again detect the situation and make the appropriate reports to the local host manager. The quality of service policy for the first instance is reinstated, and it quickly regains the resources it once held to meet quality requirements. The policy for the second is once again suspended, and its quality returns to previous levels.

Table 7.23 summarizes key results from experiments with user hints detected and considered in the management process, including convergence time, mean frame rate, policy violations, and standard deviation measured in sampled quality. Mean frame rate, policy violations, and standard deviation are all measured during period in which quality had stabilized after a change in application visibility. The metric-heuristic approach out-performed the round

**Table 7.23** Summary of Multiple MPEG Player Experiments With User Hints Detected

| Algorithm | Application State | Convergence Time (%) | Mean Frame Rate (FPS) | Policy Violations (%) | Standard Deviation (FPS) |
|---|---|---|---|---|---|
| Metric-Heuristic | Visible | 1.31% | 25.02 | 1.67% | 0.74 |
| Metric-Heuristic | Covered | N/A | 2.88 | 100.00% | 1.10 |
| Round Robin | Visible | 5.56% | 24.81 | 4.34% | 0.75 |
| Round Robin | Covered | N/A | 3.61 | 100.00% | 1.17 |

robin approach in nearly all aspects. The only case in which the round robin approach fared better, was in the mean frame rate delivered by the covered application. It is likely that the smaller overhead imposed by the round robin algorithm allowed for more resources to be available to the otherwise suffering application, as was observed in earlier experiments.

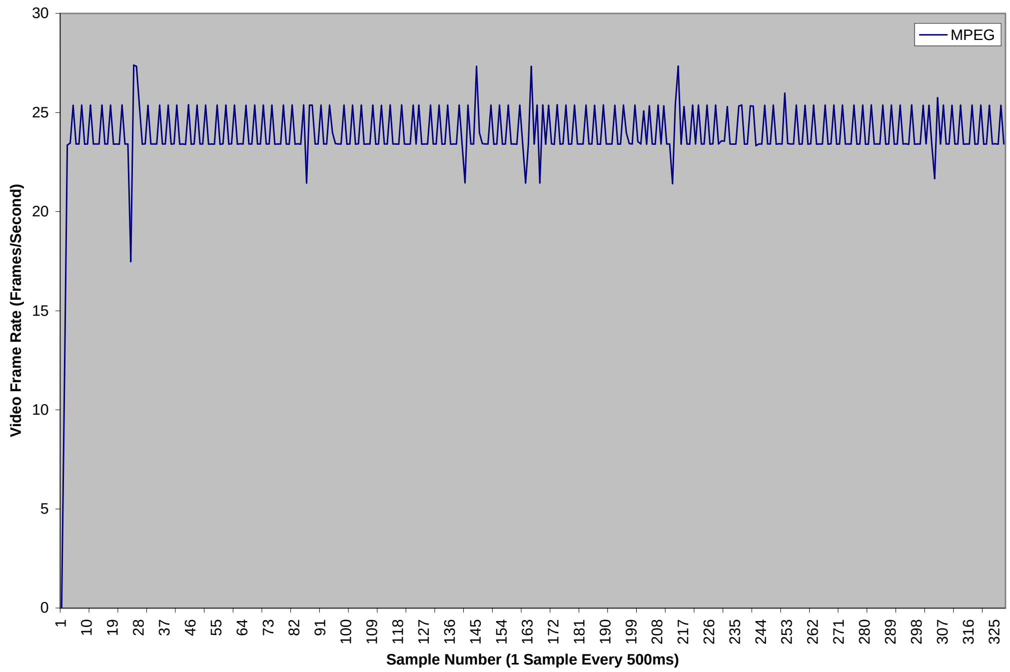
As can be seen from these results, our user hint mechanisms allow us to route resources to applications which truly need them. Doing so means that quality of service can be delivered where it is most important. This allows more applications in need of quality support to be handled at once, assuming they are not all in active use at the same time. User hints also reduce the overhead of managing multiple applications, since applications that are not the focus of activity are removed from the workload of the management system. Consequently, it would appear that this approach can be quite useful to quality of service management.

### 7.7.3 Service Differentiation Experiments

As mentioned earlier in this work, service differentiation provides a general mechanism to provide preferential service to applications based on a variety of criteria, including the user of the application, the type of application, the time the application was started, and so on. In doing so, we can use enforcement policies to determine how to handle overload situations when we cannot support the quality of service requirements of all applications at the same time. In this section, we examine the use of service differentiation policies discussed in Section 6.5.3, and present results from experimentation. (Additional experimentation involving service differentiation can be found in [133].)

For experimentation in this section, we chose to use the video-on-demand application. The application workload assigned was a 10 minute 320×240 8-bit video of footage recorded by our I/O counter cameras stored in a local file. The expectation policy in place dictated that this workload be delivered at a mean frame rate of 24 frames per second, plus or minus 2 frames per second (yielding an acceptable quality range from 22 to 26 frames per second). No resources were loaded during these experiments.

**Figure 7.47** Single Video-on-Demand Application Experiments

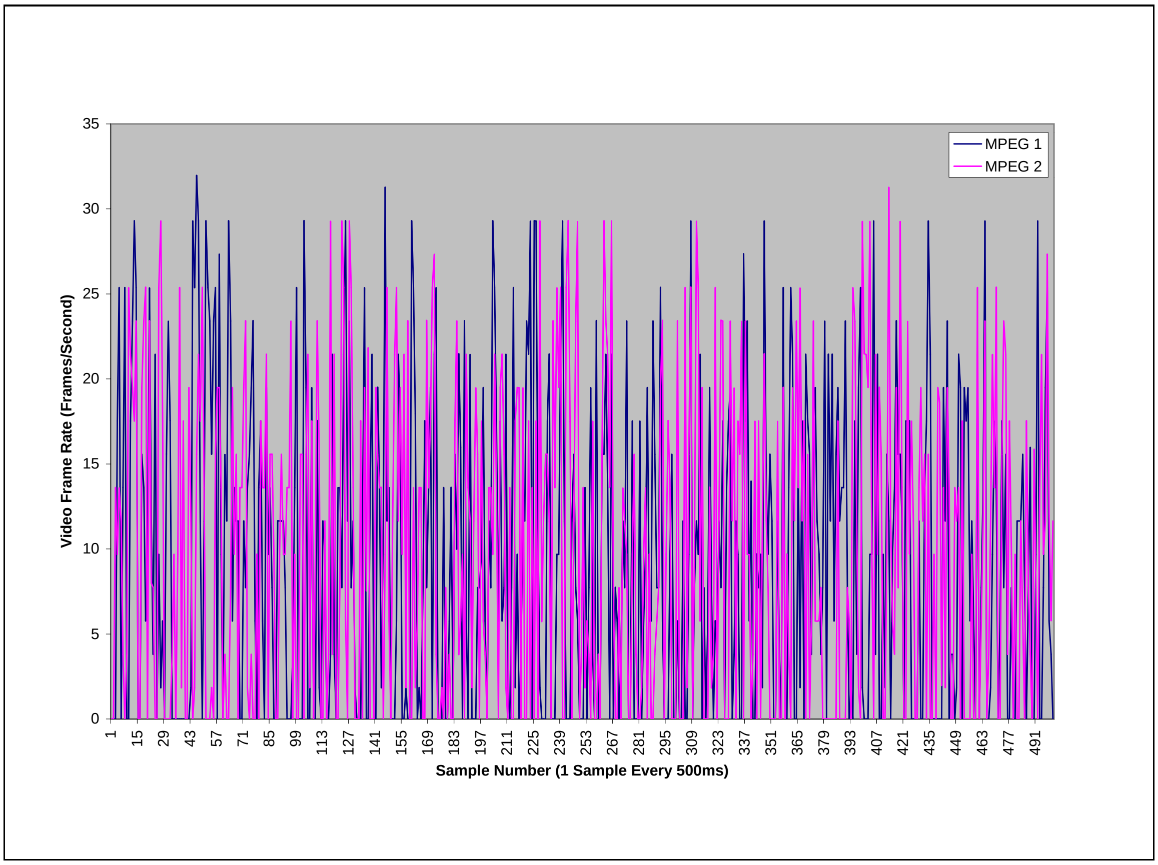



#### 7.7.3.1 Without Management

Before experimenting with service differentiation policies, we first carried out initial experiments to gauge the performance of the application without management in place. This includes executing a single instance of the application, as well as multiple instances.

Figure 7.47 presents a representative run from a single instance of the video-on-demand application, with the server executing on `farquarson`, and the client receiving and displaying video executing on `strawberry`. The application had no difficulty in delivering acceptable quality, converging at 1.23% of its run and missing requirements only 3.68% of the time after that. A mean frame rate of 23.91 frames per second was delivered on average, with a standard deviation of 1.31. It was noted during experimentation, however, that a single instance of the application consumed roughly 83% of CPU cycles on the client end; consequently, multiple instances of the application could present difficulties in meeting quality

**Figure 7.48** Multiple Video-on-Demand Application Experiments




objectives.

Figure 7.48 presents a representative run from multiple instances of the video-on-demand application. In this case, servers were executed on `farquarson` and `vanilla`, while both clients were executed on `strawberry`. (A second server host was used to avoid disk contention problems observed in experiments discussed in Section 7.6.1.) As was expected, quality of service was quite poor and highly variable. The mean frame delivered was only 9.68 frames per second, with a standard deviation of 6.51. Quality of service requirements were violated 91.9% of the time.

In short, the computing environment could simply not support multiple instances of the video-on-demand application with the given video workload. (Previous experiments in Section 7.6.2 revealed that multiple video-on-demand applications could be supported with a less resource intensive workload.) To resolve this overload condition, we will apply a service differentiation policy.

#### 7.7.3.2 With Management

Using policies discussed in Section 6.5.3, a service differentiation policy was implemented dictating that the first instance of the video-on-demand application be favoured and receive resources to meet quality of service constraints. Other applications suffering as a result of this favouring were still required to deliver quality at least at 50% of their original quality of service requirements. For these experiments, this meant that the second instance of the video-on-demand application must deliver quality of at least 12 frames per second, with the same 2 frame per second deviation allowed on the lower end (to 10 frames per second), and an the same upper bound as before (26 frames per second).

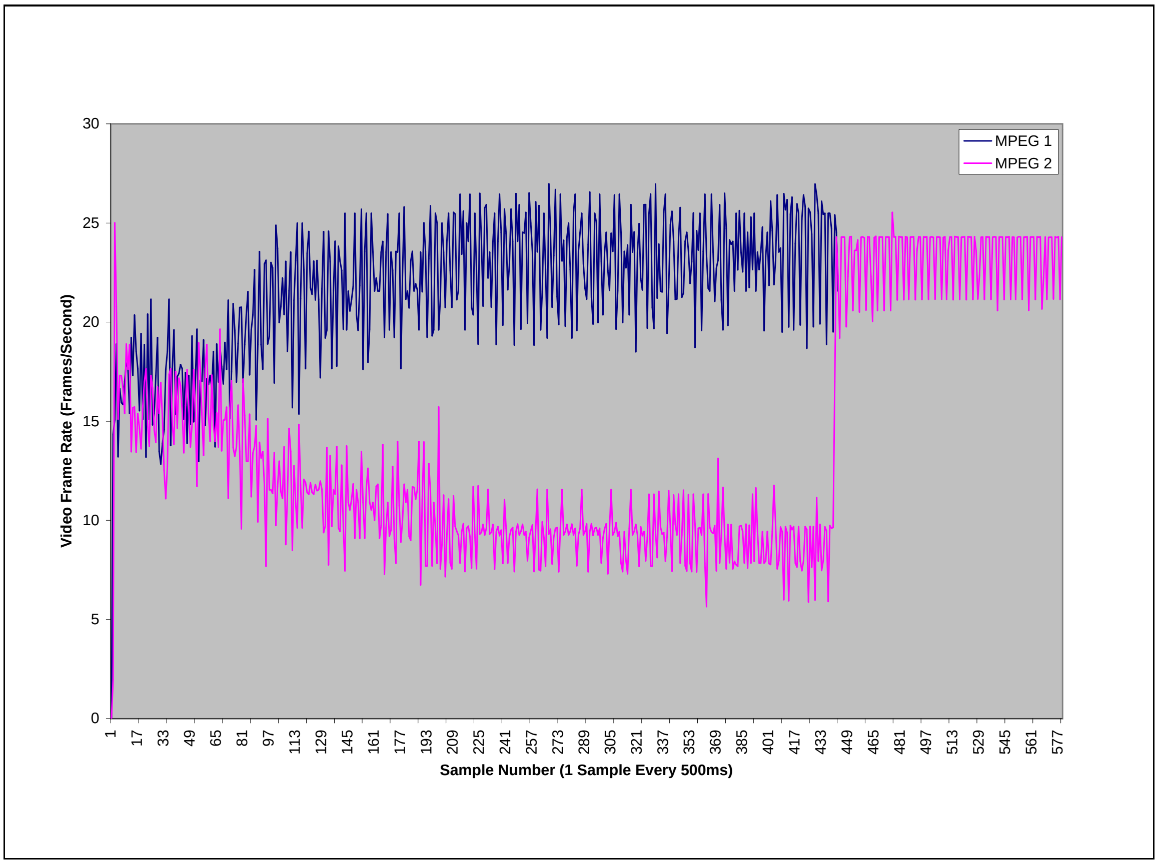
Figure 7.49 and Figure 7.50 show representative runs with the service differentiation policy in place, with resources being managed by the metric-heuristic and round robin algorithms, respectively. It is interesting to note that while the metric-heuristic approach enjoyed great success in applying the service differentiation policy, it made little difference to the round robin algorithm.

The metric-heuristic algorithm was able to enact service differentiation on average after 50 quality samples; at this point, the algorithm had exhausted its supply of CPU resources to allocate, while the demand was still high. Service differentiation was then imposed to resolve this situation. With resources freed from the second instance of the video-on-demand application and given to the first, the first instance was able to converge and deliver stable quality, while the second was still able to to meet its 50% quality requirement. There was, however, a fair amount of variation in the process.

The round robin approach was not as fortunate; in fact, it never enacted the service differentiation policies at all. Since the algorithm made more mistaken memory allocations than the metric-heuristic approach, it took the round robin approach considerably longer to exhaust CPU resources. Even then, however, the algorithm does not recognize the lack of CPU resources is an indication of the problem, as additional memory allocations could potentially solve the problem. Since the round robin approach does not use heuristics to determine when service differentiation should apply, it fails to recognize the need to do so in an attempt to satisfy quality demands with other resources. Eventually, once all resources have been exhausted, service differentiation would be imposed. Doing so, however, results in all applications suffering longer than necessary, and quality not being delivered at all in many cases.

Nevertheless, the above experimentation has shown that with differentiated services policies

**Figure 7.49** Multiple Video-on-Demand Application Experiments with Service Differentiation and Metric-Heuristic Management

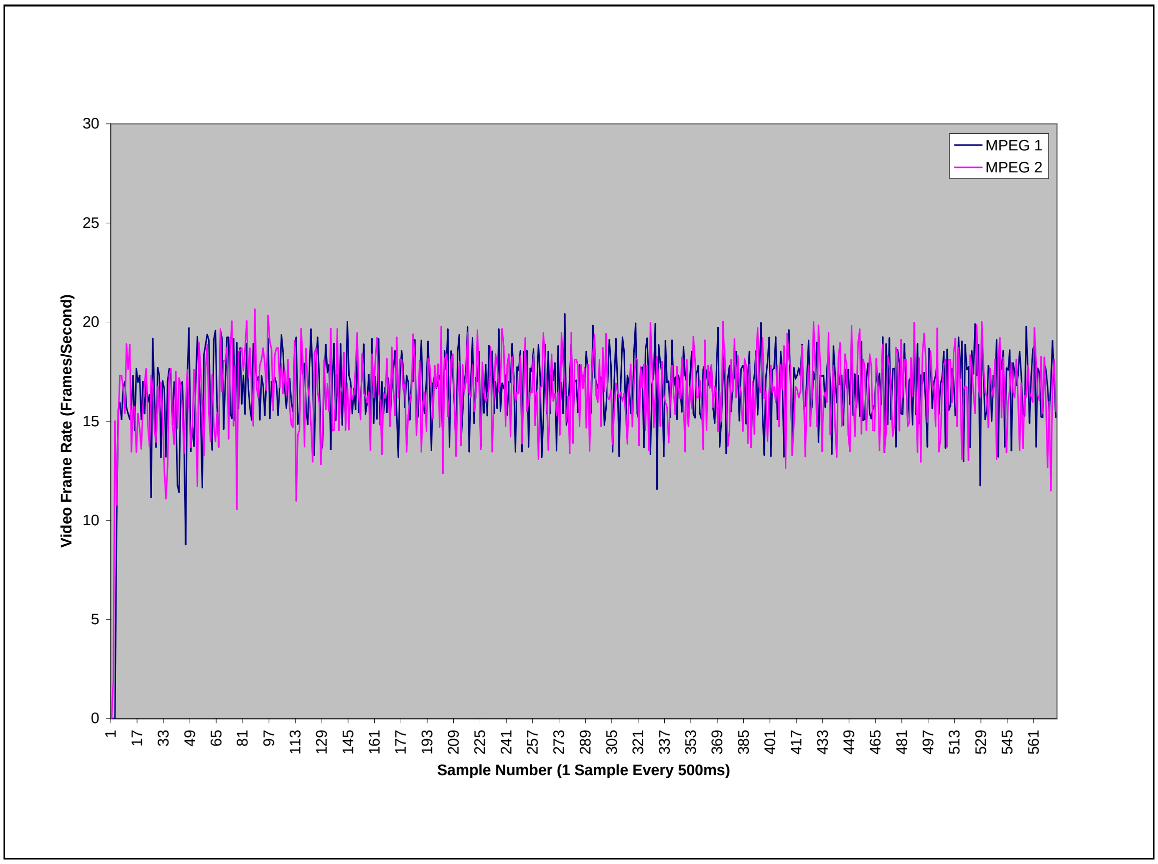



in place, we can effectively dictate which applications should receive preferential service, and which ones should not. This makes service differentiation a viable option for dealing with overload situations that would otherwise cause all applications to miss quality objectives. Experimentation has also indicated, however, that we must be careful in our resource management algorithm selection when applying service differentiation.

### 7.7.4 Multiple Enforcement Policy Experiments

Individual experimentation with three different enforcement policies for admission, control, user hints, and service differentiation produced interesting results in previous sections of this work. In this section, we examine experiments making use of all three kinds of enforcement policies at the same time.

**Figure 7.50** Multiple Video-on-Demand Application Experiments with Service Differentiation and Round Robin Management




Experimentation involved two instances of the video-on-demand application, with servers executing on `farquarson` and `vanilla`, and both clients executing on `strawberry`, as was the case with the previous service differentiation experiments. The application workload assigned was the same 10 minute 320×240 8-bit video of footage recorded by our I/O counter cameras stored in a local file used in these experiments. The expectation policy in place dictated that this workload be delivered at a mean frame rate of 24 frames per second, plus or minus 2 frames per second (yielding an acceptable quality range from 22 to 26 frames per second).

As for enforcement policies, admission control checks are performed as before to ensure the computing environment is not overloaded. User hints are now used to help guide service differentiation. When an application is obscured from view, it is marked for service differentiation in the future when resources become scarce; in such an event, obscured applications will be selected first for service differentiation, regardless of other service differentiation policies. All other things being equal, and all applications are deemed visible and usable according to user hints, the original service differentiation policy applies wherein the first instance of the video-on-demand application is favoured, and the second must deliver quality within 50% of its original requirements.

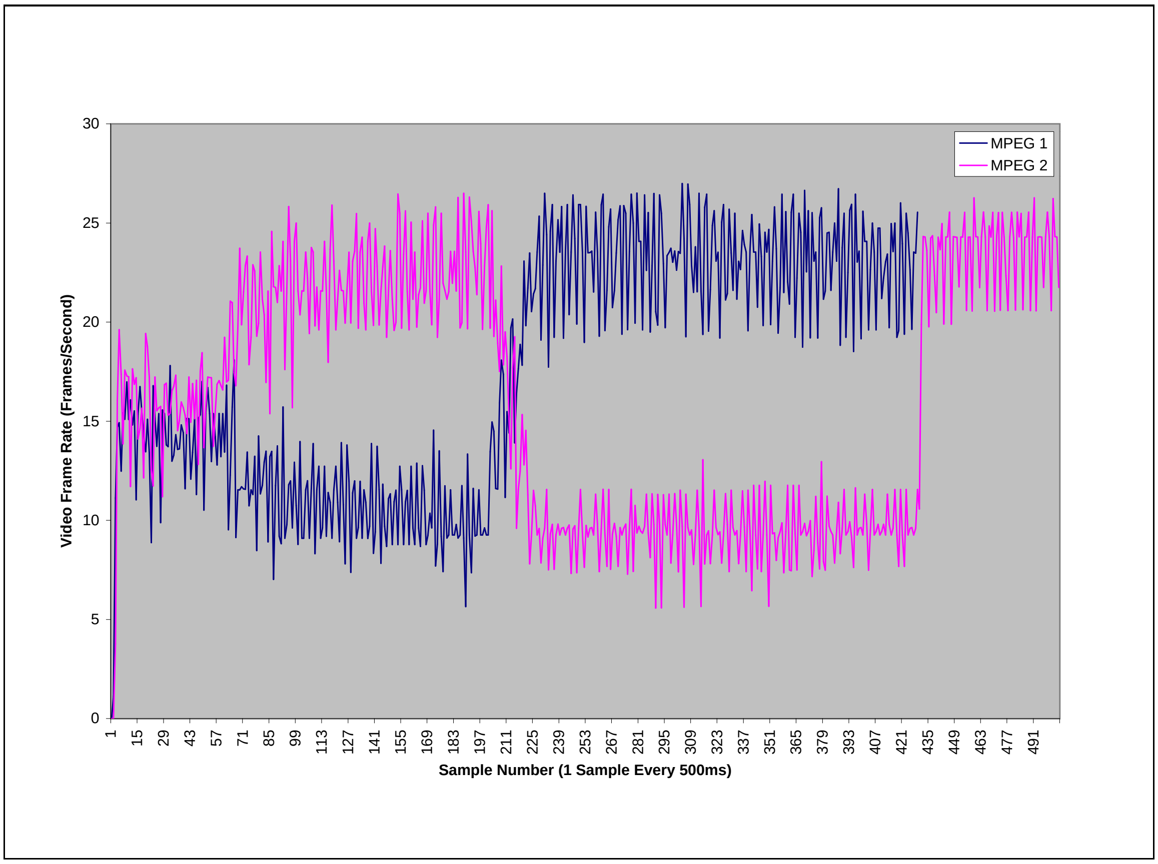
The general structure of the experiments will involve executing both applications at the same time, with the first instance being covered by the second. Since they are started nearly simultaneously, admission control would likely admit both applications, since there would be sufficient resources available at that time. User hint policies should mark the first instance to suffer under service differentiation since it is not visible. This would allow the second instance to meet quality constraints. Part way through execution, the stacking order of the video windows is reversed, so that both instances of the video-on-demand application are now visible. Since user hints no longer justify the continued hindrance of the first instance of the application, the original service differentiation policy would come into effect, and favour the first instance over the second, bringing its quality into line with requirements.

Figure 7.51 and Figure 7.52 show representative runs of this experiment using the metric-heuristic approach and round robin approach for resource management, respectively. The metric-heuristic approach appeared to be quite successful once again, while the round robin approach failed to produce acceptable results.

Under the metric-heuristic approach, both instances of the video-on-demand application are admitted. When the algorithm exhausts CPU cycles for allocation, service differentiation is enacted. Since the first instance of the application is obscured, user hints indicate that the second instance should be favoured in service differentiation. The resulting changes in resource allocation, allow the second instance to meet its quality requirements, while the first still manages to meet its 50% requirement. When the first instance is uncovered, the original service differentiation policy guides resources to this instance, allowing it to meet requirements while the second instance suffers in its place. In other words, the algorithm behaves precisely as it should.

The round robin algorithm, like the metric-heuristic approach, admitted both instances to the computing environment. However, the round robin approach never enacted service differentiation because, once again, it attempted to solve quality deficiencies through continued memory allocations once CPU resources were exhausted. Eventually, once memory resources had also been depleted, service differentiation would have been enforced with similar effects, but before this could happen, the applications had completed their workloads

**Figure 7.51** Multiple Video-on-Demand Application Experiments with Multiple Enforcement Policies and Metric-Heuristic Management



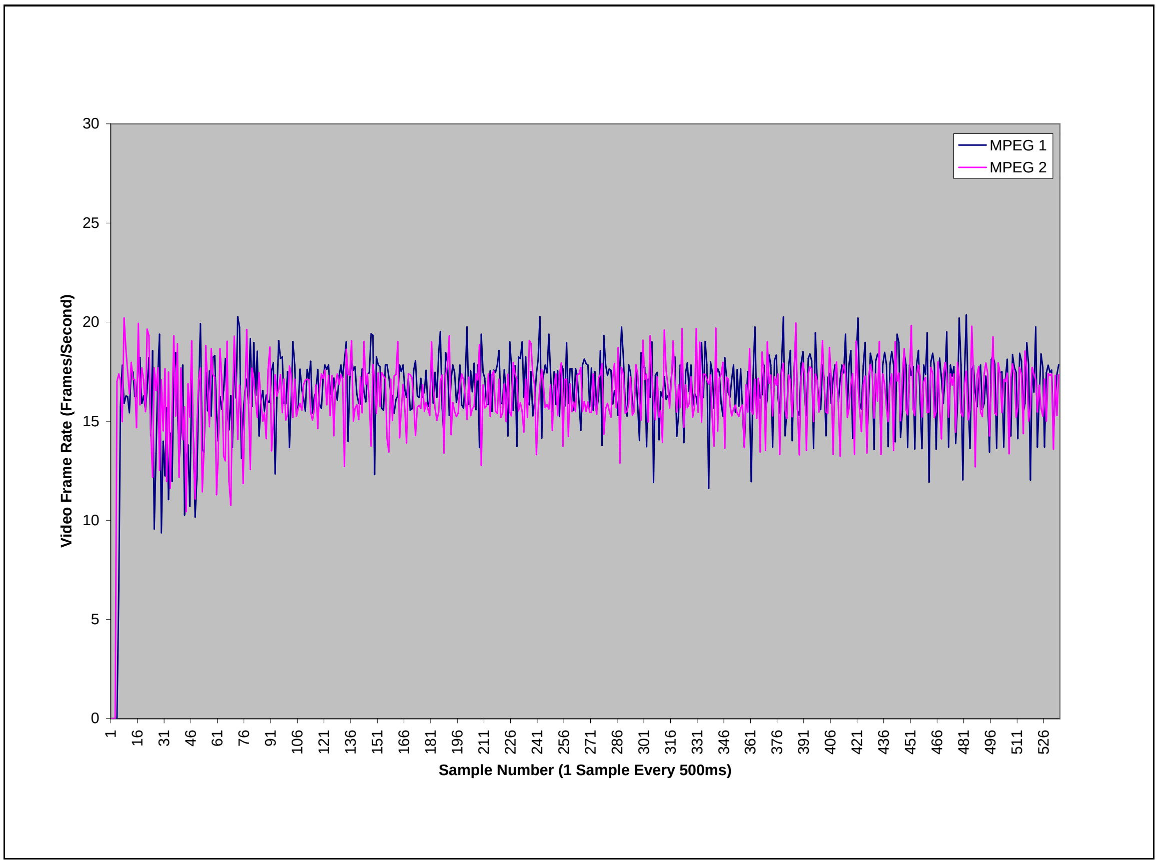

and terminated.

It would appear that it is quite possible, and useful, for several enforcement policies to be in effect at the same time. Doing so allows more control over the quality of service management of the computing environment, and resources to be directed where they are needed most depending on the situation.

## 7.8 Discussion

The previous sections in this chapter have presented a wide variety of experiments conducted under various conditions. In this section, we summarize these results and present relevant observations for discussion.

**Figure 7.52** Multiple Video-on-Demand Application Experiments with Multiple Enforcement Policies and Round Robin Management




### 7.8.1 On the Nature of Quality of Service Management

**Observation 1** *In all cases, without management support, none of the applications were able to converge or deliver stable quality of service when subjected to significant load on a single or multiple resources.*

When load was applied to either the CPU or memory, unmanaged applications were simply unable to cope and deliver acceptable quality of service. When load was applied to both, the situation worsened, and in some cases, applications were unable to make any progress whatsoever. This held for all applications used in our experimentation.

Consequently, it appears that traditional approaches to resource allocation and scheduling are indeed insufficient when it comes to delivering quality of service. Quality of service management, like that provided by our system, is clearly necessary.

**Observation 2** *Applications do not respond to resource loads in the same way.*

It was found during experimentation that different applications were affected by loads in different ways. For example, the RTVC video-conferencing application and its variants were worse off with a high CPU load, compared to a high memory load. The video-on-demand application and its variants, however, seemed to suffer more under a high memory load than a high CPU load. This is interesting considering both applications were video based applications with similar quality and timeliness needs.

It was found, however, that all application delivered their worst quality of service when both resources were loaded. The amount of difficulty encountered, however, also varied depending on the applications in question.

This indicates that attempts to classify applications to simplify quality of service management must be done with great care—just because two applications perform similar functions under similar constraints does not mean that their resource dependencies are the same. Quality of service management must be able to handle the intricacies and eccentricities of individual applications.

**Observation 3** *Making a single-process application part of a distributed application can change its quality characteristic functions substantially.*

The applications selected for experimentation were chosen, in part, because we had both distributed and single process versions of the same application, from the same code base. We found during experimentation that breaking an application into multiple parts can change the relationship between resources allocated and quality of service actually delivered.

For example, we found that two instances of the MPEG player could co-exist and deliver acceptable quality of service when executed simultaneously with no load on the system, as shown in Figure 7.13. Its distributed counterpart, the video-on-demand player, could not deliver acceptable quality of service under the same workload and circumstances, as shown in Figure 7.31, likely due to contention for disk resources. This is odd, considering that the single process version did not encounter such difficulties.

In other cases, however, we found that distribution did not have the same kind of impact. For example, behaviour of the video conferencing program was reasonably consistent in both its single process and distributed form.

This highlights the fact that we must not take anything for granted in quality of service management. Quality of service solutions must be able to adapt to handle a variety of unexpected situations.

**Observation 4** *For a quality of service management solution to be effective, it must manage all computing resources.*

In several experiments that were conducted, we found that quality could not be achieved, even with management in place, because of difficulties with unmanaged resources. In one case, discussed in Section 7.4.2.4, high memory load during experiments induced a high disk load from paging, which interfered with the ability to access stored data in a timely fashion. In another, discussed in Section 7.5.1.3, a high CPU load prevented timely system access to network data, resulting in dropped packets and reduced quality of service. Other experiments encountered similar difficulties.

These findings highlight the inherent, and sometimes hidden, dependencies between computing resources. Even though application processes may not be directly consuming a given resource, the scarcity of another resource might indirectly reduce its accessibility. To be able to handle these situations, all resource must be managed to ensure adequate levels of quality of service.

### 7.8.2 On the Performance of Our Solution and Algorithms

**Observation 5** *Whenever a managed application was able to converge, quality was stable. The degree of stability, however, could vary significantly depending on the conditions present.*

It was found in experimentation that, after convergence, quality delivered by applications managed by our system was stable. There were occasional disturbances, but quality was always able to re-converge within an acceptable period of time.

It was also found, however, that the degree of stability could vary widely depending on the applications involved, the conditions imposed, the resource management algorithms in use, and the metric used to measure stability. In terms of deviations from requirements after convergence, applications violated policies anywhere from 0.89% of the time up to 52.89% of the time on average. In terms of variation, applications delivered quality with standard deviations from 0.91 to 4.53 frames per second, and mean absolute deviations from 0.32 to 2.07 frames per second.

It is interesting to note that the different measures of stability did not always coincide; it is not uncommon to encounter a scenario with a low number of policy violations but a higher variation, or one with a high number of violations, but a lower variation. The prior case indicates a situation where quality varied wildly, hitting the boundaries of the acceptable range of quality, but usually still remaining within them. The latter case indicates that

Table 7.24 Quality of Service Convergence Time Summary

| Algorithm | Minimum (%) | Maximum (%) | Mean (%) |
|---|---|---|---|
| Metric-Heuristic | 0.73% | 39.22% | 10.63% |
| Multidimensional Search | 0.75% | 92.72% | 48.00 % |
| Round Robin | 0.74% | 71.67% | 31.12% |

quality, by and large, was quite close to the middle of the target range of quality, with more spikes in results outside the acceptable range.

**Observation 6** *All algorithms had the same general success in providing quality in that if one could converge and be stable in delivering quality, then all algorithms could.*

This is an interesting observation. It was originally thought that there would be some situations in which one of the algorithms would be successful in delivering acceptable quality of service and another would consistently fail. This did not arise, however, during our experimentation. Whenever one algorithm could converge, they could all do so. To a certain extent, this makes sense, since all of the algorithms are based upon the same dynamic approach to resource management, but it was thought that their differences might cause more variation in results.

It is important to note, however, that the degree of success enjoyed by the algorithms varied widely, depending on the situation. This will be discussed further below.

**Observation 7** *When quality of service requirements could be met, the metric-heuristic algorithm tended to perform the best in terms of convergence time and stability.*

As mentioned above, all three algorithms were able to converge under the same experimental conditions. However, the three algorithms converged at different times in the life of the applications they were managing. Table 7.24 shows the minimum, maximum, and mean convergence times over all experiment configurations for each algorithm. Convergence time is expressed as the percentage of application life-time passed when convergence occurred.

As can be seen, the metric-heuristic approach was superior on all counts, posting the best minimum, maximum, and mean convergence times. The round robin approach was next best, but posted maximum and mean results far higher than the metric-heuristic approach. The multidimensional search approach performed the worst in terms of convergence times, with maximum and mean results even higher than the round robin approach. With results this high, where applications can expect to spend nearly one-third to one-half their life on average without adequate quality of service (or much more in worst case scenarios), one must question the suitability to task of these last two algorithms.

**Table 7.25** Quality of Service Policy Violation Summary

| Algorithm | Minimum (%) | Maximum (%) | Mean (%) |
|---|---|---|---|
| Metric-Heuristic | 0.87% | 42.24% | 14.21% |
| Multidimensional Search | 9.52% | 39.92% | 27.15% |
| Round Robin | 2.20% | 52.89% | 23.10% |

**Table 7.26** Quality of Service Standard Deviation Summary

| Algorithm | Minimum (FPS) | Maximum (FPS) | Mean (FPS) |
|---|---|---|---|
| Metric-Heuristic | 0.77 FPS | 2.07 FPS | 1.28 FPS |
| Multidimensional Search | 0.67 FPS | 4.53 FPS | 1.92 FPS |
| Round Robin | 0.89 FPS | 3.25 FPS | 1.71 FPS |

In terms of stability, we first look at the number of policy violations encountered by each application after convergence has occurred, as shown in Table 7.25. Once again, we look at minimum, maximum, and mean values. Numbers quoted represent the percentage of samples taken after convergence that violated quality of service requirements.

Once again, the metric-heuristic approach seemed to do the best of the three algorithms. It scored the best minimum and mean results, but had the worst maximum result. In all fairness, however, this algorithm averaged this percentage of violations in only one experimental configuration, and the next closest was more than 12% below this result. The round robin approach faired next best in most aspects, followed again by the multidimensional search approach. The multidimensional approach posted the lowest maximum value of the three algorithms; however, it must be noted that the experiment configurations producing the higher maximum values for the other algorithms also resulted in convergence times higher than 90% for the multidimensional search algorithm. Consequently, it was much easier for this algorithm to have fewer violations, since there was less than 10% of its life remaining; the other two algorithms had much longer to maintain stability.

Lastly, we consider the variation in quality of service delivered after convergence, as measured by standard and mean absolute deviation. These results are shown, respectively, in Table 7.26 and Table 7.27. Once again, minimum, maximum, and mean values are given. This time, results are reported in terms of the quality metric used for management, frames per second.

The results between the two different measures of variation seem to coincide reasonably well. The metric-heuristic approach again appears to be the best algorithm in this respect, posting the best maximum and mean results. It was bested by the other algorithms only in terms of the minimum deviations reported. The round robin approach was next best posting the best minimum deviation and the second best maximum and mean deviations.

**Table 7.27** Quality of Service Mean Absolute Deviation Summary

| Algorithm | Minimum (FPS) | Maximum (FPS) | Mean (FPS) |
|---|---|---|---|
| Metric-Heuristic | 0.47 FPS | 1.01 FPS | 0.65 FPS |
| Multidimensional Search | 0.41 FPS | 2.07 FPS | 0.94 FPS |
| Round Robin | 0.32 FPS | 1.62 FPS | 0.87 FPS |

**Table 7.28** Quality of Service Resource Overhead Summary When Converged

| Algorithm | Minimum (%) | Maximum (%) | Mean (%) |
|---|---|---|---|
| Metric-Heuristic | 0.14% | 1.81% | 0.89% |
| Multidimensional Search | 1.04% | 6.07% | 2.89% |
| Round Robin | 0.27% | 3.66% | 1.81% |

The multidimensional search approach again trailed behind the other two algorithms.

Overall, when quality of service requirements could be met, and convergence resulted, the metric-heuristic approach performed the best, followed by the round robin, and then the multidimensional search approaches. This was particularly true in distributed and multi-application experiments, as the metric-heuristic approach could effectively eliminate resources from management decisions, which translates to large savings in the long run. Whether these findings hold when other criteria are examined remains to be seen.

**Observation 8** *The metric-heuristic algorithm tended to consume the most resources on average.*

When we first looked at various algorithms in Chapter 5, we felt that the metric-heuristic algorithm was likely going to be more costly than the multidimensional search and round robin approaches, because of the additional time required for collecting and analysing metric data used in making resource management decisions. At first, however, experiment results seemed to show the opposite.

Table 7.28 shows the resource overhead for the three different algorithms, in terms of the percentage of CPU time consumed by management during experiments. As an important note, this table only presents results for cases in which convergence occured. Once again, minimum, maximum, and mean values are presented. It would appear from these results, that the metric-heuristic approach posts the best results with the lowest minimum, maximum, and mean overhead results. The round robin approach fared the next best, followed by the multidimensional search approach, which appeared to be quite heavy in resource usage, on average. This would seem to go against our earlier speculation.

An entirely different story, however, is told when we only examine cases in which convergence did not occur. These results are shown in Table 7.29. In this case, the metric-heuristic

**Table 7.29** Quality of Service Resource Overhead Summary When Could Not Converge

| Algorithm | Minimum (%) | Maximum (%) | Mean (%) |
|---|---|---|---|
| Metric-Heuristic | 2.58% | 7.58% | 5.13% |
| Multidimensional Search | 0.61% | 6.50% | 3.55% |
| Round Robin | 1.96% | 6.29% | 3.45% |

approach fared the worst in all aspects. The round robin approach seemed to do the best in most respects, except for the minimum resource usage, which the multidimensional search performed best. It should be noted, however, that the cases where the multidimensional search reported such low resource consumption were those in which it missed requirements by a tiny margin, where the other approaches were successful in converging.

This last set of results, in which quality requirements are missed, is perhaps a better measure of the resource consumption of each algorithm than the previous table, in which requirements are met. When requirements are constantly missed, each algorithm must work constantly to attempt to meet quality needs. On the other hand, when convergence occurs, management no longer needs to work as hard, because there are fewer violation reports to handle. In these cases, the resource usage will be affected by the convergence time of the algorithm.

Consequently, referring to Table 7.29, when convergence did not occur, we see that the metric-heuristic approach was the more costly of the three algorithms, when the same amount of work needed to be done. As a result, this would indicate that each iteration of the metric-heuristic approach tends to be more expensive than the other algorithms, on average. Despite this, however, we can see that the metric-heuristic approach is generally less expensive than the other approaches when convergence occurs, because it can converge quicker than the other algorithms—it has less violation reports to deal with, and less work to do in such cases. So, it would appear that the cost of the metric-heuristic approach is not significant when it can converge as quickly as it has been able to do in experimentation.

**Observation 9** *The most important factors to resource overhead appear to be the time to converge and the degree of stability provided.*

This observation follows from the above discussion. We saw that the algorithm that was most expensive on a per-iteration basis has the lowest resource utilization on average when quality meets requirements. We also observed that algorithms that are less expensive on a per-iteration basis, the multidimensional search and round robin approaches, take longer to converge and have significantly higher resource utilization, on average, when quality meets requirements.

**Table 7.30** Quality of Service Failure Summary

| Algorithm | Minimum (%) | Maximum (%) | Mean (%) |
|---|---|---|---|
| Metric-Heuristic | 73.80% | 100.00% | 93.64% |
| Multidimensional Search | 37.30% | 100.00% | 78.39% |
| Round Robin | 48.92% | 100.00% | 86.25% |

In addition, we saw that cases in which convergence to meet requirements did not occur consistently consumed significantly more resources than those cases that did converge. This difference was observed regardless of the algorithm in use. As an important note, however, we see that this increase in resource consumption comes from persistent complaints from application processes of missed quality objectives. These quality of service requirement violation reports cause the QoS Host Managers to attempt to resolve the problem, even though there are insufficient resources to do so. This process results in the increase in resource overhead. We can ease this increase somewhat by instructing application instrumentation to stop reporting policy violations until resources become available, or by employing service differentiation, which will favour selected processes, and instruct those that are not to be silent until further notice.

From these observations, we can conclude that the resource consumption benefits of converging early and delivering stable quality outweigh the higher costs of the algorithm. This introduces an interesting trade-off, in which it may be desirable to adjust algorithm parameters to use more resources in the short-term to quickly converge to meet quality requirements and then reap more benefits in the long term, as opposed to converging slower over time to keep per-iteration overhead lower. This issue needs further investigation.

**Observation 10** *When quality of service requirements could not be met, the metric-heuristic algorithm tended to perform the worst of the three algorithms.*

This is an interesting result. When we examine the behaviour of the algorithms when quality of service did not converge and was not stable, the metric-heuristic algorithm performed worse than the other two algorithms. Table 7.30 shows the minimum, maximum, and mean percentage of time that each algorithm failed to meet requirements when convergence did not occur. In a reversal from previous observations, the multidimensional search approach performed better than the other two algorithms, followed by the round robin approach, and lastly the metric-heuristic approach.

When we examine the distance between delivered quality of service and expected quality in these cases, we see a similar pattern. Table 7.31 shows the difference between delivered and expected quality of service in frames per second, as well as how close delivered quality of service came to expected quality as a percentage. Once again, the multidimensional search

**Table 7.31** Quality of Service Failure Distance Summary

| Algorithm | Minimum | Maximum | Mean |
|---|---|---|---|
| Metric-Heuristic | 2.41 FPS (89.96%) | 21.64 FPS (9.84%) | 8.57 FPS (63.25%) |
| Multidimensional Search | 1.17 FPS (95.13%) | 20.59 FPS (14.20%) | 6.28 FPS (73.16%) |
| Round Robin | 1.87 FPS (91.51%) | 17.40 FPS (27.50%) | 6.63 FPS (71.74%) |

approach fared best, followed by the round robin approach, and finally the metric-heuristic approach.

At first, it seemed quite unusual for the metric-heuristic approach to perform so well in most other aspects, but so poorly in adverse conditions. Its poor performance in situations where quality of service requirements could not be met can be attributed to the resource consumption of the algorithm. As was noted earlier, the metric-heuristic approach consumes more resources than the other algorithms. When application quality requirements cannot be met, it is because the applications cannot be furnished with sufficient resources. With its higher per-iteration resource overhead, the metric-heuristic approach leaves even fewer resources available to struggling applications, causing the applications to perform worse. With the multidimensional search or round robin approaches, fewer resources are consumed by management itself, leaving more available to applications, allowing them to deliver better quality.

**Observation 11** *When quality of service requirements could not be met, management still provided a boost in actual quality delivered.*

It was found that in experiments where quality could not converge to meet requirements, that there still tended to be improvements over similar situations in which management support was not provided. In fact, we found that in almost 95% of such experiment cases, there were either fewer quality violations, or the difference between actual delivered quality and expected quality had shrunk when management was present. Consequently, even in seemingly hopeless situations, quality improvements can still be realized through proper management of computing resources.

**Observation 12** *Enforcement policies can have a positive effect on quality of service.*

In some of our latter experiments, we employed three different sets of enforcement policies relating to admission control, user hints, and service differentiation. Admission control is intended to be applied before or at the beginning of an application's life. User hint handling is intended for use throughout an application's life to route resources to applications that are the focus of user activity. Service differentiation is primarily intended to handle situations of resource shortages, when quality objectives cannot be met for all applications simultaneously. As such, all sets of enforcement policies can be used at once quite effectively for interesting results.

Experimentation with admission control, user hints, and service differentiation found that resources could be effectively routed to applications based on usage, ownership, application type, start-up time, and a variety of other factors. This allows quality of service requirements to be met in applications according to policies selected by administrators or individual users. Consequently, more applications can be supported by a system, and quality can be provided where it is wanted or needed most.

Through the careful selection of enforcement policies, resource management decisions governing quality of service can be effectively guided. This gives more power and flexibility to our quality of service management system.

**Observation 13** *The metric-heuristic algorithm out-performed the round robin algorithm with enforcement policies in place.*

In earlier experimentation involving enforcement policies, comparisons were made between the performance of the metric-heuristic approach and the round robin approach to resource management. (The multidimensional search approach was omitted due to earlier findings.) In almost all respects, under each kind of enforcement policy, the metric-heuristic approach performed better than the round robin approach.

The round robin approach scored best in areas dictated by single iteration efficiency and low overhead, where the simplicity of the algorithm gave it an advantage. In most areas, however, the better decisions made by the metric-heuristic algorithm allowed it to analyse the situations it faced, and act more appropriately. While this generated more overhead, this ability gave the metric-heuristic algorithm an edge in most respects.

While it would appear that the metric-heuristic approach is the overall choice as a resource management algorithm, the round robin approach may still have its uses, particularly when resources are extremely tight and a low overhead solution is necessary. However, in most realistic operating situations, it would appear that the metric-heuristic approach is the algorithm of choice.

### 7.8.3 On Potential Drawbacks and Pitfalls to Consider

**Observation 14** *Practical issues can make using the multidimensional search problem difficult to deploy and use.*

While the multidimensional search algorithm has several interesting qualities and appears quite promising in theory, there are several aspects of implementing and using it that present problems in practice, some of which were mentioned earlier in Section 5.3. These aspects, together with mediocre performance, all but rule out the multidimensional search as a practical resource management algorithm.

- During our experiments, as mentioned throughout this chapter, we were able to obtain information necessary to initialize the multidimensional search algorithm by regression analyses of base case experiments conducted without management. Using these results, we could split the target quality specified by expectation policies into a vector specifying the portion of quality that must be provided by allocations of each resource. In practice, however, conducting such experiments for each application installed on a system would be tedious and time consuming.

  If system configuration was to change, through the addition or removal of a computing resource, we would need to change policies guiding the initialization of the multidimensional search for each application. In such cases, we would also likely need to conduct new regression analyses to give us the necessary data to develop the rules. Once again, this would be incredibly cumbersome.

  An adaptive version of the algorithm could be written to do the required analyse at run-time to generate the data for composing the rules. Unfortunately, such an algorithm would likely be complex to produce. Even it was simple to do, one would expect severe quality problems on the first few runs of any new application on the system, which is quite likely an unacceptable compromise.

- When an application registers to a host manager using the multidimensional search algorithm, the host manager must initialize state used in the algorithm, as discussed in Section 5.3. One piece of state information needing initialization is the current amount of delivered quality attributable to each resource. In the algorithm outlined in Algorithm 3, this vector was initialized to all zero values, since the application was to start with no resources allocated.

  In practice, however, things are not so simple. By the time of registration, an application is already executing with some default resource allocation made by the operating system, and is already delivering some non-zero level of quality. The question then becomes how to attribute this quality to the different resources allocated to the application. Unfortunately, there is no way to be certain how this split should be made accurately. If this split is done incorrectly, the algorithm will be more prone to error, since it will always be mistaken about its current position in the quality search space.

This problem arose and was mentioned in Section 7.3.3.

One possible solution would be to discard the resource allocations made by the system at registration and start the program at the minimal allocation required, so we could initialize the starting quality vector to zero values. In practice, however, experiments have shown that this causes an application to take an excessive amount of time to execute sufficiently to generate enough policy violation reports to get its quality adjusted to acceptable levels, particularly when the system is already loaded. Consequently, this approach is not feasible.

In the end, the algorithm had no choice but to estimate its initial split of quality based on the same split used to break up the single target quality value into its vector. This procedure, unfortunately, is likely to be error prone, and could explain some of the difficulties encountered by the algorithm.

**Observation 15** *Delays in quality changes after resource adjustments could have adverse effects on both the multidimensional search and round robin approaches.*

Both the multidimensional search and dynamically weighted round robin approaches assume that, whenever a resource adjustment is made, any change in quality will occur by the next iteration of the algorithm. If this assumption holds, the algorithms know to which resource to attribute the change in quality, to help guide future resource allocation decisions. However, if this assumption does not hold, and changes in quality appear later, the algorithms may inadvertently credit changes in quality to the wrong resource. This, naturally, would cause difficulties in converging to quality requirements.

Unfortunately, analysis of experiments has indicate that this assumption does not always hold, and quality changes are not observed until after subsequent iterations of the algorithm. This can be handled by increasing the minimum time between policy violation reports, which trigger the next iterations of the algorithm, but this would also make management less responsive in general. A solution to this dilemma is still being sought.

**Observation 16** *Collecting accurate metric data for the metric-heuristic algorithm can be difficult, and tends to require the collection of several pieces of data for each resource, as a result.*

Even the metric-heuristic algorithm has some difficulties when put into practice. Despite the fact that most operating systems provide interfaces to collect a variety of statistics on applications and resource usage, a great deal of this information is still not collected or provided. Since this ideal information is not available, more work needs to be done to be able to make reasonable analyses and decisions.

For example, the page fault recorder in our physical memory manager taps into the operating system to retrieve page fault information on individual processes. While the slots for this information are made available, the operating system does not appear to use them in practice whatsoever. Consequently, while we measure this data, the results are essentially useless, and we cannot determine the page fault rate for individual processes.

To compensate, we need to collect much more data, including the global page fault rate for all processes on the system, the amount of time dedicated to paging activity, the number of free virtual memory pages, and the percentage of each process's address space that is currently resident. By juggling all of these values together, we can say with reasonable certainty when a memory problem exists on the system, but it is still difficult to conclude absolutely that a given application is suffering because of it. The best we can do is use the existence of a system-wide memory problem as a predictor for per-process problems. This is unfortunate, when this information should be made available to us.

Similar problems have been encountered in obtaining metric information for other resources. However, many of those problems have been overcome largely by digging deeper into the operating system and examining internal system source code. Nevertheless, an absence of solid, accurate metric data poses some difficulties for the metric-heuristic approach. Fortunately, this issue is far less serious than the problems afflicting the multidimensional search approach.

**Observation 17** *Simple versions of the round robin approach do not appear to have the kind of practical, implementation issues that the other algorithms do.*

Because of the simplicity of the round robin approach, there do not seem to be the same kinds of implementation problems that the other algorithms do. There are no dependencies on outside data; the basic forms of the algorithm do everything internally. With this simplicity also comes low overhead; however, it also comes with a cost of potentially poor convergence and stability results, as shown by our experimental results.

**Observation 18** *Under heavy loading conditions application quality can suffer heavily at first until management can take over.*

As noted in Section 7.3.3, and observed elsewhere, it is possible for applications to spend a long time after they are launched before they can even reach our management system for help. Consequently, there can potentially be a large start up delay for an application before quality can be delivered, especially under high loads. This is because an application must first register and then report a quality problem before our quality of service management system can do anything; under high loads, it might take some time for an application to do even that.

We can improve the situation somewhat by giving applications additional resources on registration, perhaps by tracking the last resource allocation given to the application in its most recent execution. Unfortunately, even in such a case, there could still be a significant delay from the launch of an application until registration. In the mean time, we could do nothing. The only way around this, would require either modifications to the operating system itself, or the development of a new system shell that somehow involves the quality of service managers earlier in the execution of a new application. Further investigation into this issue is required.

# Chapter 8

# Conclusions

The general goal of this work was to develop a new approach to quality of service resource management that addresses many of the outstanding problems and issues in the field. Starting from an analysis of the background and related work in the area of quality of service management, key problems were identified. A theoretical foundation was laid examining these problems, which demonstrated the inherent complexity in formulating direct algorithmic solutions. This led to the development of a heuristic approach to quality of service management, including a general quality of service resource management architecture, and a number of resource management algorithms to be used in this framework. Based on the implementation of a prototype system, experiments were conducted to both validate our approach and investigate its effectiveness and efficiency. Results gleaned from these experiments are positive and promising.

In this chapter, we first highlight the key findings and accomplishments of this work. We then conclude this work by presenting several avenues for continued research in the future.

## 8.1 Contributions

The major contribution of this work is the general approach to quality of service management developed to meet the goals and requirements outlined in Section 2.3. To briefly summarize, our solution has the following characteristics:

**Dynamic Quality of Service Management:** Our approach is dynamic in that it will adjust resource allocations throughout an application's life to meet current quality of service needs. As such, it can compensate for translation difficulties from quality of service requirements to resource allocations, as well as changes in user requirements, application needs, or computing environment conditions. Further, we do not need to know in advance the relationship between resources allocated and quality delivered as a result—our dynamic approach will adapt resources to meet quality needs regardless.

All components of our quality of service resource management architecture are dynamic, allowing the tuning of management behaviour at run-time to reflect these changes. For example, instrumentation sensors can adjust their thresholds to detect potential violations of new quality of service requirements.

**User/Application Level Quality of Service Management:** All quality of service specifications are made at the application level. Our approach will dynamically determine the resources required to meet this specification, as discussed above. Some elements of user behaviour, in the form of user hints, are tracked and can be used to guide the management process.

**Environment Adjustments First, Then Application Adjustments if Necessary:** All attempts are made to adjust the computing environment to the applications being managed, typically through resource allocation adjustments. If these adjustments fail to deliver acceptable quality, functionality is in place to support application adjustments to rectify the situation.

**Application Independence:** No assumptions are placed on the kinds of applications to manage. As long as the appropriate instrumentation is added, virtually any application is manageable. In practice, our implementation has been used to manage a variety of applications.

There are also no assumptions placed on the programming language or programming environment needed. As long as language bindings are provided to use the instrumentation library, or if the instrumentation library is ported to the language natively, applications can be managed by our approach. Thus, even though the quality of service management system is implemented in C, C++, and CLIPS, applications written in Java or other languages can still be managed.

**Computing Environment Independence:** No assumptions are placed on the computing environment whose resources are being managed. There are also no assumptions made of the operating system or middleware software in the computing environment. As long as we construct resource managers to manage its resources, our solution will work.

**Algorithm Independence:** No assumptions are placed on the resource management algorithms in use. Our algorithms can be changed without affecting the rest of the solution, and vice versa.

**Resource Independence:** No assumptions are placed on the types of resources managed, or their characteristics. Virtually any computing resource can be managed, as long as a resource manager for it is constructed conforming to our generic resource manager interface. Consequently, our approach is end-to-end; capable of managing all computing resources required to support any application.

While we have yet to implement resource managers to manage all computing resources, we have had a great deal of success using the ones at our disposal while new managers for other resources are under construction. Further, adding these new resource managers to our solution does not affect the rest of the solution, except for the resource management coordinator, which will only require the modification of simple rules.

A great number of these qualities do not exist in current work in this area, as discussed earlier in Chapter 2. As indicated in Section 2.3.5, this work is unique, and is different from other related work in this area.

In addition to its general approach to quality of service management, this work has produced other achievements. This includes the following:

- We have demonstrated that most realistic quality of service resource management scenarios are NP-complete problems.
- We have designed several resource management algorithms with various characteristics to support improved quality of service. In addition, we have provided variations of each algorithm, as well as possible future extensions.
- We have developed a working prototype of a quality of service management system. This includes instrumentation to interface with application processes, resource managers to manage various computing resources, and rules to support a variety of enforcement policies. All of this was possible without any operating system modifications required, which entailed some rather interesting technical innovations, as discussed in Chapter 6.
- We developed an experimental testbed to facilitate quality of service management experiments. Not only has this testbed been used for conducting experiments in this work, it has also been employed in experiments carried out to support other theses and research.
- Using our experimental testbed, we have carried out a variety of experiments to validate and investigate the intricacies of our quality of service solution. Through our experiments, we have been able to shed new light on the problem of resource management for quality of service.

## 8.2 Future Work

The contributions of this work to the area of quality of service management represent a great deal of progress towards providing a comprehensive solution to ensuring end-user satisfaction. There are, however, many issues still outstanding that need to be addressed, and many possible extensions to this work that need to be investigated. In the sections below, we present and discuss these directions for future research.

### 8.2.1 Exploring Constraints and Restrictions on the Problem

In Chapter 3, we examined various instances of the quality of service resource management problem under different conditions. Some cases were found to be in P; others were found to be NP-complete. It would be interesting to revisit those cases that were NP-complete to see if it is possible to impose reasonable constraints or restrictions that would place the problem in P instead. In doing so, we may be able to develop more efficient or more accurate algorithmic solutions to directly solve the problem in these conditions.

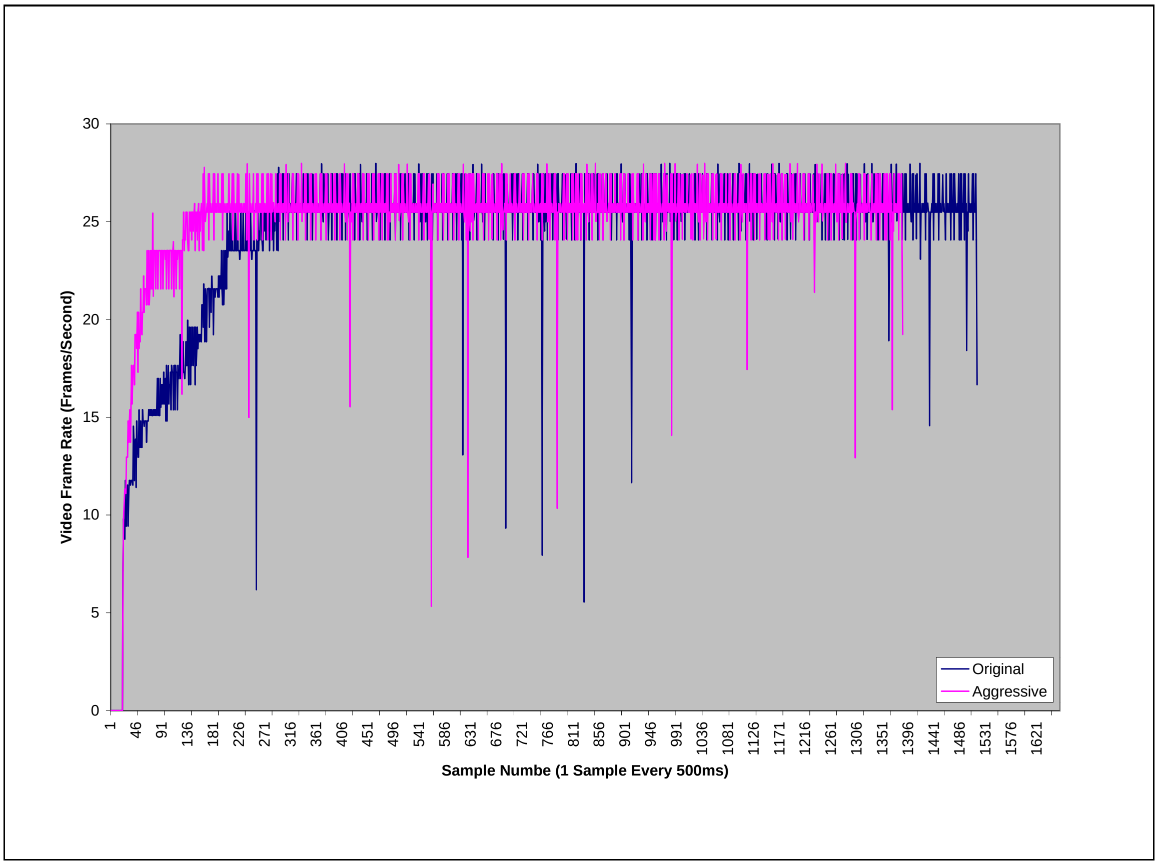
### 8.2.2 Further Algorithm Development

The algorithms presented in Chapter 5 take a variety of different approaches to quality of service resource management. Some are quite simple, such as the round robin algorithm, while others are more complex, such as the multidimensional search and metric-heuristic algorithms. In the future, it would be interesting to investigate both entirely new algorithms, as well as additional variations to these algorithms. While we have obtained acceptable performance using these algorithms, we may be able to do even better with further analyses.

Further investigations are also required to fine-tune the multitude of parameters to the existing algorithms already developed and implemented, including initialization parameters, adjustment parameters, adaptation parameters, and so on. Many of these parameters can be changed at run time by changing enforcement policies used by host managers; others can be changed only by restarting the managers entirely. None of the parameters require recompilation or system rebooting.

As an example of the effects of changing a tuning parameter, consider Figure 8.1, which presents two representative runs of the metric-heuristic approach. Both runs are from the RTVC video conferencing program on `farquarson` Both runs instruct the program to capture and display 18000 frames of a 320×240 8-bit video stream at a frame rate of 26 frames per second, plus or minus 2 frames (giving an acceptable range of 24 to 28 frames per second). In both runs, the metric-heuristic algorithm was managing only the CPU as a

**Figure 8.1** Results of Tuning: More Aggressive Resource Allocation




resource. The only difference between the two runs is that one run, labelled "Aggressive" was instructed to double the size of resource adjustments compared to the "Original" size used by default. The result was that quality converged at 9.96% of the run for the "Aggressive" approach, as compared to 17.69% for the "Original" approach.

In other words, doubling the size of resource adjustments in this case resulted in cutting convergence time nearly in half. The drawback of doubling resource adjustments is that it may become harder to converge to meet a tighter range of quality requirements, as the now larger adjustment size might miss its target, shooting above or below the target range. With a large enough range of acceptable quality to aim for, doubling the size of adjustments should not have serious problems. Nevertheless, more experimentation with further tuning is clearly required.

### 8.2.3 Additional Resource Support

As experiments in Chapter 7 indicated time and time again, management of all computing resources is required for a complete quality of service solution. At present, we have implemented support for CPU management (both time-sharing and real-time), and physical memory management. As was mentioned in Chapter 6, our research group has already begun work on providing DiffServ support [3] and RSVP support for quality of service management. Further work, however, is required to support disk I/O resources.

Another avenue for research is the concept of application-specific resource management. The general idea is that many applications provide services that are essentially "resources" for use either by themselves or other applications. These application-specific logical resources can be adjusted and tuned in much the same way that we can manage physical computing resources. For example, consider a database management system that provides database resources in the form of buffer pools, caches, locks, and so on. The performance of database applications can be tuned by adjusting and controlling their usage of these resources. (As noted earlier, some work towards this end can be found in [122, 123].) As another example, consider a web server with a configuration file dictating the number of server processes to run, memory pools, caching, and other parameters. By tuning these parameters at run time, we can tune the performance of the server and client applications it supports. By structuring code to control and adjust these application-specific logical resources as traditional resource managers conforming to our standard interface, they can be easily and effectively managed by our existing quality of service management infrastructure.

Finally, as was the case with algorithms, existing resource managers have been coded with many parameters that can tune and control their behaviour, performance, and overhead at run time. Once again, some of these parameters can be tuned while the managers are executing; others, require the managers to restart. Further experiments are required to fully investigate this issue.

### 8.2.4 Porting Efforts

As was mentioned in Chapter 6, our existing prototype has been implemented for the Solaris 2.6 operating system using Sparc or UltraSparc processors (the prototype would most likely function correctly using the same operating system on the Intel platform, but it has not been tested). All platform specific code, however, is limited to the resource managers, and some sensors with very specific functionality. Consequently, porting our system to other platforms should not be exceedingly difficult. Target platforms in mind include Linux, as well as Microsoft Windows (likely NT, 2000, or some version of XP).

Another interesting porting effort would be to the Java virtual machine. As a "virtual machine", Java has resources that must be allocated and scheduled for its applications, applets, and servelets; for quality of service support, management of these resource is also essential. Tying the management of Java logical resources to the management of the underlying physical computing resources would pose interesting challenges. As a first step, a Java port of our instrumentation library is currently under way.

### 8.2.5 Investigating Application Adaptations

When quality of service expectation violations are detected, our management system responds by attempting to adjust the computing environment to application needs through resource adjustments, as mentioned earlier. When that approach fails, due to insufficient supplies of computing resources to meet demands, we have provided service differentiation mechanisms to determine which of the executing applications should receive preferential service, and which should suffer. There are alternatives to doing this, however.

One such approach is to request that applications adapt to the computing environment to continue meeting requirements when it is not possible for the environment to adapt itself. As mentioned in Section 6.1.3.4, we have provided the basic functionality required to support this in the form of quality adaptation actuators. However, we have yet to derive any classes of actuators from this class, or use them in applications. Consequently, further implementation and experimentation are required to investigate what kinds of application adaptations are useful, and how application adaptations can affect delivered quality of service.

### 8.2.6 Proactive Quality of Service Management

By the time application instrumentation detects the violation of a quality of service requirement, it is, to a certain extent, too late. The application user has already experienced the unsatisfactory level of quality of service, and will continue to do so as the quality of service resource management system determines the necessary adaptations to resolve the situation, and puts them in place. This problem occurs in all reactive approaches to quality of service management, particularly in dynamic situations when the management system in place must keep up with constantly changing application resource needs.

One possible solution to this problem, is to be more proactive or predictive in quality of service management. In such a case, we would attempt to determine in advance when resource needs for an application were about to change, and ensure that the necessary adjustments were in place by the time the new resource allocations were required. While preliminary work in [70, 72, 190, 193] looks promising, further investigations are required.

### 8.2.7 Instrumentation Automation

Making an application manageable in our environment is reasonably simple; inserting the necessary instrumentation probes at the appropriate locations and linking code with our instrumentation library is all that is required. This, on its own, is a far cry from the effort and knowledge developers must put towards bringing quality of service to applications without using our approach.

All of the applications that we have instrumented for quality of service management are third-party application of which we had no prior knowledge. From our experiences, instrumenting them took little time. For example, we were able to instrument the source code for Id Software's popular game Doom [78] and begin experimentation in less than an hour after the code was received and test-compiled. When instructed to use our instrumentation library to make an application manageable, a recent senior class in Distributed Application Management found the process more difficult, but attributed problem to a lack of available documentation, which is currently being rectified.

Nevertheless, we have recognized that tools to automate the instrumentation of application would be extremely useful to developers. To that end, work in our research group has examined this problem [71, 96]. There is still much more that could be done in this area, however.

### 8.2.8 Managing Application with Dependencies

We currently manage each application separately, and assume that they are independent, as dealing with the required dependency-discovery and resource synchronization issues exceed the scope of this work. So far, our approach has done reasonably well with some application dependencies; for example, the components of distributed applications depend upon one another, and we have had a measure of success in meeting quality requirements for these applications. In the future, however, more work needs to be done and our assumptions must be relaxed to investigate the management of applications with dependencies.

A more complete solution that considers these dependencies requires the synchronization of managed resources across multiple hosts. For example, if process A sends a message to process B, scheduling of resources should be adjusted so that B commences executing with all of the necessary resources just in time to receive and process the message with no delay whatsoever. Doing such a thing for processes on a single host is difficult enough; doing so for applications spread across multiple hosts is even more difficult.

This problem is becoming increasingly difficult as developers try to blur and hide application dependencies to make application development easier; for example, by replacing explicit message passing with remote procedure calls, shared memory, and so on. Nevertheless, support for managing applications with dependencies will become more important as quality demands for our applications become more and more stringent.

### 8.2.9 Policy Based Resource Management Algorithm Selection

Currently, we have rudimentary methods to select the resource management algorithm used to manage a particular application based on its name, owner, start time, and other information registered by the application on initialization. It would be interesting to examine more elegant selection of resource management algorithms based on other factors. These factors could include system load, prior history of algorithm success, and a variety of other conditions.

Having better control over algorithm selection can lead to improved quality of service under a variety of situations. For example, in some scenarios, one resource management algorithm may perform better, but in other cases, another algorithm might be more suitable. Consequently, this issue deserves further attention.

### 8.2.10 Enforcement Policy Refinements

As discussed in Section 6.5, and examined during experimentation, we have provided rule sets for enforcement policies. Of particular note are our policies to support admission control, user hint handling, as well as service differentiation. In the future, we need to investigate and implement other kinds of enforcement policies as well.

In addition to new enforcement policies, however, more work can be done on those already defined. For admission control, additional work can be carried out to help determine good resource levels at which new applications should be blocked from entering the computing environment. Further interesting work can be carried out to investigate various "overbooking" techniques that allow more applications to be admitted than usual, optimistically hoping that they will not all require peak levels of resource allocations at the same time. Other directions for admission control work are possible as well.

In the case of user hints, we can examine more kinds of general user hints, as well as more application-specific types of user hints. As another interesting avenue of research, we should investigate using user hints for predictive purposes and proactive quality of service management, as discussed above. User hints could be collected (both individually and aggregated) and used to develop patterns of behaviour with different applications; these patterns could then be used to help guide resource management decisions.

For example, consider an electronic commerce application in which users interact with a web server to make purchases, browse through catalogues, and so on, through the use of a web browser client. Over time, user behaviour is collected and analysed at the web server to determine user patterns. These patterns can be consulted to determine the typical paths through the web site taken by users so that the appropriate content can be pre-fetched and transmitted to users' web browsers to improve response times when a user appears to be following a given pattern. These patterns could also be used to determine average user think times between requests to the electronic commerce site; during these times, resources from one user session could be safely directed to support other user sessions, and returned just before the think time is complete. In such cases, more users could be supported, without users noticing any degradation in quality, on average.

In addition to more work on admission control and user hints, further work could be done on service differentiation. This includes both methods used to distinguish between user applications for preferential service, and different techniques used to actually deliver this preferential service.

### 8.2.11 Overarching Administrative Policies

Most quality of service policies user in our management system, both expectation and enforcement, are made on a per application or per user basis. As another avenue of future research, it would be interesting to investigate the use of overarching administrative policies dictating how management is to be done across an entire system or distributed computing environment. These policies could be used to specify general goals for the entire managed environment, relying on other underlying policies to be specified or adjusted to support the higher level goal. Such policies could include the following:

- Convergence time for applications should be minimized. This could require the use of particular resource management algorithms, and settings to provide quicker convergence, likely at the cost of increased management overhead.
- Stability for applications should be maximized. Again, this may require the use of a particular resource management algorithm, and specific algorithm and resource manager parameters.

- The distance between delivered and expected quality should be minimized. Once again, this may require particular resource management algorithms, and appropriate parameter settings.

- Computing resource utilization should be maximized. These approaches are intended to get the most out of resources available, but usually result in lower average user satisfaction.

- The number of applications meeting quality of service objectives concurrently should be maximized. Such a policy would likely require a lower overhead resource management approach, and would likely cost in terms of convergence time and stability.

Naturally, many of these general goals would conflict with one another, which opens additional research directions in terms of policy conflict analysis and resolution. Further research in this area is again necessary.

# Appendix A

# Specifying Facts and Rules

This appendix discusses the general syntax used for representing facts and rules in the fact bases and rule bases of QoS Host and Domain Managers. The syntax for both facts and rules has been taken from the CLIPS (C Language Integrated Production System) language [63]. In addition, this appendix provides example facts and rules in this syntax, and describes them in detail.

## A.1 Facts

Simply put, facts in the fact base are represented as tuples of information, with one fact per tuple. Virtually any tuple of information can be a valid CLIPS fact, though it is generally a good idea to place meaningful information in each tuple to ensure it is humanly readable. Every fact has the following general form:

( *field_1 field_2 ... field_n* )

In this case each field is filled in by some appropriate CLIPS symbol, including text, numbers, and so on. Each fact must have at least one field; if there is exactly one field, it must be text of some kind, and not just a number. For additional information on the types of allowed fields in CLIPS, refer to [63].

For example, when a process registers itself with its QoS Host Manager, it supplies registration information to the manager, which asserts this information into the fact base as a fact of the form:

( reginfo *process_id start_time app_name user_id host* )

In this case, italicized tokens are replaced with actual values from the registration process. For a sample fact of this form, suppose that user `bob` executed a stand-alone RTVC video process on the host `farquarson`. This would result in the following fact in the fact base, assuming the process had identifier 1234 and a start time stamp of 12345678:

```
( reginfo 1234 12345678 rtvc_standalone bob farquarson )
```

Suppose that the process registered above violated its quality of service requirements, that it should maintain a video frame rate within a certain range, measured in frames per second (FPS). This results in its coordinator forwarding a violation report to its QoS Host Manager. Facts of the following forms would be asserted into the fact base:

```
( violation_object identifier )
( registered policy_id identifier )
( registered process_id identifier )
( registered user_id user )
( registered host host )
( registered application_name name )
( fps condition current_value target_value)
```

Again, italicized tokens would be filled in by appropriate values. Supposing that quality was to be maintained at 24 frames per second, and that it was actually delivered at 15 frames per second, and allowing for identifiers filled in appropriately, we would have the following facts, as an example:

```
( violation_object 987654321 )
( registered policy_id frame_per_second )
( registered process_id 1234 )
( registered user_id bob )
( registered host farquarson )
( registered application_name rtvc_standalone )
( fps low 15 24 )
```

With these sample facts in place, we can now examine rules that make use of these facts. This will be done in the next section.

## A.2 Rules

Rules in the CLIPS language are specified as a left-hand-side and a right-hand-side. The left-hand-side lists the fact patterns that must be matched for the rule to be activated. Fact patterns can either be complete facts, specifying the contents of each field, or partial facts, where some fields are left as variables to be filled in by the matching process. The right-hand-side lists the actions to be carried out when the rule is activated. The general form for a rule is as follows:

```
(defrule rule_name "optional_comment"
   (pattern_1)
   (pattern_2)
       .
       .
       .
   (pattern_n)
=>
   (action_1)
   (action_2)
       .
       .
       .
   (action_m)
)
```

The special text => is used to split the fact patterns in the left-hand-side from actions in the right-hand-side. All rules must provide a header beginning with the `defrule` token, giving the name of the rule, as well as an optional comment describing the rule. The entire rule must be enclosed within matching parentheses. There are other rule formatting guidelines restrictions; refer to [63] for further details.

As a sample rule, we provide one of the adjustment rules for the metric-heuristic algorithm discussed in Section 5.2. This rule handles metric-heuristic resource adjustments for video applications delivering a frame rate below requirements, as measured in frames per second. (Other rules are provided in cases when the frame rate is too high, and for handling other conditions.) This rule was constructed assuming that facts like those discussed in the previous section are asserted when a quality of service problem is detected and passed on to the QoS Host Manager. This manager invokes the inferencing engine to then apply this rule to the facts asserted into the fact base. This rule is given below.

```
1  (defrule fps_low_local "Handles MH resource adjustments under low FPS."
2      (registered process_id ?pid)
3      (fps low ?current_fps ?target_fps)
4  =>
5      (bind ?amount (resource_distance ?current_fps ?target_fps 10))
6
7      (if (> (probability_condition ?pid rtcpu ?amount) 0.5)
8         then
9         (if (< (possibility_condition ?pid rtcpu ?amount) 0.1)
10           then
11           (assert (service_diff rtcpu))
12           else
13           (adjust_resource ?pid rtcpu ?amount)
14        )
15     )
16
17     (if (> (probability_condition ?pid memory ?amount) 0.5)
18        then
19        (if (< (possibility_condition ?pid memory ?amount) 0.1)
20           then
21           (assert (service_diff memory))
22           else
23           (adjust_resource ?pid memory ?amount)
24        )
25     )
26
27     (set_reset_delay 1)
28 )
```

For this rule to activate, the fact patterns on lines 2 and 3 must be matched. In other words, we require the process identifier of the process that detected the violation, and we require that the violation resulted from a frame rate delivered below its target value. Line 5 then determines the amount of resource allocation required using a function `resource_distance` that we have added to CLIPS, based on the current frame rate, the target frame rate, and a scaling factor selected as 10 in the rule above. Lines 7–15 collects metrics and performs heuristics for the real-time CPU resource. Line 7 uses another function added to CLIPS, `probability_condition`, to determine the probability that allocating the given amount of resources will improve the quality of the process in question. If it is greater than 50%, line 9 then checks to see how possible it is to make such an allocation, given the remaining supply of the resource. This check is done using another function added to CLIPS, possibility_condition. If chances are less than 10and identifies that service differentiation should be applied with respect to this resource by adding a new fact to the fact base; otherwise, the resource adjustment is made using another function added to CLIPS, adjust_resource. Lines 17–25 perform analagous steps for memory resources instead. Finally, line 27 indicates that the QoS Host Manager should not accept further violations from this process for 1 second (to allow for the adjustment to take effect before further complaints are heard).

As an example of a more complex enforcement policy in rule form, we provide a sample service differentiation rule, as mentioned in Section 6.5.3. When a resource has been flagged for service differentiation using the above frame rate rule, the rule below determines which processes to favour, and which ones to hold back:

```
1  (defrule service_differentiation "Handle service differentiation"
2      (service_diff ?resource)
3      (registered process_id ?favoured_pid)
4      (reginfo ?favoured_pid ?timestamp rtvc_standalone ?user ?host)
5      (reginfo ?hindered_pid ?timestamp2 ?application ?user2 ?host)
6  =>
7      (if (> ?timestamp2 ?timestamp)
8        then
9        (set_diff_flag ?favoured_pid favoured ?host)
10       (set_diff_flag ?hindered_pid hindered ?host)
11       (disable_adjustment ?hindered_pid ?resource)
12       (action_directive ?host ?hindered_pid
13        "coordinator->disable_all()" CLIPS_FACT "(disable sensors)")
14     )
15 )
```

Essentially, this rule states that if a stand-alone RTVC video application reports a violation of quality of service requirements, and encounters a shortage of resources in doing so, then all applications started on the same host afterwards should be penalized. In this case, such applications are penalized by losing their allocations of the resource in question, and essentially become best effort applications in that respect.

Line 2 of this rule ensures that this rule is only activated when some resource has been flagged for service differentiation due to a resource shortage. Line 3 obtains the process identifier of the process reporting the quality of service violation. Line 4 matches this to registration information so we can obtain the name of the application, its start time, and the host it is executing on. In this way, we can ensure that this rule is only activated when the process reporting the violation of quality of service requirements is a stand-alone RTVC process. Line 5 is then used to match all other processes executing on the same host. Line 7 is used to filter out all processes that started before or at the same time as the process reporting the violation, so we are left with only those that started afterwards. Line 9 flags the favoured process as such, using the `set_diff_flag` function added to CLIPS, so the QoS Host Manager can restore hindered application processes to their original states when the favoured process terminates. Line 10 flags the application to be hindered as such, and then cancels its resource allocation on line 11, using another function added to CLIPS, `disable_adjustment`. Line 12 instructs the coordinator for the hindered process to disable its sensors to prevent the process from complaining that its resources have been taken away, and that it is in violation of its quality of service requirements. This is done though another function added to CLIPS, `action_directive`.

# Appendix B

# Commonly Used Acronyms

In this appendix, we present expansions of the common acronyms used throughout this thesis. This is done in the table below.

| Acronym | Expansion |
|---|---|
| API | Application Program Interface |
| ATM | Asynchronous Transfer Mode |
| BSD | Berkeley Standard Distribution |
| C-LOOK | Cyclical LOOK |
| C-SCAN | Cyclical SCAN |
| CAS | Centre for Advanced Studies |
| CLIPS | C Language Integrated Production System |
| CPU | Central Processing Unit |
| CORBA | Common Object Request Broker Architecture |
| DB | Data Base |
| DOCSIS | Data Over Cable Service Interface Specification |
| EDF | Earliest Deadline First |
| FCFS | First-Come, First-Serve |
| FPS | Frames Per Second |
| GIF | Graphics Interchange Format |
| GQoS | Generic Quality of Service |
| I/O | Input/Output |
| IBM | International Business Machines |
| IDE | Integrated Drive Electronics |
| IP | Internet Protocol, also known as Internet Protocol, version 4 |
| IPv6 | Internet Protocol, version 6 |

| Acronym | Expansion |
|---|---|
| JPEG | Joint Photographic Experts Group |
| LOOK | Non an acronym—a variation of the SCAN algorithm [208] |
| LRU | Least Recently Used |
| ME | Millennium Edition |
| MH | Metric-Heuristic |
| MPEG | Moving Picture Experts Group |
| MPLS | Multi-Protocol Label Switching |
| MS | Multidimensional Search |
| NP | Nondeterministic Polynomial |
| NSERC | Natural Sciences and Engineering Research Council of Canada |
| NT | New Technology |
| OMT | Object Modeling Technique |
| P | Polynomial |
| PCI | Peripheral Component Interconnect/Interface |
| PGPS | Packet-by-Packet Generalized Processor Sharing |
| POSIX | Portable Operating System Interface for Unix |
| QoS | Quality of Service |
| RCSP | Rate-Controlled Static Priority |
| REQS | Resource-Efficient Quality of Service |
| RM | Rate Monotonic |
| RM | Resource Management |
| RR | Round Robin |
| RSVP | Resource ReSerVation Protocol |
| RTP | Real-Time Protocol |
| RTVC | Real Time Video Compression |
| SCAN | Not an acronym—the name of a disk scheduling algorithm [208] |
| SCED | Service Curve-based Earliest Deadline first |
| SCSI | Small Computer System Interface |
| SSTF | Shortest Seek Time First |
| TCP | Transmission Control Protocol |
| UDB | Universal Database |
| UDP | User Datagram Protocol |
| VOD | Video On Demand |
| WFQ | Weighted Fair Queueing |
| WS | Working Set |
| WSClock | Working Set Clock |
| XP | eXPerience |
| YFQ | Yet another Fair Queueing |

# References

[1] V. Abrossimov and M. Rozier M. Gien. Virtual Memory Management in Chorus. *Progress in Distributed Operating Systems and Distributed Systems Management*, Springer Verlag, Berlin, 1989. (cited on page 33)

[2] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. *Proceedings of the 1986 Summer USENIX Conference*, Atlanta, Georgia, Summer 1986. (cited on page 33)

[3] H. Akhand. A Mechanism to Implemented Differentiated Services in IP-Based Network. Master's thesis, The University of Western Ontario, January 2001. (cited on pages 119, 249)

[4] M. Alfano. Design and Implementation of a Cooperative Multimedia Environment with QoS Control. *Computer Communications (21), Elsevier Science*, Fall 1997. (cited on pages 31, 32, 40)

[5] M. Alfano and N. Radouniklis. A Cooperative Multimedia Enviroment with QoS Control: Architectural and Implementation Issues. *Technical Report TR-96-040, International Computer Science Institute, Berkeley, California*, September 1996. (cited on page 24)

[6] E. Amir, S. McCanne, and R. Katz. Receiver-driven Bandwidth Adaptation for Lightweight Sessions. *Proceedings of the Fifth ACM International Conference on Multimedia*, Seattle, Washington, November 1997. (cited on page 32)

[7] C. Aurrecoechea, A. Campbell, and L. Hauw. A Survey of QoS Architectures. *Multimedia Systems Journal, Special Issue on QoS Architecture*, May 1998. (cited on page 23)

[8] V. Baiceanu, C. Cowan, D. McNamee, C. Pu, and J. Walpole. Multimedia Application Require Adaptive CPU Scheduling. *Proceedings of the 1996 Workshop on Resource Allocation Problems in Multimedia Systems*, Washington DC, December 1996. (cited on page 38)

[9] G. Banga, P. Druschel, and J. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*, New Orleans, Louisiana, February 1999. (cited on page 28)

[10] L. Bannon, A. Cypher, S. Greenspan, and M. Monty. Evaluation and Analysis of Users' Activity Organization. *Proceedings of the 1983 ACM Conference on Human Factors in Computing Systems*, Boston, Massachusetts, December 1983. (cited on page 110)

[11] M. Barabanov and V. Yodaiken. Introducting Real-Time Linux. *The Linux Journal, Issue #34*, February 1997. (cited on page 22)

[12] A. Basso, İ. Dalgiç, F. Tobagi, and C. van den Branden Lambrecht. Study of MPEG-2 Encoding Performance Based on a Perceptual Quality Metric. *Proceedings of the 1996 Picture Coding Symposium*, Melbourne, Australia, July 1996. (cited on page 31)

[13] G. Beaton. A Feedback-Based Quality of Service Management Scheme. *Proceedings of the Third International Workshop on High Performance Protocol Architectures*, Uppsala, Sweden, June 1997. (cited on pages 23, 38–41)

[14] J. Bentley. Multidimensional Binary Search Trees in Database Applications. *IEEE Transactions on Software Engineering, Volume SE-5, Number 4*, July 1979. (cited on pages 89, 95)

[15] J. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM, Volume 18, Number 9*, September 1975. (cited on pages 89, 95)

[16] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, December 1995. (cited on page 33)

[17] M. Biddiscombe, J. Midwinter, and S. Sabesan. Application of Free-Market Principles to Telecoms Resource Allocation. *Electronics Letters, Vol. 35, No. 4*, February 1999. (cited on page 30)

[18] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. *IETF RFC 2475*, December 1998. (cited on pages 35, 119)

[19] P. Bodorik. Quality of Service throught Middleware that Accesses Data Stores (DBs). *Presented at the 2000 Centre for Advanced Studies Confererence*, Toronto, Canada, November 2000. (cited on pages 3, 40, 43)

[20] W. Bolosky, R. Draves, R. Fitzgerald, C. Fraser, M. Jones, T. Knoblock, and R. Rashid. Operating System Directions for the Next Millennium. *Microsoft Research Article*, September 2001. (cited on pages 22, 39, 41)

[21] A. Bondavalli, F. Di Giandomenico, and I. Mura. Value-Driven Resource Assignment in Object-Oriented Real-Time Dependable Systems. *Proceedings of the Third International Workshop on Object-oriented Real-time Dependable Systems*, Newport Beach, California, February 1997. (cited on page 42)

[22] K. Brown, M. Mehta, M. Carey, and M. Livny. Towards Automated Performance Tuning for Complex Workloads. *Proceedings of the 20th International VLDB Conference*, 1994. (cited on page 29)

[23] J. Bruno, J. Brustoloni, E. Gabber, M. McShea, M. Ozden, and A. Silberschatz. Disk Scheduling with Quality of Service Guarantees. *Proceedings of the 1999 IEEE International Conference Multimedia Computing and Systems*, Florence, Italy, June 1999. (cited on pages 19, 37)

[24] J. Bruno, J. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Retrofitting Quality of Service into a Time-Sharing Operating System. *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, California, June 1999. (cited on page 28)

[25] A. Campbell, G. Coulson, and D. Hutchison. A Multimedia Enhanced Transport Service in a Quality of Service Architecture. *Proceedings of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, Lancaster, England, November 1993. (cited on pages 23, 36)

[26] R. Carr and J. Hennessy. WSClock – A Simple and Effective Algorithm for Virtual Memory Management. *Proceedings of the Eight Symposium on Operating Systems Principles*, Pacific Grove, California, December 1981. (cited on page 19)

[27] F. Chang and G. Gibson. Automatic I/O Hint Generation through Speculative Execution. *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, Louisiana, February 1999. (cited on page 29)

[28] S. Chatterjee, J. Sydir, B. Sabata, and T. Lawrence. Modeling Applications for Adaptive QoS-based Resource Management. *Proceedings of the Second IEEE High Assurance Systems Engineering Workshop*, Bethesda, Maryland, August 1997. (cited on page 30)

[29] L. Chen, K. Li, E. Manning, and S. Khan. Building an Adaptive Multimedia System Using the Utility Model. *Proceedings of the 1999 International Workshop on Parallel and Distributed Realtime Systems*, San Juan, Peurto Rico, April 1999. (cited on page 30)

[30] S. Chen and M. Thapar. I/O Channel and Real-Time Disk Scheduling for Video Servers. *Hewlett Packard Labs Technical Report HPL-96-126*, August 1996. (cited on page 37)

[31] D. Cheriton. The V Distributed System. *Communications of the ACM, 31(3)*, March 1988. (cited on page 33)

[32] H. Cho and A. Seneviratne. Dynamic QoS Control without the Knowledge of Resource Requirements. *Submitted to IEEE Trans. Computing.* (cited on pages 25, 39–41)

[33] H. Chu and K. Nahrstedt. A Soft Real Time Scheduling Server in UNIX Operating System. *European Workshop on Interactive Distributed Multimedia Systems and*

*Telecommunication Services*, Darmstadt, Germany, September 1997. (cited on pages 3, 20, 34, 122)

[34] H. Chu and K. Nahrstedt. Memory Management for Soft Real-Time Multimedia Applications. *Technical Report, University of Illinois at Urbana Champaign*, October 1997. (cited on page 34)

[35] R. Conway, W. Maxwell, and L. Miller. *Theory of Scheduling.* Addison-Wesley Publishing Company, 1967. (cited on page 19)

[36] F. Corbato. A Paging Experiment with the Multics System. *MIT Project MAC Report MAC-M-384*, May 1968. (cited on page 19)

[37] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 1992. (cited on page 55)

[38] C. Cranor and G. Parulkar. Design of Universal Continuous Media I/O. *Lecture Notes in Computer Science Series, Volume 1018, Springer-Verlag*, 1995. (cited on page 37)

[39] B. Davison and H. Hirsh. Predicting Sequences of User Actions. *Proceedings of the AAAI/ICML 1998 Workshop on Predicting the Future: AI Approaches to Time-Series Analysis*, New Brunswick, New Jersey, July 1998. (cited on page 32)

[40] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. *Internet RFC 1883*, December 1995. (cited on page 35)

[41] L. Delgrossi, R. Herrtwich, and F. Hoffmann. An Implementation of ST-II for the Heidelberg Transport System. *Proceedings of the 1992 IEEE Globcom Conference*, Orlando, Florida, 1992. (cited on pages 23, 36)

[42] P. Denning. The Working Set Model for Program Behaviour. *Communications of the ACM, Volume 11, Number 5*, May 1968. (cited on pages 19, 129)

[43] R. Dianda. *Addition of the ITU Modify Capability to UNI and PNNI.* ATM Forum Contribution 97-0238, Mountain View, California, April 1997. (cited on page 36)

[44] R. Dillon. A User-Centred Approach to QoS Requirements. *Presented at the 1999 Centre for Advanced Studies Confererence*, Toronto, Canada, November 1999. (cited on pages 3, 4, 31, 39)

[45] C. Diot, C. Huitema, and T. Turletti. Multimedia Applications Should be Adaptive. *Proceedings of the Thrid IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, Mystic, Connecticut, August 1995. (cited on page 38)

[46] D.Waddington and D.Hutchison. A General Model for QoS Adaptation. *Proceedings of the 6th International Workshop on Quality of Service*, Napa, California, May 1998. (cited on page 27)

[47] D.Waddington and D.Hutchison. End-to-end QoS Provisioning Through Resource Adaptation . *Proceedings of 8th IFIP Conference on High Performance Networking*, Vienna, Austria, September 1998. (cited on page 27)

[48] M. Easton and P. Franaszek. Use Bit Scanning in Replacement Decisions. *RC-6192, IBM Research, Yorktown Heights, New York*, September 1976. (cited on pages 19, 128)

[49] J. Edwards and P. Cao. User-Oriented Resource Scheduling in UNIX. *Technical Report, University of Wisconsin, Department of Computer Science, CS-TR-96-1318*, 1996. (cited on pages 22, 35, 37)

[50] D. Engler, M. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995. (cited on page 33)

[51] M. Fabbi. Bandwidth Throwing – Yesterday's LAN Design Method. *Gartner Group Research Note*, December 1997. (cited on page 18)

[52] A. Fait and Z. Rosen. Experimental Studies of Access Graph Based Heuristics: Beating the LRU Standard? *Proceedings of the 1997 SIAM Symposium on Discrete Algorithms*, January 1997. (cited on page 19)

[53] D. F. Ferguson, C. Nikolaou, J. Sairamesh, and Y. Yemini. Economic Models for Allocating Resources in Computer Systems. *Market Based Control of Distributed Systems*, Editor Scott Clearwater, World Scientific Press, 1996. (cited on page 30)

[54] D. Ferrari, A. Banerjea, and H. Zhang. Network Support for Multimedia. *Technical Report TR-92-072, The International Computer Science Institute, Berkeley, California*, November 1992. (cited on pages 23, 36)

[55] G. Fine, P. McIsaac, and C. Jackson. Dynamic Program Behaviour Under Paging. *Proceedings of the ACM 21st National Conference*, May 1966. (cited on page 19)

[56] P. Florissi, Y. Yemini, and D. Florissi. QoSockets: A New Extension to the Sockets API for End-to-End Application QoS Management. *Proceedings of the Sixth IFIP/IEEE International Symposium on Integrated Network Management*, Boston, Massachusetts, May, 1999. (cited on page 36)

[57] K. Gaylin. How are Windows Used? Some Notes on Creating an Empirically-Based Windowing Benchmark Task. *Proceedings of the 1986 ACM Conference on Human Factors in Computing Systems*, Boston, Massachusetts, April 1986. (cited on page 110)

[58] C. Gbaguidi, O. Verscheure, and J. P. Hubaux. A New Flexible and Modular QoS Mapping Framework based on Psychophysics. *Proceedings of the 1997 Management of Multimedia Networks and Services Conference*, Montréal, Canada, July 1997. (cited on page 31)

[59] N. D. Georganas. Multimedia: Enabling Technologies and Applications. *Proceedings of SIMTEL'96*, Vina del Mar, Chile, May 1996. (cited on page 1)

[60] L. Georgiadis, C. Nikolaou, and I. Viniotis. Adaptive Scheduling Algorithms that Satisfy Average Response Time Objectives. *IBM Research Report RC 14851 (#66555)*, August 1989. (cited on pages 3, 29)

[61] M. Gervasio, W. Iba, and P. Langley. Learning User Evaluation Functions for Adaptive Scheduling Assistance. *Proceedings of the Sixteenth International Conference on Machine Learning*, Bled, Slovenia, June 1999. (cited on page 32)

[62] M. Gervasio, W. Iba, and P. Langley. Learning to Predict User Operations for Adaptive Scheduling. *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, Madison, Wisconsin, July 1998. (cited on page 32)

[63] J. Giarratano and G. Riley. *Expert Systems: Principles and Programming*. PWS Publishing Company, 1998. (cited on pages 116, 255, 257)

[64] M. Gibbs. Doing More With What You Have. *Network World (Volume 18, Number 45)*, November 5, 2001. (cited on page 18)

[65] G. Gibson, J. Vitter, and J. Wilkes. Report of the Working Group on Storage I/O for Large-Scale Computing. *ACM Computing Surveys, 28 (4)*, December 1996. (cited on page 36)

[66] A. Goel, D. Steere, C. Pu, and J. Walpole. Adaptive Resource Management Via Modular Feedback Control. *Submitted to the Seventh Workshop on Hot Topics in Operating Systems*, Rio Rico, Arizona, March 1999. (cited on page 27)

[67] R. Gopalakrishnan. Efficient Quality of Service Support Within Endsystems for High Speed Multimedia Networking. *PhD Thesis, Department of Computer Science, Washington University*, December 1996. (cited on page 33)

[68] P. Goyal, X. Guo, and H. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996. (cited on page 33)

[69] K. Harty and D. Cheriton. Application-Controlled Physical Memory Using External Page-Cache Management. *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, April 1991. (cited on page 33)

[70] J. L. Hellerstein, F. Zhang, and P. Shahabuddin. An Approach to Predictive Detection for Service Management. *Presented at the Sixth IFIP/IEEE International Symposium on Integrated Network Management*, Boston, Massachusetts, May 1999. (cited on pages 36, 251)

[71] J. W. Hong, G. W. Gee, and M. A. Bauer. Towards Automating Instrumentation of Systems and Applications for Management. *Proceedings of the 1995 IEEE Global Telecommunications Conference*, Singapore, November 1995. (cited on page 251)

[72] C. Hood and C. Ji. Automated Proactive Anomaly Detection. *Proceedings of the Fifth International Symposium on Integrated Network Management*, May 1997. (cited on pages 36, 251)

[73] E. Horvitz. Thinking Ahead: Continual Computation Policies for Allocating Idle and Real-Time Resources to Solve Future Challenges. *Proceedings of the Sixteenth Joint Conference on Artificial Intelligence*, Stockholm, Sweden, July 1999. (cited on page 32)

[74] E. Horvitz, J. Breese, D. Heckerman, D. Hovel, and K. Rommelse. The Lumiere Project: Bayesian User Modeling for Inferring the Goals and Needs of Software Users. *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, Madison, Wisconsin, July 1998. (cited on page 32)

[75] E. Horvitz, A. Jacobs, and D. Hovel. Attention-Sensitive Alerting. *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, Stockholm, Sweden, July 1999. (cited on page 32)

[76] S. Howard. *Policy-Driven Management for Distributed Object Systems.* PhD thesis, The University of Western Ontario, August 1999. (cited on page 63)

[77] M. Hubel. Drive DB2 Faster with Database-GUYS. *Presented at the 2000 Centre for Advanced Studies Confererence*, Toronto, Canada, November 2000. (cited on page 42)

[78] Id Software Inc. *DOOM.* December 1997. (cited on page 251)

[79] ISO. *Quality of Service Framework.* International Organization for Standardization ISO/IEC JTC1/SC21/WG1 N9680, 1995. (cited on page 23)

[80] ITU-R Recommendations BT.500-8. *Methodology for the Subjective Assessment of the Quality of Television Pictures.* February 1998. (cited on pages 8, 31)

[81] D. Ivan-Roşu and K. Schwan. Improving Protocol Performance by Dynamic Control of Communication Resources. *Technical Report GIT-CC-96-04, Georgia Institute of Technology*, February 1996. (cited on page 42)

[82] E. Jackson, W. Vandeputte, and C. Cappuccio. The OpenBSD FAQ. *Available from http://www.openbsd.org*, June 2000. (cited on page 22)

[83] R. Jain. *The Art of Computer Systems Performance Analysis.* Wiley Professional Computing, 1991. (cited on pages 8, 90, 100, 149)

[84] K. Jeffay and D. Bennett. A Rate-Based Execution Abstraction for Multimedia Computing. *Lecture Notes in Computer Science 1018:64-78*, 1995. (cited on pages 33, 34)

[85] M. Jones, J. Barrera III, A. Forin, P. Leach, D. Roşu, and M. Roşu. An Overview of the Rialto Real-Time Architecture. *Proceedings of the Seventh ACM SIGOPS European Workshop*, September 1996. (cited on page 33)

[86] M. Jones, D. Roşu, and M. Roşu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. *Proceedings of the Sixteenth ACM Symposium on Operation Systems Principles*, St. Malo, France, October 1997. (cited on pages 33, 42)

[87] J. Kamada, M. Yuhara, and E. Ono. User-level Realtime Scheduler Exploiting Kernel-level Fixed Priority Scheduler. *Multimedia Japan*, March 1996. (cited on pages 20, 34, 122)

[88] E. Kandogan and B. Shneiderman. Elastic Windows: Evaluation of Multi-Window Operations. *Proceedings of the 1997 ACM Conference on Human Factors in Computing Systems*, Atlanta, Georgia, March 1997. (cited on page 110)

[89] M. J. Katchabaw, H. L. Lutfiyya, and M. A. Bauer. Driving Resource Management With Application-Level Quality of Service Specifications. *Proceedings of First International Conference on Information and Computation Economies*, Charleston, South Carolina, October 1998. (cited on page 151)

[90] M. J. Katchabaw, H. L. Lutfiyya, and M. A. Bauer. Using User Hints to Guide Resource Management for Quality of Service. *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, June 1999. (cited on page 216)

[91] M. J. Katchabaw, H. L. Lutfiyya, and M. A. Bauer. A Model for Resource Management to Support End-to-End Application-Driven Quality of Service. *Proceedings of the Tenth International Conference on Parallel and Distributed Computing Systems*, Las Vegas, Nevada, October 1998. (cited on page 151)

[92] M. J. Katchabaw, H. L. Lutfiyya, and M. A. Bauer. A Survey of Windowing Environment Usage Patterns. *Techinical Report #540, Department of Computer Science, The University of Western Ontario*, May 1999. (cited on page 110)

[93] M. J. Katchabaw, H. L. Lutfiyya, and M. A. Bauer. A Quality of Service Management Testbed. *Proceedings of the Third IEEE International Workshop on Systems Management*, Newport, Rhode Island, April 1998. (cited on pages 143, 151)

[94] M. J. Katchabaw, H. L. Lutfiyya, and M. A. Bauer. Memory Management for Quality of Service Provisioning. *Techinical Report #546, Department of Computer Science, The University of Western Ontario*, November 1999. (cited on page 151)

[95] R. Kath. The Virtual-Memory Manager in Windows NT. *Microsoft Developer Network Library Technical Article*, December 1992. (cited on page 21)

[96] S. Kaushal. Automating the Instrumentation in Distributed Applications. Master's thesis, The University of Western Ontario, December 1998. (cited on page 251)

[97] K. Kawachiya, , and H. Tokuda. Dynamic QoS Control Based on the QOS-Ticket Model. *Proceedings of the 1996 IEEE International Conference on Multimedia Computing and Systems*, Hiroshima, Japan, June 1996. (cited on page 25)

[98] K. Kawachiya, , and H. Tokuda. QOS-Ticket: A New Resource-Management Mechanism for Dynamic QoS Control of Multimedia. *Proceedings of IPSJ Multimedia Japan '96*, Yokohama, Japan, March 1996. (cited on page 25)

[99] S. Khan, K. Li, and E. Manning. The Utility Model for Adaptive Multimedia Systems. *Proceedings of the 1997 International Conference on Multimedia Modeling*, Singapore, November 1997. (cited on page 30)

[100] B. Korvemaker and R. Greiner. Predicting Unix Command Lines: Adjusting to User Patterns. *Proceedings of the 17th National Conference on Artificial Intelligence*, Austin, Texas, July 2000. (cited on page 32)

[101] K. Lakshman, R. Yavatkar, and R. Finkel. Integrated CPU and Network-I/O QoS Management in an Endsystem. *Computer Communications Special Issue on Guiding Quality of Service into Distributed Systems*, August 1997. (cited on page 26)

[102] S. Lalis, C. Nikolaou, D. Papdakis, and M. Marazakis. Market-driven Service Allocation in a QoS-capable Environment. *Proceedings of First International Conference on Information and Computation Economies*, Charleston, South Carolina, October 1998. (cited on page 30)

[103] B. A. LaMacchia. Basis reduction algorithms and subset sum problems. Master's thesis, Massachusetts Institute of Technology, May, 1991. (cited on page 55)

[104] B. Landfeldt, A. Seneviratne, and C. Diot. User Services Assistant: An End-to-End Reactive QoS Architecture. *Proceedings of the Sixth International Workshop on Quality of Service*, Napa, California, May 1998. (cited on pages 24, 25, 39–41)

[105] A. A. Lazar, S. Bhonsle, and K. Lim. A Binding Architecture for Multimedia Networks. *Proceedings of the 1994 Conference on Multimedia Transport and Teleservices*, Vienna, Austria, November 1994. (cited on page 23)

[106] C. Lee. *On Quality of Service Management.* PhD thesis, Carnegie Mellon University, August 1999. (cited on page 30)

[107] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen. A scalable solution to the multi-resource qos problem. *Proceedings of the 20th IEEE Real-Time Systems Symposium*, Scottsdale, Arizona, December 1999. (cited on page 30)

[108] C. Lee, R. Rajkumar, and C. Mercer. Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach. *Proceedings of Multimedia Japan*, March 1996. (cited on pages 33, 34)

[109] S. Leffler, W. Joy, and M. McKusick. Unix Programmer's Manual. *Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley*, 1983. (cited on page 36)

[110] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support

Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, June 1997. (cited on page 33)

[111] B. Li. *Agilos: A Middleware Control Architecture for Application-Aware Quality of Service Adaptations.* PhD thesis, University of Illinois at Urbana Champaign, May 2000. (cited on page 28)

[112] B. Li and K. Nahrstedt. Impact of Control Theory on QoS Adaptation in Distributed Middleware Systems. *Proceedings of the 2001 American Control Congference*, Arlington, Virginia, June 2001. (cited on page 28)

[113] B. Li and K. Nahrstedt. A Control Theoretical Model for Quality of Service Adaptations. *Proceedings of the Sixth International Workshop on Quality of Service*, Napa California, May 1998. (cited on pages 28, 39–42)

[114] B. Li, D. Xu, and K. Nahrstedt. Optimal State Prediction for Feedback-Based QoS Adaptations. *Proceedings of the Seventh International Workshop on Quality of Service*, London, England, May 1999. (cited on page 28)

[115] C. Lin, H. Chu, and K. Nahrstedt. A Soft Real Time Scheduling Server on the Windows NT. *Proceedings of the Second USENIX Windows NT Symposium*, Seattle, Washington, August 1998. (cited on pages 21, 34)

[116] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real Time Envioronment. *Journal of the ACM, 20(1)*, 1973. (cited on page 19)

[117] J. Loyall, D. Bakken, R. Schantz, J. Zinky, D. Karr, R. Vanegas, and K. Anderson. QoS Aspect Languages and Their Runtime Integration. *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Pittsburgh, Pennsylvania, May 1998. (cited on page 26)

[118] H. L. Lutfiyya, G. P. Molenkamp, M. J. Katchabaw, and M. A. Bauer. Issues in Managing Soft QoS Requirements in Distributed Systems Using a Policy-Based Framework. *Proceedings of the International Workshop on Policies*, Bristol, England, January 2001. (cited on page 151)

[119] J. MacKie-Mason and H. Varian. Pricing Congestible Network Resources. *IEEE Journal on Selected Areas in Communications, 13 (7)*, 1995. (cited on page 30)

[120] U. Manber. *Introduction to Algorithms: A Creative Approach.* Addison-Wesley, Reading, Massachusetts, 1989. (cited on pages 54, 96)

[121] E. Manning. Optimal Internet Admission Controls For Multimedia: The Best Way to Design an Admission Controller. *Invited Talk to the Department of Computer Science at the University of Western Ontario*, October 12, 2001. (cited on page 30)

[122] P. Martin. Managing Database Server Performance within an Electronic Commerce Framework. *Presented at the 1999 International Symposium on Electronic Commerce*, Beijing, China, May 1999. (cited on pages 29, 76, 249)

[123] P. Martin. Self-Managing DBMSs (A Researcher's View). *Presented at the 2000 Centre for Advanced Studies Confererence*, Toronto, Canada, November 2000. (cited on pages 29, 249)

[124] H. Massalin and C. Pu. Fine-Grain Adaptive Scheduling Using Feedback. *Computing Systems 3(1)*, Winter 1990. (cited on pages 27, 42)

[125] J. Mauro and R. McDougall. *Solaris Internals: Core Kernel Architecture*. Sun Microsystemd Press, 2001. (cited on pages 20, 120, 121, 125, 128)

[126] S. McCanne and V. Jacobson. LBNL Audio Conferencing Tool (vat). *On-line Documentation from http://www-nrg.ee.lbl.gov/vat*, December 1996. (cited on page 32)

[127] S. McCanne and V. Jacobson. vic: A Flexible Framework for Packet Video. *Proceedings of ACM Multimedia '95*, November 1995. (cited on page 32)

[128] D. McNamee and K. Armstrong. Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies. *Proceedings of the USENIX Mach Symposium*, Burlington, Vermont, October 1990. (cited on page 33)

[129] C. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, Boston, Massachusetts, May 1994. (cited on page 33)

[130] Microsoft Corporation. Windows Sockets 2 Application Programming Interface. *Microsoft White Paper*, May, 1996. (cited on page 35)

[131] Microsoft Corporation. The Microsoft QoS Components. *Microsoft White Paper*, November, 1999. (cited on page 21)

[132] G. Molenkamp. Quality of Service Violation Detection and Location in Distributed Applications. Master's thesis, The University of Western Ontario, September 2000. (cited on pages 70, 71, 74, 77, 107, 115, 150)

[133] G. P. Molenkamp, M. J. Katchabaw, H. L. Lutfiyya, and M. A. Bauer. Distributed Resource Management to Support Distributed Application-Specific Quality of Service. *Proceedings of the Fourth IFIP/IEEE International Conference on the Management of Multimedia Netowkrs and Services*, Chicago, Illinois, October 2001. (cited on page 222)

[134] G. P. Molenkamp, M. J. Katchabaw, H. L. Lutfiyya, and M. A. Bauer. Managing Soft QoS Requirements in Distributed Systems. *Proceedings of the Multimedia Systems Workshop (ICPP)*, Toronto, Canada, August 2000. (cited on page 151)

[135] H. Motoda, T. Washio, T. Kayama, and K. Yoshida. Extracting Behavioral Patterns from Relational History Data. *Proceedings of the Sixth International Conference on User Modeling*, Chia Laguna, Sardinia, June 1997. (cited on page 32)

[136] R. Muntz and J. Pasquale. System-level Design Issues for Storage I/O, Strategic Directions in Computing Research: Working Group on Storage I/O for Large-Scale Computing. *ACM Computing Surveys, 28A (4)*, December 1996. (cited on page 37)

[137] R. Nagarajan, J. Kurose, and D. Towsley. Allocation of Local Quality of Service Constraints to meet End-to-End Requirements. *Proceedings of the IFIP Workshop on the Performance Analysis of ATM Systems*, Martinique, January 1993. (cited on page 42)

[138] K. Nahrstedt, H. Chu, and S. Narayan. QoS-aware Resource Management for Distributed Multimedia Applications. *Journal on High-Speed Networking, Special Issue on Multimedia Networking, IOS Press, Vol. 8, No. 3-4*, 1998. (cited on page 25)

[139] K. Nahrstedt and J. Smith. An Experimental Investigation of Issues in End-to-End QoS. *Technical Report MS-CIS-94-08, DSL 76, University of Pennsylvania*, Februrary 1994. (cited on page 39)

[140] K. Nahrstedt and J. Smith. The QoS Broker. *Technical Report MS-CIS-94-13, University of Pennsylvania*, March 1994. (cited on pages 23, 36)

[141] K. Nahrstedt and J. Smith. Design, Implementation and Experiences of the OMEGA End-Point Architecture. *Technical Report MS-CIS-95-22, University of Pennsylvania*, May 1995. (cited on pages 23, 25)

[142] J. Nieh, J. G. Hanko, J. D. Northcut, and G. A. Wall. SVR4UNIX Scheduler Unacceptable for Multimedia Applications. *Proceedings of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, Lancaster, England, November 1993. (cited on page 109)

[143] J. Nieh and M. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. *Proceedings of the Sixteenth ACM Symposium on Operation Systems Principles*, St. Malo, France, October 1997. (cited on page 33)

[144] C. Nikolaou, D. Ferguson, and P. Constantopoulos. Towards Goal Oriented Resource Management. *First Workshop on Load Balancing on High Performance Parallel and Distributed Systems*, June 2, 1995. (cited on pages 29, 39)

[145] N. Nishio and H. Tokuda. QOS Translation and Session Coordination Technique for Multimedia Systems. *Proceedings of the Sixth International Workshop on Network and Operating System Support for Digital Audio and Video*, Zushi, Japan, April 1996. (cited on page 24)

[146] R. Onvural and R. Cherukuri. *Signaling in ATM Networks.* Artech House, Boston, Massachusetts, 1997. (cited on page 36)

[147] M. Ott, G. Michelitsch, D. Reininger, and G. Welling. An Architecture for Adaptive QoS and its Application to Multimedia Systems Design. *Computer Communications Special Issue on Guiding Quality of Service into Distributed Systems*, August 1997. (cited on pages 26, 32)

[148] P. Pal, J. Loyall, R. Schantz, J. Zinky, R. Shapiro, and J. Megquier. Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration. *Proceedings of the Third IEEE International Symposium on Object-oriented Real-time Distributed Computing*, Newport Beach, California, March 1999. (cited on page 26)

[149] S. Parekh, N. Gandhi, J. L. Hellerstein, D. Tilbury, T. S. Jayram, and J. Bigus. Using Control Theory to Achieve Service Level Objectives in Performance Management. *Submitted to the Seventh IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, Washington, May 2001. (cited on pages 28, 42)

[150] K. Park, M. Sitharam, and S. Chen. Quality of Service Provision in Noncooperative Networks: Heterogenous Preferences, Multidimensional QoS Vectors, and Burstiness. *Proceedings of First International Conference on Information and Computation Economies*, Charleston, South Carolina, October 1998. (cited on page 30)

[151] J. Pasquale, E. Anderson, and K. Muller. Container Shipping: Operating System Support for I/O-Intensive Applications. *IEEE Computer, 27 (3)*, March 1994. (cited on page 37)

[152] K. Patel, B. Smith, and L. Rowe. Performance of a Software MPEG Video Decoder. *Proceedings of the 1993 ACM Multimedia Conference*, Anaheim, California, August 1993. (cited on page 148)

[153] D. Patterson and J. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 1996. (cited on page 34)

[154] D. Petrou, J. Milford, and G. Gibson. Implementing Lottery Scheduling: Matching the Specializations in Traditional Schedulers. *Proceedings of USENIX 1999*, Monterey, California, June 1999. (cited on page 33)

[155] T. Plagemann, A. Gotti, and B. Plattner. CoRA - A Heuristic for Protocol Configuration and Resource Allocation. *Proceedings of the 1994 IFIP Workshop on Protocols for High-Speed Networks*, Vancouver, Canada, August 1994. (cited on page 24)

[156] M. De Prycker. *Asynchronous Transfer Mode.* Ellis Horwood, New York, 1996. (cited on page 36)

[157] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for qos management. *Proceedings of the 18th IEEE Real-Time Systems Symposium*, San Fransisco, California, December 1997. (cited on page 30)

[158] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, Chicago, Illinois, July 1998. (cited on page 25)

[159] S. Rampal, D. Reeves, and I. Viniotis. Dynamic Resource Management Based on Measured QoS. *Technical Report TR 96-2, North Carolia State University*, 1996. (cited on pages 19, 38)

[160] D. Reininger, D. Raychaudhuri, and M. Ott. Market Based Bandwidth Allocation Policies for QoS Control in Broadband Networks. *Proceedings of First International Conference on Information and Computation Economies*, Charleston, South Carolina, October 1998. (cited on page 31)

[161] L. Reiss and J. Radin. *X Window Inside and Out.* McGraw Hill Incorporated, 1992. (cited on page 111)

[162] J. Richter. *Advanced Windows, Third Edition.* Microsoft Press, 1997. (cited on page 21)

[163] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Chen, and B. Bershad. Instrumentation and Optimization of Win32/Intel Executables Using Etch. *Proceedings of the 1997 USENIX Windows NT Workshop*, Seattle, Washington, August 1997. (cited on page 29)

[164] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. *IETF RFC 3031*, January 2001. (cited on page 35)

[165] M. Rozier, V. Abrossimov, F. Armand, J. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus Distributed Operating System. *Technical Report CS/TR-90-25.1, Chorus Systemèmes*, France, 1990. (cited on page 33)

[166] C. Ruemmler and J. Wilkes. Disk Shuffling. *Hewlett Packard Labs Technical Report HPL-91-156*, October 1991. (cited on page 37)

[167] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991. (cited on page 105)

[168] H. Sariowan, R. Cruz, and G. Polyzos. Scheduling for Quality of Service Guarantees via Service Curves. *Proceedings of the 1995 International Conference on Computer Communications and Networks*, Las Vegas, Nevada, September 1995. (cited on page 19)

[169] R. Schantz, J. Zinky, D. Karr, D. Bakken, J. Megquier, and J. Loyall. An Object-level Gateway Supporting Integrated-Property Quality of Service. *Proceedings of the Second IEEE International Symposium on Object-oriented Real-time Distributed Computing*, Saint Malo, France, May 1999. (cited on page 26)

[170] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, 1986. (cited on page 55)

[171] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. *Internet RFC 1889*, January 1996. (cited on page 35)

[172] M. Seltzer, P. Chen, and J. Ousterhous. Disk Scheduling Revisited. *Proceedings of the 1990 Winter Usenix Conference*, Washington, D.C., January 1990. (cited on page 37)

[173] C. Shen, O. Gonzaález, K. Ramamritham, and I. Mizunuma. User Level Scheduling of Communicating Real-Time Tasks. *Proceedings of the Fifth IEEE Conference on Real-Time Technology and Applications*, Vancouver, Canada, June 1999. (cited on page 34)

[174] P. Shenoy and H. Vin. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. *Proceedings of International Conference on Measurement and Modeling of Computer Systems*, Madison, Wisconsin, June 1998. (cited on page 37)

[175] A. Silberschatz and P. Galvin. *Operating System Concepts, Fifth Edition.* John Wiley and Sons, Inc., 1998. (cited on page 19)

[176] T. So. *Bandwidth Change Indication Proposal.* ATM Forum Contribution 97-0561, Mountain View, California, 1997. (cited on page 36)

[177] Startdust.com. Introduction to qos policies. Technical report, Stardust.com, Inc., July 1999. (cited on page 63)

[178] D. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, Louisiana, February 1999. (cited on page 27)

[179] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. *Proceedings of the Real-Time Systems Symposium*, Washington, DC, December 1996. (cited on pages 19, 33)

[180] D. Stokes. Availability and Performance Management in Distributed Systems. Master's thesis, The University of Western Ontario, September 1995. (cited on page 64)

[181] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staelin. An Economic Paradigm for Query Processing and Data Migration in Mariposa. *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, Austin, Texas, September 1994. (cited on page 30)

[182] Sun Microsystems Inc. *Multithreading and Real-Time in Solaris: A White Paper.* Mountain View, California, 1992. (cited on page 20)

[183] Sun Microsystems Inc. madvise – Provide Advice To VM System. *Solaris 2.6 Reference Manual Answerbook*, Mountain View, California, 1996. (cited on page 20)

[184] Sun Microsystems Inc. priocntl – display or set scheduling parameters of specified process(es). *Solaris 2.6 Reference Manual Answerbook*, Mountain View, California, 1996. (cited on page 120)

[185] Sun Microsystems Inc. Solaris Bandwidth Manager 1.6 Developers Guide. *Solaris Bandwidth Manager 1.6 Answerbook*, Mountain View, California, 2000. (cited on page 20)

[186] Sun Microsystems Inc. Solaris Bandwidth Manager 1.6 System Administration Guide. *Solaris Bandwidth Manager 1.6 Answerbook*, Mountain View, California, 2000. (cited on page 20)

[187] Sun Microsystems Inc. *SunVideo User's Guide.* Mountain View, California, August 1994. (cited on page 148)

[188] Sun Microsystems Inc. *The Complete Guide to ShowMe 2.0.1.* Mountain View, California, May 1994. (cited on page 32)

[189] V. Sunkad. Quality-of-Service: A DOCSIS/PacketCable Perspective. *Specs News & Technology From CableLabs, 12 (3)*, April 2000. (cited on page 36)

[190] S. Taylor. Predicting Violations of QoS Levels in Distributed Computing Environments (Tentative title). Master's thesis, The University of Western Ontario, In Preparation. (cited on page 251)

[191] M. Thadani and Y. Khalidi. An Efficient Zero-Copy I/O Framework for UNIX. *Sun Microsystems Laboratories, Inc. Technical Report SMLI TR-95-39*, May 1995. (cited on page 37)

[192] The FreeBSD Documentation Project. The FreeBSD Handbook. *Available from http://www.freebsd.org*, June 2000. (cited on page 22)

[193] M. Thottan and C. Ji. Adaptive Thresholding for Proactive Network Problem Detection. *Proceedings of the Third IEEE International Workshop on Systems Management*, Newport, Rhode Island, April 1998. (cited on pages 36, 251)

[194] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Toward a Predictable Real-Time System. *Proceedings of USENIX Mach Workshop*, October 1990. (cited on page 33)

[195] C. Topolcic. Experimental Internet Stream Protocol, Version 2 (ST-II). *Internet RFC 1190*, 1990. (cited on page 35)

[196] U. Vahalia. *Unix Internals: The New Frontiers.* Prentice Hall, 1996. (cited on pages 120, 121, 125, 128)

[197] R. Vallee and F. Kaudel. *Connection Management Upon Dynamic Bandwidth Changes in IMA, WATM and ADSL.* ATM Forum Contribution 96-1768, San Diego, California, February 1997. (cited on page 36)

[198] C. van den Branden Lambrecht and O. Verscheure. Perceptual Quality Measure using a Spatio-Temporal Model of the Human Visual System. *Proceedings of the 1996 SPIE International Symposium on Visual Communications and Image Processing*, San Jose, California, February 1996. (cited on page 31)

[199] R. Vanegas, J. Zinky, J. Loyall, D. Karr, R. Schantz, and D. Bakken. QuO's Runtime Support for Quality of Service in Distributed Objects. *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, England, September 1998. (cited on page 26)

[200] C. Venkatramani and T. Chiueh. Design, Implementation, and Evaluation of A Software-Driven Real-Time Ethernet Protocol. *Proceedings of the 1995 ACM SIGCOMM Conference*, Cambridge, Massachusetts, August 1995. (cited on page 36)

[201] A. Vogel, B. Kerhevré, G. V. Bochmann, and J. Gecsei. Distributed Multimedia Applications and Quality of Service: A Survey. *Proceedings of the 1994 Centre for Advanced Studies Confererence*, Toronto, Canada, November 1994. (cited on page 1)

[202] C. Waldspurger and W. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Management. *Technical Memo MIT/LCS/TM-528*, Cambridge, Massachusetts, June 1995. (cited on page 33)

[203] J. Walpole, R. Koster, S. Cen, C. Cowan, D. Maier, D. McNamee, C. Pu, D. Steere, and L. Yu. A Player for Adaptive MPEG Video Streaming Over The Internet. *Proceedings of the 26th Applied Imagery Pattern Recognition Workshop*, Washington DC, October 1997. (cited on page 32)

[204] W. Walsh, M. Wellman, P. Wurman, and J. MacKie-Mason. Some Economics of Marked-Based Distrubted Scheduling. *Proceedings of the Eighteenth International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998. (cited on page 30)

[205] F. Wang, C. Wen, C. Cheng, M. Lee, T. Lin, S. Wang, and Y. Oyang. Memory and Disk Bandwidth Management in Video-on-Demand Servers. *Proceedings of the Sixth International Workshop on Network and Operating System Support for Digital and Audio Video*, Zushi, Japan, April 1996. (cited on page 37)

[206] M. Wiles and R. Cherukuri. *Connection Modify.* ATM Forum Contribution 97-0231, Chicago, Illinois, April 1997. (cited on page 36)

[207] L. C. Wolf. *Resource Management for Distributed Multimedia Systems.* Kluwer Academic Publishers, Norwell, Massachusetts, 1996. (cited on page 3)

[208] B. Worthington, G. Ganger, and Y. Patt. Scheduling Algorithms for Modern Disk Drives. *Proceedings of the 1994 ACM Sigmetrics Conference*, May 1994. (cited on pages 19, 262)

[209] D. Yau and S. Lam. Adaptive Rate-Controlled Scheduling for Multimedia Applications. *Proceedings of the 1996 ACM Multimedia Conference*, Boston, Massachusetts, November 1996. (cited on pages 33, 34)

[210] H. Zhang and S. Keshav. Comparison of Rate-Based Service Disciplines. *Proceedings of SIGCOMM'91*, Zurich, Switzerland, September 1991. (cited on page 19)

[211] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource Reservation Protocol. *IEEE Network Magazine, 9 (5)*, September 1993. (cited on pages 35, 119)

[212] D. Zilio. Self-Managed Databases. *Presented at the 2000 Centre for Advanced Studies Confererence*, Toronto, Canada, November 2000. (cited on page 29)

[213] J. Zinky, D. Bakken, and R. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems 3(1)*, April, 1997. (cited on page 26)

[214] Zona Research Inc. The Need for Speed. *Zona Research Report*, June 1999. (cited on page 2)

[215] Zona Research Inc. Shop Until You Drop? A Glimpse into Internet Shopping Success. *Zona Assessment Paper, Issue 6*, October 1998. (cited on page 2)

# Vita

## Personal Information

**Name:** Michael James Katchabaw

**Place of Birth:** London, Ontario, Canada

**Year of Birth:** 1972

## Educational Background

1. 1991-1995: B. Sc. Honours Computer Science, The University of Western Ontario, Canada.

    - Thesis: Event and Notification Enhancements to GDMO and OSIMIS
    - Supervisors: Dr. J. W. K. Hong and Dr. M. A. Bauer

2. 1995-2002: Ph.D. Computer Science (Fall Convocation), The University of Western Ontario, Canada.

    - Dissertation: Quality of Service Resource Management
    - Supervisors: Dr. M. A. Bauer and Dr. H. L. Lutfiyya

## Awards, Scholarships, and Fellowships Received

| Date(s) | Award | Location of Tenure |
|---|---|---|
| April 2000 | Computer Science Graduate Award of Excellence | The University of Western Ontario |
| August 1999 | University Student's Council Teaching Honour Roll | The University of Western Ontario |
| Jan. 1999 - Dec. 1999 | IBM Research Fellowship | Centre for Advanced Studies, IBM Canada |
| Sept. 1998 | Graduate Tuition Scholarship | The University of Western Ontario |
| Sept. 1998 | Special University Scholarship | The University of Western Ontario |
| June 1998 | Faculty of Science Graduate Teaching Award | The University of Western Ontario |
| Jan. 1998 - Dec. 1998 | IBM Research Fellowship | Centre for Advanced Studies, IBM Canada |
| Sept. 1997 - Aug. 1999 | NSERC Post Graduate Scholarship | The University of Western Ontario |
| Sept. 1997 | Graduate Tuition Scholarship | The University of Western Ontario |
| Sept. 1997 | Special University Scholarship | The University of Western Ontario |
| Sept. 1997 | Graduate Teaching Assistantship | The University of Western Ontario |
| Jan. 1997 - Dec. 1997 | IBM Research Fellowship | Centre for Advanced Studies, IBM Canada |
| Sept. 1996 | Graduate Tuition Scholarship | The University of Western Ontario |
| Sept. 1996 | Special University Scholarship | The University of Western Ontario |
| Sept. 1996 | Graduate Teaching Assistantship | The University of Western Ontario |

| Date(s) | Award | Location of Tenure |
|---|---|---|
| Sept. 1995 - Aug. 1997 | NSERC Post Graduate Scholarship | The University of Western Ontario |
| Sept. 1995 | Graduate Tuition Scholarship | The University of Western Ontario |
| Sept. 1995 | Special University Scholarship | The University of Western Ontario |
| Sept. 1995 | Graduate Teaching Assistantship | The University of Western Ontario |
| June 1995 | Governor General Academic Medal | The University of Western Ontario |
| June 1995 | The J. B. Bancroft Science Prize and Medal | The University of Western Ontario |
| June 1995 | The UWO Gold Medal in Honours Computer Science | The University of Western Ontario |
| Sept. 1994 - May. 1995 | Computing Research Association Honourable Mention | The University of Western Ontario |
| May 1994 - Sept. 1994 | NSERC Summer Undergraduate Research Award | The University of Western Ontario |
| Fall 1993 - Spring 1994 | Computer Science Edna Jeffery Scholarship | The University of Western Ontario |
| Fall 1993 - Spring 1994 | The University of Western Ontario Faculty Association Scholarship | The University of Western Ontario |
| Fall 1991 - Spring 1995 | Canada Scholarship | The University of Western Ontario |
| Fall 1991 - Spring 1995 | Faculty of Science Ivan C. Rand Scholarship | The University of Western Ontario |

# Research

## Areas of Specialization

Future research is expected to focus on the following general areas:

- Distributed computing
- Operating systems
- High-speed networking
- Quality of service management
- Development of manageable applications
- Software engineering for manageability

## Publications

### Book Chapters

1. M. J. Katchabaw, H. L. Lutfiyya, M. A. Bauer. A Model for Resource Management to Support Quality of Service in Electronic Commerce Applications. Appeared in *Electronic Commerce Technology Trends: Challenges and Opportunities*, Midrange Computing/IIR, 1999, pp. 239-254.

### Refereed Journal Papers

2. M. J. Katchabaw, H. L. Lutfiyya, M. A. Bauer. Driving Resource Management With Application-Level Quality of Service Specifications. Decision Support Systems Journal, Volume 28, Elsevier Science Press, 2000, pp. 71-87.

3. M. J. Katchabaw, S. L. Howard, H. L. Lutfiyya, A. D. Marshall, M. A. Bauer. Making Distributed Applications Manageable Through Instrumentation. *Journal of Systems and Software*, 45 (1999), 1999, pp. 81-97.

**Refereed Conference Papers**


4. G. P. Molenkamp, H. L. Lutfiyya, M. J. Katchabaw, and M. A. Bauer. Diagnosing Quality of Service Faults in Distributed Applications. To Appear in the Proceedings of the 21st IEEE International Performance, Computing, and Communications Conference, Phoenix, Arizona, April 2002.

5. G. P. Molenkamp, M. J. Katchabaw, H. L. Lutfiyya, and M. A. Bauer. Distributed Resource Management to Support Distributed Application-Specific Quality of Service. Proceedings of the Fourth IFIP/IEEE International Conference on Management of Multimedia Networks and Services, Chicago, Illinois, October 2001, pp. 142-159.

6. H. L. Lutfiyya, G. P. Molenkamp, M. J. Katchabaw, and M. A. Bauer. Issues in Managing Soft QoS Requirements in Distributed Systems Using a Policy-Based Framework. Proceedings of the International Workshop on Policies, Bristol, England, January 2001, pp. 461-468.

7. G. P. Molenkamp, M. J. Katchabaw, H. L. Lutfiyya, and M. A. Bauer. Managing Soft QoS Requirements in Distributed Systems. Proceedings of Multimedia Systems Workshop (ICPP), Toronto, Canada, August 2000, pp. 461-468.

8. M. J. Katchabaw, H. L. Lutfiyya, M. A. Bauer. Using User Hints to Guide Resource Management for Quality of Service. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, June 1999, pp. 1069-1075.

9. M. J. Katchabaw, H. L. Lutfiyya, M. A. Bauer. Driving Resource Management With Application-Level Quality of Service Specifications. Proceedings of the First International Conference on Information and Computation Economies, Charleston, South Carolina, October 1998, pp. 83-91.

10. M. J. Katchabaw, H. L. Lutfiyya, M. A. Bauer. A Model for Resource Management to Support End-to-End Application-Driven Quality of Service. *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems*, Las Vegas, Nevada, October 1998.

11. M. J. Katchabaw, H. L. Lutfiyya, M. A. Bauer. A Quality of Service Management Testbed. *Proceedings of the Third International Workshop on Systems Management*, Newport, Rhode Island, April 1998, pp. 57-67.

12. M. J. Katchabaw, S. L. Howard, H. L. Lutfiyya, A. D. Marshall, M. A. Bauer. Making Distributed Applications Manageable Through Instrumentation. *Proceedings of*

*the 1997 Workshop on Parallel and Distributed Software Engineering*, Boston, Massachusetts, May 1997, pp. 84-94.

13. S. L. Howard, H. L. Lutfiyya, M. J. Katchabaw, M. A. Bauer. Supporting Dynamic Policy Change Using CORBA System Management Facilities. *IEEE International Symposium on Integrated Network Management*, San Diego, California, May 1997, pp. 527-538.

14. M. J. Katchabaw, S. L. Howard, A. D. Marshall, M. A. Bauer. Evaluating the Costs of Management: A Distributed Applications Management Testbed. *Proceedings of the 1996 IBM Centre for Advanced Studies Conference*, Toronto, Canada, November, 1996, pp. 29-41[1].

15. M. J. Katchabaw, H. L. Lutfiyya, A. D. Marshall, M. A. Bauer. Policy-Driven Fault Management in Distributed Systems. *Proceedings of the International Symposium on Software Reliability Engineering*, White Plains, New York, November 1996, pp. 236-245.

16. M. J. Katchabaw, S. L. Howard, H. L. Lutfiyya, M. A. Bauer. Efficient Management Data Acquisition and Run-time Control of DCE Applications Using the OSI Management Framework. *Proceedings of the Second International Workshop on Systems Management*, Toronto, Canada, June 1996, pp. 104-111.

17. A. D. Marshall, M. J. Katchabaw, M. A. Bauer. Using Software Metrics Tools for Maintenance Decisions: A Classroom Exercise. *Proceedings of the International Symposium on the Assessment of Software Tools*, Toronto, Canada, May 1996, pp. 47-58.

18. M. J. Katchabaw, M. M. Khurana, J. W. Hong, M. A. Bauer. Interworking of Traders in a Distributed Computing Environment. *Proceedings of the 1995 IBM Centre for Advanced Studies Conference*, Toronto, Canada, November, 1995, pp. 1-12.

19. M. J. Katchabaw, J. W. Hong, M. A. Bauer. Behavioural Specification and Notification Enhancements to GDMO. *Proceedings of the 1995 International Workshop on Distributed Systems Operations and Management*, Ottawa, Canada, October 1995, sec. 6.3.

20. J. W. Hong, M. J. Katchabaw, M. A. Bauer, H. Lutfiyya. Modeling and Management of Distributed Applications and Services Using the OSI Management Framework. *Proceedings of the International Conference on Computer Communication*, Seoul, Korea, August 1995, pp. 215-220.

[1]This paper received the Best Paper award at this conference.

**Technical Reports**

21. H. L. Lutfiyya, G. P. Molenkamp, M. J. Katchabaw, M. A. Bauer. Managing Quality of Service Requirements. Technical Report #547, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999, pp. 21.

22. M. J. Katchabaw, H. L. Lutfiyya, M. A. Bauer. Memory Management for Quality of Service Provisioning. Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999, pp. 26.

23. M. J. Katchabaw, H. L. Lutfiyya, M. A. Bauer. A Survey of Windowing Environment Usage Patterns. Technical Report #540, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999, pp. 37.

24. M. J. Katchabaw, H. L. Lutfiyya, M. A. Bauer. A Model for Resource Management to Support End-to-End Application-Driven Quality of Service. Technical Report #528, Department of Computer Science, The University of Western Ontario, London, Canada, July 1998, pp. 15.

25. J. W. Hong, M. J. Katchabaw, M. A. Bauer, H. L. Lutfiyya. *Distributed Applications Management Using The OSI Management Framework.* Technical Report #448, Department of Computer Science, The University of Western Ontario, London, Canada, January 1995, pp. 22.

26. J. W. Hong, M. A. Bauer, J. A. O'Neill, M. J. Katchabaw. *The X.500 Directory Service Testbed.* Technical Report #365, Department of Computer Science, The University of Western Ontario, London, Canada, January 1995, pp. 127.

**Invited Talks**

27. M. J. Katchabaw. User Hints: An Informed Approach to Quality of Service Resource Management. Presented at the 1999 IBM Centre for Advanced Studies Conference, Toronto, Canada, November, 1999.

28. M. J. Katchabaw. A Model for Resource Management to Support Quality of Service in Electronic Commerce Applications. Presented at the First International Workshop on the Technological Challenges of Electronic Commerce. Toronto, Canada, September 1998.

**Poster Presentations**

29. M. J. Katchabaw, G. P. Molenkamp, H. L. Lutfiyya, M. A. Bauer. QoS Management: A Policy Driven Approach. Presented at the 2000 IBM Centre for Advanced Studies Conference, Toronto, Canada, November 2000.

30. M. J. Katchabaw, H. L. Lutfiyya, M. A. Bauer. Dynamic Memory Management for Quality of Service Provisioning. Presented at the 1999 IBM Centre for Advanced Studies Conference, Toronto, Canada, November, 1999.

31. G. P. Molenkamp, M. J. Katchabaw, H. L. Lutfiyya, M. A. Bauer. Policy-Driven Quality of Service Management. Presented at the 1999 CITO Research Forum. Ottawa, Canada, May 1999.

**Department Presentations**

32. M. J. Katchabaw. Policy Based Quality of Service Management. Presented at the 2000 University of Western Ontario Research in Computer Science Workshop (UWORCS 2000). London, Canada, September, 2000.

33. M. J. Katchabaw. User Hints: An Informed Approach to Quality of Service Resource Management. Presented at the 1999 University of Western Ontario Research in Computer Science Workshop (UWORCS'99). London, Canada, November 1999[2].

34. M. J. Katchabaw. Investigating the Impact of End-System Utilization on Application Quality of Service. Presented at the 1997 University of Western Ontario Research in Computer Science Workshop (UWORCS'97). London, Canada, November 1997.

[2] This presentation received the Best Presentation award at this workshop.

## Research Experience

### X.500 Management

- Dates: 1993-1994
- Scope: Departmental CORDS (COnsortium for Research on Distributed Systems) group
- Notes: This work was used for other research and graduate course work.

### Distributed Project Management

- Dates: 1993-1994
- Scope: Entire CORDS group (members from four IBM research laboratories, six Canadian universities, four American universities, and other centres)
- Notes: Demonstrated at the 1993 IBM Centre for Advanced Studies Conference.

### Distributed Applications and Systems Management

- Dates: 1994-1996
- Scope: Entire MANDAS (MANagement of Distributed Applications and Systems) group (members from the IBM Toronto Lab, five Canadian universities, one American university)
- Notes: Demonstrated at 1994, 1995, and 1996 IBM Centre for Advanced Studies Conferences.

### Quality of Service Management

- Dates: 1995-Present
- Scope: Ph.D Thesis and the departmental Distributed Systems and Software Engineering research group
- Notes: Demonstrated at the 1999 and 2000 IBM Centre for Advanced Studies Conferences, as well as the 1999 CITO Research Forum.

**ATM Networking**

- Dates: 1996-2000
- Scope: Originally joint between the Department of Computer Science and the Department of Electrical and Computer Engineering, later expanded formally with Applied Health Sciences.

**QoS Management for Electronic Commerce Applications**

- Dates: 1997-1999
- Scope: Managing Data for Electronic Commerce research group (including members from the IBM Toronto Lab and five Canadian universities)

# Teaching

## Teaching Experience

| Dates | Employer | Position Held |
|---|---|---|
| Jan. 2002 - Present | Department of Computer Science, The University of Western Ontario | Sessional Lecturer, Computer Science 211 |
| Sept. 1996 - May 1998 | Department of Computer Science, The University of Western Ontario | Sessional Lecturer, Computer Science 402 (Overall Effectiveness: 6.2 and 6.6, on a scale from 1 to 7) |
| May 1996 - Aug. 1996 | Department of Graduate Studies The University of Western Ontario | Completed GS500, *"The Theory and Practice of University Teaching"*, with Distinction |
| June 1995 - Dec. 1997 | Department of Computer Science, The University of Western Ontario | Teaching Assistant, Computer Science 305, Computer Science 402, Computer Science 457 |
| Sept. 1993 - May 1996 | Department of Computer Science, The University of Western Ontario | Course Consultant, Computer Science 612 |

## Guest Lectures

1. M. J. Katchabaw. The Kitchen Sink Over IP – Quality of Service Issues for Next Generation Applications. Given for Computer Science 457, Computer Networks II. Department of Computer Science, The University of Western Ontario, London, Canada. November 3, 1999. Repeated in 2000.

2. M. J. Katchabaw. Memory Management for Quality of Service Provisioning. Given for Computer Science 305, Operating Systems. Department of Computer Science, The University of Western Ontario, London, Canada. April 8, 1999.

# Service and Administrative Activity

## Committee Work

I have had the pleasure to serve on the following committees at the University of Western Ontario:

| Dates | Committee |
|---|---|
| 1999-2001 | Subcommittee on the World Wide Web (a subcommittee of SCITS) |
| 1999-2000 | Subcommittee on Information Security (a subcommittee of SCITS, successor of the Campus Computer Security Subcommittee) |
| 1998-Present | Computer Science Department Outreach Committee |
| 1998-1999 | Senate Committee on Information Technology Services (SCITS) |
| 1998-1999 | Subcommittee on Computing and Networking Services (a subcommittee of SCITS) |
| 1998-1999 | Campus Computer Security Subcommittee (a subcommittee of SCITS) |

## Conference/Workshop Organization

- Chair of the 1997 University of Western Ontario Research in Computer Science Workshop (UWORCS'97). London, Canada, November 1997.

## Other Administrative Activity

In addition to the above activities, experience has been gained in terms of promoting the Computer Science Department at the University of Western Ontario, and the recruitment of new students through a variety of other programs and activities over the past several years:

- Discovery Days: Various demonstrations and presentations to high school students from the surrounding area.
- March Break Activities: Demonstrations and discussions with high school students from the surrounding area.
- What's Your Major: Meeting with first year students in campus residences and discussing programs in Computer Science.
- Faculty of Science Receptions: Meeting with students during orientation to discuss questions about the Computer Science Department and its programs.

- Various other tours and demonstrations for visiting guests, Homecoming Weekend activities, and so on.
- Welcoming and meeting with new undergraduate and graduate students.

## Other Activities to Note

### Refereeing and Reviewing

| Dates | Publication |
|---|---|
| Nov. 2001 | Proceedings of the 2001 IBM Centre for Advanced Studies Conference |
| Nov. 2000 | Proceedings of the 2000 IBM Centre for Advanced Studies Conference |
| Nov. 1999 | Proceedings of the 1999 IBM Centre for Advanced Studies Conference |
| Sept. 1999 | Q. Mahmoud. Distributed Programming with Java. Reviewed for and published by Manning Publications Co., 1999. (Several other books and proposals have been reviewed for Manning Publications Co. that did not make it to press.) |
| Nov. 1998 | Proceedings of the 1998 IBM Centre for Advanced Studies Conference |
| Aug. 1998 | Proceedings of the Thirty-second Annual Hawaii International Conference on System Sciences (HICSS-32), Cluster Computing Minitrack |
| Apr. 1998 | Proceedings of the Third International Workshop on Systems Management |
| Aug. 1997 | Proceedings of the Thirty-first Annual Hawaii International Conference on System Sciences (HICSS-31), Performance Evaluation of Distributed Systems Minitrack |
| June 1996 | Proceedings of the Second International Workshop on Systems Management |

**Consulting**

| Dates | Employer | Position |
|---|---|---|
| Mar. 1999 | uuserve& | Unix and Solaris Consulting for Electronic Commerce and Electronic Business |