Managing QoS Requirements

Hanan Lutfiyya, Gary Molenkamp, Michael Katchabaw, and Mike Bauer
Department of Computer Science
The University of Western Ontario
London, Ontario CANADA N6A 5B7
+1 519 661 2111 (Ext. 86888)
hanan@csd.uwo.ca

November 22, 1999

ABSTRACT

We address the problem of soft Quality of Service (QoS) requirements for multimedia applications (e.g., distance education, telemedicine, electronic commerce). These applications will co-exist with more traditional applications for transaction and data processing and have soft real-time requirements. Unlike most other work in QoS management, we provide a framework that does not require either users or application developers to have detailed knowledge of the resources needed and resource scheduling and allocation techniques in use. These underlying details are effectively hidden. In this paper, we describe our strategy, an architecture of services to support the strategy and a prototype.

0.1 Keywords

QoS Requirements, Policies, Distributed Systems

1 Introduction

There is an increase in distributed applications requiring real-time services. This includes new multimedia applications such as video-on-demand, distance eduction, telemedicine, teleconferencing and electronic commerce. These applications will co-exist with more traditional applications for transaction processing, data processing, and software development.

Technical Report #547, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

Users of these application expect a high level of quality of service (QoS). By quality of service, we are referring to non-functional requirements such as performance or availability requirements. An example of a QoS requirement for a multimedia application that receives a video stream is the following: "The number of video frames per second displayed to the user must be at least 25 plus or minus 2 frames". Most of the QoS requirements for multimedia applications are considered to be *soft* in that the applications are still considered functionally correct if QoS requirements are not satisfied.

The allocation and scheduling of resources is referred to as *QoS management*. Many QoS management techniques provide a guarantee that resources will be available when needed by statically allocating resources based on worst-case needs. This is important in systems where applications must meeting their timing constraints to avoid disastrous consequences. These types of applications include flight control systems, chemical process control systems, and patient-monitoring systems. However, most multimedia applications have *soft* real-time requirements. Some QoS management techniques support these types of applications by initially allocating resources based on optimistic estimates of resource needs and assuming that the resource will be available when needed. The application is informed if it is not possible to satisfy its resource needs. The application can then either negotiate with the operating system and/or adapt its behaviour.

Much of this work assumes that the user is aware of resource needs in advance. For example, not only does there need to be a specification such as "The number of video frames per second displayed to the user must be at lest 25", but also the number of CPU cycles needed to get the 25 frames per second. The derivation of this information needs intricate knowledge of the hardware architecture, network and system software in the target environment. This places an additional burden on the developers of these applications.

This paper defines a framework that is *policy-driven* to address the problem of QoS management for multimedia applications that have soft real-time requirements, but does not not require either users or application developers to have detailed knowledge of the resources needed and resource scheduling and allocation techniques in use. These underlying details are effectively hidden.

This paper is organised as follows. In section 2, we describe our strategy. Section 3 describes the services needed to support the strategy described in Section 2. Section 4 describes a language for specifying requirements. Sections 5 and 6 describe an architecture for supporting the services described in Section 3. Section 6 describes our prototype. Sections 7 and 8 describe insights and future work.

Technical Report #547, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

2 Strategy

The general strategy that we use is shown in Figure 1. It is based on the ability to change the allocation of resources to a process during the process's session.

Given: Application p.
Initial resource allocation a.
QOS metric m.
QOS expectations e.

---

p is started with resource allocation a.
while (p **not done**) **do:**
    Execute code in p.
    $q \leftarrow$ QOS measured with m.
    if ($q > e$) **then**
        Determine allocation reduction r.
        $a \leftarrow a - r$.
    else if ($q < e$) **then**
        Determine allocation improvement i.
        $a \leftarrow a + i$.
    endif
endwhile

Figure 1: Application-Driven Resource Management Strategy

We will now use an example to illustrate the relationship between QoS requirements (or user expectations) and the resources allocated using this strategy. Assume that we have the policy that defines a QoS requirement that is informally stated as follows: "A given video application is to deliver video at a frame rate of 25 frames per second, plus or minus 2 frames". The process expecting the video is given an initial resource allocation. When a frame is received the metric representing the quality of service is measured (number of frames). If it exceeds the specified expectation the resource allocation is reduced. This allows the resource to be freed up for other processes. Otherwise, an allocation improvement is made. As we can see this strategy adjusts resource allocations until a suitable allocation is made that satisfies expectations.

This strategy does not guarantee that a QoS requirement will be satisfied. It tries to dynamically adjust resources as needed by the application. However, this may not always be possible. For example, let us assume that there are many other multimedia processes on the same host machine that also have soft QoS requirements. Administrative requirements may dictate that each of these processes should have equal access to resources; hence, it may be decided to allow each of the processes to degrade. On the otherhand, it may be decided that some applications have priority over other applications. Hence, resource allocation is prioritised and a differentiation resource allocation is allowed.

3 Services Needed to Implement the Strategy

We will now describe the services needed to implement the strategy presented in Section 2.

3.1 Enforcement

We will use the following example QoS requirement to illustrate enforcement: "A given video application is to deliver video at a frame rate of 25 frames per second, plus or minus 2 frames".

The services needed to make a system comply with QoS requirements are referred to as *enforcement* services.

3.1.1 Violation Detection

Violation detection occurs when the application behaviour is observed not to satisfy the specified QoS requirements. The violation (also called a *symptom*) is a manifestation of a fault in the system. The detection of a violation of a QoS requirement requires that we have the means of mapping the QoS requirement into mechanisms that collect data that characterise run-time behaviour.

3.1.2 Violation Location

Violation location takes symptoms and identifies the cause and location of the fault in the system. In the example QoS requirement, a violation of the requirement is that the video application is not receiving frames at the specified frame rate of between 23 and 27 frames per second. There are many possible causes for this. For example, the video application may not be getting enough local processor cycles, the server process is not getting enough cycles, a process failed or there is an unexpected load on a network switch. Locating the cause is an important step in determining the appropriate resource allocation. For example, if the problem is that there is an unexpected load on a network switch then there is no need to adjust resources on the video application process's host.

3.1.3 Adaptation

Adaptation refers to the corrective actions taken to repair or otherwise recover from a fault, so that the system returns to compliance with the QoS requirements. Examples include providing more processor cycles, restarting a failed

Technical Report #547, Department of Computer Science, The University of Western Ontario,
London, Canada, November 1999

process or rerouting traffic around the congested network switch. The corrective action(s) to be taken depends on the cause of the QoS requirement violation, but it will also depend on the constraints imposed to achieving the QoS requirement. For example, one possible corrective action is to adjust the CPU priority of the video application receiving the video stream. However, what if there are several multimedia applications on the same host. One constraint may be that if the other applications are not multimedia then the CPU adjustment may be r. On the otherhand, if the other applications are multimedia, then look at the priority associated with the user of the video application.

These constraints are a second category of requirements. However, these are not user QoS requirements expected of an application, but rather they are administrative or organisational requirements. These requirements will vary between different administrative domains and will vary over time.

3.2 Policy Specification and Distribution

We have identified two different types of requirements: requirements that express user expectations of the run-time behaviour of an application and requirements that express constraints on the set of possible corrective actions. It must be possible to specify the requirements and store the requirements. To do this, we allow the requirements to expressed as *policies* that can be stored, queried and distributed. We need services that allow for the following:

- The ability of administrators must be able to define and add new policies. This requires a formal language for specifying policies.

- The accommodation of changes in requirements. For example, the requirements of an application depend on the user who has invoked the application. Thus, different sessions of the same application will have different QoS requirements. In addition, the constraints on the possible adaptations will also change during the lifetime of a system. The implication of these changes is that it should be possible for the administrator to add and delete policies and have the policy specification dynamically distributed to appropriate QoS management system components.

4 Policy Specification

Policies are treated as first-class objects i.e., they can be stored and manipulated. We will distinguish between the two different kinds of policies. The policies that express user QoS expectations will be referred to as *policy* while the policies that express constraints on the possible adaptations are referred to as *rules*. We will now describe how to specify QoS policies. The formal specification of rules is left as future work.

Technical Report #547, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

The QoS policy is expressed as a set of constraints on attribute values. An attribute is a property associated with an entity.

A partial (modified) grammar for specifying the QoS policies is given below. It is based from the grammar found in [25].

```
policy        : policy-header "-" policy-body """
policy-header : "POLICY" policyid
policy-body   : "MODE" mode
                "SUBJECT" subject
                ["TARGET"  target]
                "ATTRIBUTE-CONSTRAINT" const-expr
                 ["ACTION" action]
                 ["DESCRIPTION" description]
policyid      : identifier
mode          : "O" | "A"
subject       : identifier
target        : identifier
action        : "NONE" |
                {object "->" method, }*
                object "->" method
object        : identifier
method        : identifier
description   : string
domain-expr   : domainid
```

The role of each of these components is examined.

Policy Identifier: Each policy is assigned a unique identifier when it is created.

Mode: In general, policies fall into one of two general categories [25]: *Authorisation policies* (MODE A) define what activities are permitted while *obligation policies* (MODE O) define what activities must be carried out. In this paper, we are dealing with obligation policies.

Attribute-Constraint: An attribute is a particular property of an entity. A constraint term identifies constraints on an attribute. Composition operators (e.g., and, or, and after) allow constraint terms to be combined into constraint expressions.

Subject: The subject defines which entity should be assigned responsibility for the policy.

Target: The target is the entity that the Attribute-Constraint is being applied to. It is responsible for carrying out the action (specified in the ACTION clause) upon the Attribute-Constraint being false.

4.1 Example Policy

The following example shows how the formal syntax can be used to specify policies. In the next section, we will show how the syntax can express policies that are rules.

Example 4.1: The policy below defines a QoS requirement that is informally stated as follows: "A given video application is to deliver video at a frame rate of 25 frames per second, plus or minus 2 frames and a jitter rate of less than 1.25".

```
POLICY reportPoorQualityVideo {
DESCRIPTION  Defines the expected frame rate
SUBJECT               (...)/QoS.HostManager
TARGET                (...)/VideoApplication
ATTRIBUTE-CONSTRAINT  frame_rate = 25(+2)(-2)
                      AND jitter_rate < 1.25
ACTION                fps->read(), buffer->read()
                      buffer->read(),
                      send readings to (...)/QoS.HostManager
}
```

In this example, frame_rate and jitter_rate are attributes. frame_rate = 25(+2)(-2) is a constraint term, where the constraint on the frame_rate attribute is that its value must be 25 give or take a couple of frames. jitter_rate is another constraint term, where the constraint on the jitter_rate attribute is that its value must be less than 1.25.

If the value of the frame_rate attribute is less than 23 or greater than 27 or the jitter_rate is less than 1.25 then the policy is considered to be violated. (...)/QoS.HostManager refers to the process (discussed in the next section) that receives notification if the entity specified in the Target does not satisfy the constraint expression specified in Attribute-Constraint. The (...) notation includes other identifying information such as hostname, etc; The action (as specified in the ACTION attribute) to be taken is to read the frames per second and buffer size. These readings are packaged into a report to be sent to (...)/QoS.HostManager.

5 Architectural Design

To support our strategic approach and the services that we have identified as necessary, we developed the architecture shown in Figure 2. This architecture is derived from our earlier work in application instrumentation [14]. This approach requires the insertion of code into applications at strategic locations to facilitate the collection of quality of service metrics and exertion of control over the applications. While some measurements can be taken by observing external

Technical Report #547, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

application behaviour and rudimentary control can be achieved through operating system interactions, work in this area has found that these approaches are limiting in both accuracy and the kinds of metrics and control available.

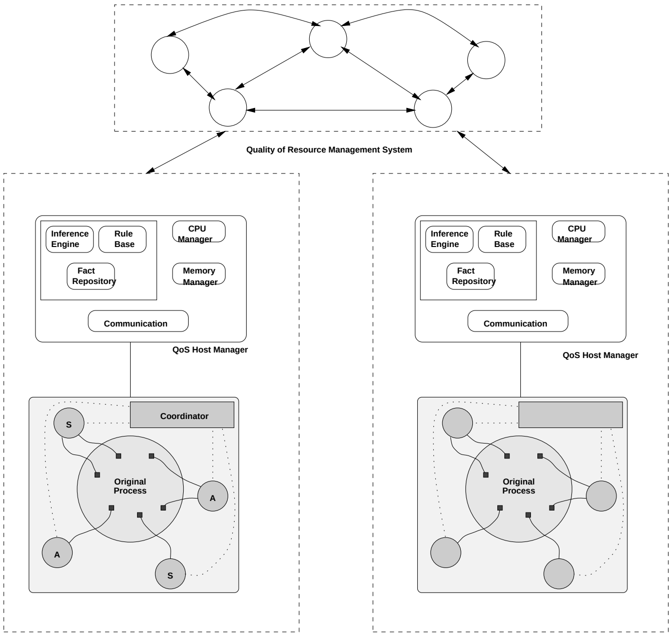

Figure 2: Application-Driven Resource Management Architecture

The components of this architecture are discussed in the sections below.

5.1 Instrumented Process

An instrumented process is an application process with embedded instrumentation code. It is this instrumentation code that enables the management of the application process. The architecture components that comprise the instrumentation code are discussed below.

Sensors. Sensors are used to collect, maintain, and (perhaps) process a wide variety of metric information within the instrumented processes. Sensors get their input data from probes inserted at strategic points in the application code or by reading other sensors. During run-time, sensors can be enabled or disabled, reporting intervals can be adjusted and thresholds can be modified.

Actuators. Actuators are used to encapsulate functions that can exert control over the instrumented process to change its operation or behaviour. In this current work, they are not used extensively, but can be used to support quality of service negotiation, adaptation, and other functions in the future.

Technical Report #547, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

Probes. Probes are embedded in process code to facilitate interactions with sensors and actuators. Probes allow sensors to collect metric 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 must be injected into the original process source code—all other components need only reside in the same address space as the process, and do not require code modifications.

Sensors and actuators are classes. Probes can either be methods of the sensors and actuators or be functions that call these methods.

Example 5.1. As an example, consider the sample pseudo-code for a video playback application in Figure 3 that has the policy stated in Example 4.1. This QoS requirement is translated into initial thresholds for sensor, s_1, capable of measuring the frame rate. The target threshold is 25 frames per second, the upper threshold is 27 frames per second and the lower threshold is 23 frames per second. This QoS requirement also translates into an initial threshold for sensor, s_2, that measures jitter.

Sensor s_1 includes at least the following two probes which are also methods of s_1: (1) An initialisation probe that takes as a parameter the default threshold target value. This probe gives a value to the target, upper and lower thresholds. (2) A probe that does the following when triggered by the process after the application retrieves a video frame, decodes it and displays it: (i) Determines the elapsed time since the last frame delivered. (ii) Checks to see if this falls within a particular range defined by the lower and upper acceptable thresholds. Unusual spikes are filtered out. (iii) If there is a problem, it will inform the coordinator c. $\Box$

How does a sensor relate to policies? A sensor collects values for an attribute of the process. This attribute is part of a policy (or policies that are being applied to the application). The application policies specify constraints on process attributes. We assume that a sensor is responsible for monitoring a specific attribute. If the sensor finds that a constraint involving that attribute is violated, it reports this to the coordinator as an *alarm report*. For example 4.1, it is assumed that two sensors are needed: one for the frame rate and one for the jitter. We also note that a sensor may provide values to be used in more than one policy. This means that there is a many to many relationship between policies and sensors.

Coordinator. The coordinator keeps track of the policies that are to be enforced for its process by maintaining a list of policy objects for each policy. Each policy object consists of a boolean expression that represents the *Attribute-Constraint* of the policy specified in the policy identifier. If the coordinator receives an alarm report from a *sensor* it determines those policy objects that represent policies that use the attribute associated with the sensor. For each such policy object, the coordinator maps the alarm report to a variable that is used in the boolean expression. This variable is set to false and the boolean

Technical Report #547, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

expression is evaluated. If it is false, then a report is sent to the QoS Host Manager. All knowledge of the QoS Host Manager is confined to the coordinator, effectively hiding it from the remaining instrumentation components.
Example 5.2. This example describes a coordinator that has the policy stated in Example 4.1. If an alarm report is received from s_1 then the coordinator checks the list of policy objects and evaluates the boolean expression associated with the policy specified in Example 4.1. If the expression evaluates to false then the QoS Host Manager is notified. This expression evaluates to false if either s_1 and s_2 have sent alarm reports. If the composition operator in Example 4.1 had been an or then the expression would evaluate to false if both s_1 and s_2 sent alarm reports. In the case of an evaluation to false, the coordinator does the action specified in the ACTION clause. In this case, it reads the frame rate, jitter rate and buffer size, put them into a message report and sends this off to the QoS Host Manager. □

Given: Video application v.
QOS expectations e.

```
Perform initialisation for v.
Initialise coordinator c.
Initialise sensor s with e.
while (v not done) do:
      Retrieve next video frame f.
      Decode and display f.
      Execute s → probe.
endwhile
```

Figure 3: Instrumentation Example

Example 5.2 makes use of an additional sensor that is used to provide further monitored information that can be used by the QoS Host Manager. This sensor monitors the length of the communication buffer. The purpose of this will be illustrated later.

5.2 Quality of Service Host Manager

The QoS Host Manager receives notifications from a process (through the process's coordinator) when a policy has been violated. The QoS Host Manager has a set of rules that are used to determine the corrective action(s). This involves determining the cause of the policy violation and based on the cause determine a corrective action(s). The process of determining the rules to be applied is called inferencing. Inferencing is used to formulate other facts or a hypothesis. Inferencing is performed by the Inference Engine component of the QoS Host Manager. The inferencer chooses which rules can be applied based on the fact repository. The inferencing that can take place can either be as complex as backward chaining (working backwards from a goal to start), forward chaining (vice-versa) or as relatively simple as a lookup. In this work we used forward chaining.

Let us go back to the policy stated in Example 4.1. A possible corrective action is to increase the CPU priority of the process receiving the frames so that it can process more frames in shorter time. However, what if the problem is not local e.g., the server process feeding the frames is on a host machine that is overloaded or the network connections between the server and the client are congested. If the problem is not local, then changing the CPU priority of the client process will not improve the frame rate. Worse yet, it will have a negative impact on the performance of other processes without any benefit.

Thus, one rule can be stated as follows: if the communication buffer size is below some threshold then send a notification to a management process which is part of the QoS Resource Management System. The implication of a small buffer size is that the process is able to process the received frames fast enough and that the problem is either a network problem or a server problem. The management process will also have a rule set for determining the cause of a problem and determining corrective action(s). This management process may have to interact with other management entities.

Another rule states that if the communication buffer size is above some threshold (implying that the process is not able to process frames fast enough) then the CPU manager component is invoked to adjust the CPU priority of the violated process. Additional rules are used to determine how much to increase CPU priority based on how close the policy is to being satisfied.

This is a very simple set of rules. A more complex set of rules would include rules that reflect administrative requirements. We have begun to look at such rules in the context of electronic commerce applications.

6 Policy Distribution

Earlier we identified that we need to be able to accommodate changes in requirements; this included both policies and rules. This means that we need to support the addition and deletion of policies/rules and the distribution of policies/rules to the relevant management components. In this work, this means that policies about an application should be able to change for each time the application runs and that the rules in the QoS Host Manager should also be able to change. These changes should not result in recompiling (unless, of course, there is a change in the actual attributes needed). In this section, we will de-

Technical Report #547, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

scribe the applications and services needed to support policy distribution. It is expected that the services are part of the Quality of Resource Management System of Figure 2. Our current implementation focusses on policies.

6.1 Domains

There is the potential for a great number of applications and users in a large distributed environment. Thus, it is not practical to define and maintain a separate set of policies for every user and application combination. What is needed is to have some kind of partition and divide the system into smaller groups. We can then specify the relationship between these groups and the mapping from entities to these groups, thus greatly reducing the complexity. The resulting group is called a *domain* [25]. A domain can be a set of users that share QoS requirements for a specific application or sets of application. Every user has a security identity which is mapped to a domain. The reason for a security identifier is that different users will be able to use the same application. However, different users may have different quality of service requirements. A security identity mapped to a domain and then having the domain specified in a policy is a way of identifying the QoS requirements that a particular user is entitled to have.

6.2 Information Model

We begin with a partial description of the information model used to represent the data needed for policy distribution (described in Section 3). We have identified the following different types of data needed. Part of the data needed and the and the relationship between the data is shown in Figure 4. The data and the relationships between the data are as follows:

Application. In this context, an application refers to an executable that can be instantiated to a process.

Sensor. Information maintained for a sensor includes sensor identifier and an attribute identifier indicating which attribute(s) that the sensor is maintaining. An application may need several sensors and a sensor may be associated with multiple applications. Hence, there is a many to many relationship between Application and Sensor.

Domain. This represents information about a domain. An attribute of the relationship between a Domain object and an Application object is a Policy object. Hence, any User object associated with (belonging) to a Domain object wanting to use the application specified in a Application object can expect the QoS requirements specified in the associated Policy object.

User. This represents information about a user. One item of information is a security identifier.

Technical Report #547, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

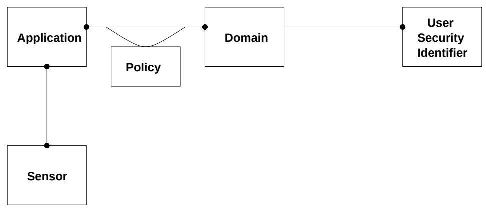

Figure 4: Information Model

6.3 Repository Service

The Repository Service allows for the storage and retrieval of the data specified in the previous subsection.

6.4 Policy Administration Applications

Authorized administrators must have a way to add and remove policies, define domains, browse policy information, etc; A policy administration application provides a user interface to facilitate this type of activity.

6.5 Policy Administration Service (PAS)

The Policy Administration Service controls external access to the information specified in Section 6.1. The PAS is used by other applications (e.g. Policy Administration Applications) for information integrity. For example, the PAS ensures that an application has sensors that collect the attributes specified in the policies.

6.6 Rules Distribution

This service distributes rules to the QoS Host Managers.

6.7 Configuration Service

The configuration service is responsible maintaining descriptive and location information about the entities of the distributed system. When a process starts up, it uses a probe called Process_instrumentationInit which registers the process with the configuration service (which in turn creates information for that object). This requires that the security identifier is passed from the application (the distribution of security identifiers to applications is not a subject of this paper). This security identifier is taken by the configuration service and passed to the policy repository service which maps it into the appropriate domain and retrieves the appropriate policies. This is sent to the coordinator component

Technical Report #547, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

that creates a list of policy objects where each policy object has the following information: (i) An expression which represents the *Attribute-Constraint* in the policy and (ii) A reference to the action to be taken if the condition in the *Attribute-Constraint* is violated.

7 Prototype

To explore and evaluate the architecture, we developed a prototype system based on the architecture presented in the previous section. This prototype has been implemented for Solaris 2.6. We implemented the sensors described in Section 5. Instrumented processes communicate with the QoS Host manager using message queues and socket calls at the initialisation of the processes. The QoS Host Manager has a simple set of rules that were described in Section 5. The QoS Host Manager adjusts allocations dynamically through changes to the time-sharing priorities assigned to processes using the *priocntl()* system call. The inference engine, rule set and fact repository are implemented using CLIPs. We have implemented limited versions of each of the services identified in Section 6 except for the Rules Distribution service. We have one management process in the Quality of Resource Management System that receives reports from QoS Host Managers. It has rules that can distinguish between a server machine problem and a network problem (this usually requires a query of other QoS Host Managers) and can tell a QoS Host Manager on a server machine to increase the CPU priority of the server process (assuming the problem in that the server process is not getting enough CPU cycles).

Figure 7 compares the mean video playback throughput, in frames per second, for the MPEG video player [23] under normal Solaris scheduling and with our QoS Host Manager (with a CPU Resource Manager) in place. This assumes that the other processes are not multimedia applications. From this figure, we can see that video throughput dropped dramatically under an increasing CPU load when normal Solaris scheduling was used. With our resource manager in place, however, throughput remained reasonably consistent around 28 frames per second – well within the acceptable limits set by the quality of service policy for the video player.

8 Related Work

There are three areas of related work: QoS management, policies and fault isolation.

8.1 QoS Management

We have already discussed some of the characteristics of existing work in QoS management in the introduction. The main characteristic that we addressed

Technical Report #547, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

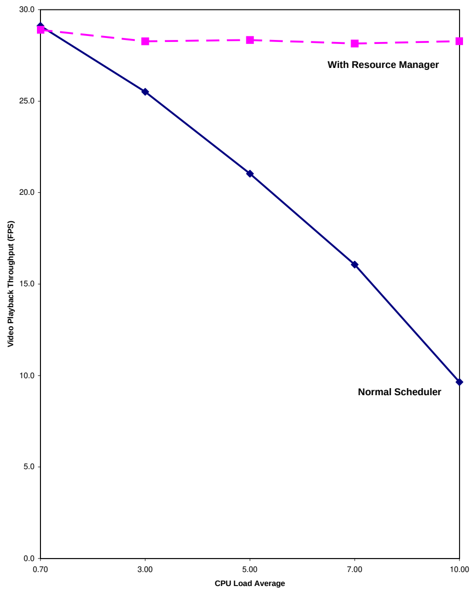

Figure 5: Video Playback Throughput Comparison

is that of requiring users to specify how to achieve quality of service expectations through low-level resource statements such as a statement of the execution rate, period, priority, deadline, time slice size, delay, packet size, or bandwidth required to meet expectations [22, 5, 26, 10, 4, 2, 20, 3, 9, 28]. Some of these ideas and concepts have been used in developing new systems such as Real-Time Mach [27]. We should note that our framework allows us to build our QoS mechanisms over any operating system. As seen in Section 7, we have implemented our framework on a Solaris platform.

8.2 Policies

The term *policy* is not unique to this work. The work described in [24] describes how to define and manage policies and analyse the policies to find conflicts. Later work [25] defines a policy specification language that we modified for our purposes. Other work includes policy distribution [19, 21]. Our work is different in the following aspects: (i) We deal with QoS requirements and not security; (ii) None of the existing work deals with fault isolation. The implication being that the action specified in the policy will be enough. This is not the case for QoS requirements.

8.3 Fault Isolation and Diagnosis

A literature survey [1, 17, 8, 11, 7, 18, 16, 15, 13] has shown that appropriate techniques exist for specific areas of fault management especially at the network layer. We use a a combination of the approaches presented in the related work, but we focus on the application layer. In addition, most of the existing work focusses on availability issues and does not specifically deal with quality of service.

8.4 Adaptable Systems

An adaptable computing system is usually defined as a computing system that changes its behaviour based on changes in its environment. Often the term applies to applications adapting to changes in the environment. This requires monitoring, agreeing on an adaptation and actually carrying out the adaptation. The work in [6] has developed an adaptive network transmission protocol and an adaptive group membership protocol. They have a of policies that is used to determine when a change of importance in the environment has occurred and what adaptive actions are to be taken. However, they do not seem to have provided a framework that accommodates changes in policies. The work in [12] looks at control systems to monitor infrastructure information systems and respond to disruptions, such as security attacks, by changing operating modes and configurations to minimise loss of functionality. Eventually, we must

Technical Report #547, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

experiment with having application adaptation when it is not possible to adapt resources. This work can contribute in this aspect.

9 Discussion

Work to date has provided a number of insights several that we will discuss in this section.
Overhead. Developers are understandably concerned about the increase in overhead resulting in a decrease in performance. From measurements taken during experimentation, the overhead from our approach is minimal. An instrumented process on our UltraSparc system requires approximately 400 microseconds more time to initialise itself and report to QoS Host Manager. If the level of quality of service delivered meets expectations, one pass through the instrumentation code requires only 11 microseconds, on average. On the other hand, whenever a quality of service violation is detected, it takes on average 474 microseconds to report the violation to our Quality of Resource Host Manager, and to have it make appropriate priority adjustments. This is acceptable since quality has already degraded at this point, and the additional time to correct the problem is necessary. We are still experimenting with actions that take place remotely. The difficulty in this measurement is the variation in the network load.
Ease of Application Development. We have instrumented several third party applications (e.g., DOOM, Apache Web Server). The only knowledge needed was the name of the probes and knowing which libraries to link in. It was not necessary for the instrumentor to have any knowledge of what sort of QoS management was taking place.

In the applications that we have instrumented, the instrumentation took little time. However, we could see that as applications grow larger it will be more difficult for the developer to determine the probe points, especially if this is decided after the application has been written. This suggests that the instrumentation be partially automated.
Ease of Developing Policies and Rules. Policies expressing user QoS requirements seem relatively easy to specify. However, the policies specifying the constraints on adaptations are much more difficult. The learning curve for developing these types of policies is high. We found that learning how to put together a set of rules is difficult and time-consuming. These rules heavily interact with each other. This makes it difficult to debug a set of rules. This is similar to existing problems in developing rule sets for expert systems. We will be examining techniques for simplifying this development.
Dynamic Rule Distribution. The ability to change rules in a QoS Host Manager is very important, especially when taking into account the difficulty in debugging the set of rules. We believe that in a real-world environment that will be impossible to always know all the dependencies or correct resource

Technical Report #547, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

allocations. Thus, it is very important to be able to dynamically add or delete rules and have this distributed to different management components at run-time. Thus, we need to have a formal way of specifying and storing rules (as we are doing for policies). We are developing a specification language (based on the one described in Section 4) for specifying rules.

There are multiple rule sets in different parts of the system. They are not independent. For example, a rule on a QoS Host Manager may specify that a message is to be sent to another management process if it is believed that a policy violation is not local to the host that the QoS Host Manager is on. There is a dependency between the rule set in the QoS Host Manager and the rule set used by the other management process. If the rule sets are changed that affect multiple management components on different hosts then we need to make sure that the changes are atomic. Otherwise, we may have management entities using sets of rules that may be inconsistent.

Structure of Quality of Resource Management. The Quality of Resource Management System in Figure 2 includes many management entities including those entities providing the services in Section 6. It also includes management entities that have rules for isolating faults and determining corrective actions. The relationship between these management entities is not clear. Should it be hierarchical or the will optimal relationship between the management entities be more arbitrary depending on organisational requirements and the relationship between different organisations.

10 Conclusions and Future Work

Our initial work has shown the feasibility of pursuing a policy-oriented approach to QoS management. Our work shows that (i) QoS management does not have to put an additional burden on application developers by forcing them to specify resource allocations in advance. (ii) Applications may be started with different QoS expectations.

We have identified a number of issues to examine in the previous section. Other future work includes the following: (i) Further developing fault diagnosis techniques (ii) We will work on supporting additional resources distributed across multiple hosts. (iii) We need to extend our work to handle overload conditions when there simply are not enough resources to meet demand. We will be looking at different application domain areas. (iv) The work we have done to date is reactive—when quality of service violations occur, we correct them. Another approach that we are investigating is proactive quality of service, where potential problems are detected and handled before they actually occur.

Technical Report #547, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

11 Acknowledgements

This work is supported by the National Sciences and Engineering Research Council (NSERC) of Canada, the IBM Centre of Advanced Studies in Toronto, Canada, Communications and Information Technology Ontario (CITO) and Canadian Institute of Telecommunications Research (CITR)

References

[1] A. Bouloutas, S. Calo, and A. Finkel. Alarm correlation and fault identification in communication networks. Technical Report TR-17967, IBM Technical Report, 1992.

[2] H. Chu and K. Nahrstedt. A Soft Real Time Scheduling Server in UNIX Operating System. *European Workshop on Interactive Distributed MultimediaSystems and Telecommunication Services*, Darmstadt, Germany, September 1997.

[3] L. Delgrossi, R. Herrtwich, and F. Hoffmann. An Implementation of ST-II for the Heidelberg Transpor t System. *Proceedings of the 1992 IEEE Globcom Conference*, Orlando, Florida, 1992.

[4] 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.

[5] P. Goyal, X. Guo, and H. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. *Proceedings of the Second Symposium on Operating System s Design and Implementation*, Seattle, Washington, October 1996.

[6] M. Hiltunen and R. Schlichting. Adaptive distributed and fault-tolerant systems. *International Journal of Computer Systems Science and Engineering*, 11(5):125–133, September, 1996.

[7] K. Houck, S. Calo, and A. Finkel. Towards a practical alarm correlation system. In *Proceedings 4th IFIP/IEEE International Symposium on Integrated Nework Management*, pages 519–530, 1995.

[8] G. Jakobson and M. Weissmann. Alarm correlation. *IEEE Network*, 1993.

[9] K. Jeffay and D. Bennett. A Rate-Based Execution Abstraction for Multimedia Comp uting. *Lecture Notes in Computer Science 1018:64-78*, 1995.

[10] 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.

Technical Report #547, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

[11] J. Jordaan and M. Paterok. Event correlation in hetergeneous networks using the osi management framework. In *Proceedings ISINM'93, IFIP TC6/WG 6.6 International Symposium on Integrated Network Management*, pages 328–340, 1993.

[12] X. Du K. Sullivan, J. Knight and S. Geist. Information survivability control systems. *21st International Conference on Software Engineering(ICSE99)*, pages 184–192, May 1999.

[13] KaGe97. A Generic Model for Fault Isolation in Integrated Management Systems . *Journal of Network and Systems Management*, 5(2), 1997.

[14] M. Katchabaw, S. Howard, H. Lutfiyya, A. Marshall, and M. Bauer. Making distributed applications manageable through instrumentation. *Computer Communications*, 45:81–97, 1999.

[15] S. Katker. A modelling framework for integrated distributed systems fault management. In *Proceedings IFIP/IEEE International Conference on Distributed Platforms*, pages 186–198, 1996.

[16] S. Katker and M. Paterok. Fault isolation and event correlation for integrated fault management. In *Proceedings of the 5th International Sypmposium on Integrated Network Management*, 1997.

[17] W. Kehl and H. Hopfmuller. Model-based reasoning for the management of telecommunication networks. In *Proceedings ICC'93 IEEE International Conference on Communications*, pages 13–17, 1993.

[18] S. Kliger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo. A coding approach to event correlation. In *Proceedings of the 4th International Sypmposium on Integrated Network Management*, 1995.

[19] T. Koch, B. Kramer, and G. Rohde. On a Rule-Based Management Architecture. *Proceedings of the Second International Workshop on Services in Distributed and Networked Environment*, Whistler, British Columbia, Canada, June 1995.

[20] C. Lee, R. Rajkumar, and C. Mercer. Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach. *Proceedings of Multimedia Japan*, March 1996.

[21] D. Marriot and M. Sloman. Implementation of a Management Agent for Interpreting Obligation Policy. In *Proceedings of the Seventh IFIP/IEEE International Workshop on Distributed Systems: Operations & Management*, pages 430–444, October 1996.

Technical Report #547, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

[22] C. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. *Proceedings of the IEEE International Conference on Mul timedia Computing and Systems*, Boston, Massachusetts, May 1994.

[23] 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.

[24] M. Sloman. Policy Driven Management for Distributed Systems. *Journal of Network and Systems Management*, 2(4), 1994.

[25] M. Sloman and K. Twidle. Domains: A framework for structuring management policy. In M. Sloman, editor, *Network and Distributed Systems Management*, pages 433–453. Addison-Wesley, 1994.

[26] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Barua h, 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.

[27] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Toward a Predictable Real-Time System . *Proceedings of USENIX Mach Workshop*, October 1990.

[28] D. Yau and S. Lam. Adaptive Rate-Controlled Scheduling for Multimedia Applications. *Proceedings of the 1996 ACM Multimedia Conference*, Boston, Massachusetts, November 1996.

Technical Report #547, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999