Issues in Managing Soft QoS Requirements in Distributed Systems Using a Policy-Based Framework

Hanan Lutfiyya, Gary Molenkamp, Michael Katchabaw, and Michael Bauer

Department of Computer Science
The University of Western Ontario
London, Ontario, Canada N6A 5B7
hanan@csd.uwo.ca

Abstract. We address the problem of Quality of Service (QoS) requirements for multimedia applications (e.g., distance education, telemedicine, electronic commerce). These applications need to be able to 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 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.
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.

Users of these applications 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 even if QoS requirements are not satisfied.

We refer to the allocation and scheduling of resources to meet QoS requirements as *QoS management*. Many QoS management techniques (e.g., [5]) 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 meet their timing constraints to avoid disastrous consequences, e.g.,

M. Sloman, J. Lobo, and E. Lupu (Eds.): POLICY 2001, LNCS 1995, pp. 185–201, 2001.

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 renegotiate a new resource usage allocation with the operating system and/or adapt its behaviour.

Much of this work assumes that the user or developer 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 least 25", but also the number of CPU cycles needed to get the 25 frames per second. The derivation of this information needs in-depth knowledge of the hardware architecture, network and system software in the target environment. This places an additional burden on the developers of these applications and in most cases is not doable without having access to the environment in which the application is being deployed.

This paper defines a framework that is *policy-driven* to address the problem of QoS management for applications that have soft real-time requirements. This strategy does not require users or application developers to have detailed knowledge of either the resources needed nor the scheduling and allocation techniques used.

This paper is organised as follows. In section 2, we describe our strategy and in section 3 we outline the elements needed to implement the strategy. Section 4 describes how policies are used in the strategy. Sections 5 and 6 present details on an architecture. Section 7 presents some details of the related work. Sections 8 describes related work. Sections 9 and 10 describe our insights and future work.

2 Strategy

The following example illustrates the relationship between QoS requirements (or user expectations) and the resources allocated using our strategy. Assume that we have 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, say this process is given a particular CPU priority among running processes. 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, i.e., the process's priority is reduced. This frees some of the CPU to be used for other processes. Otherwise, the process's priority is increased to gain more use of the CPU. Resource allocations are adjusted until a suitable one is found 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. This may not always be possible. For example, there may have been other multimedia

processes with similar QoS requirements on the same host machine and it might not be possible to set CPU priorities to satisfy the processor needs of all.

There may also be additional constraints that dictate how the system should react in such situations. Such constraints might dictate that each of these processes should have equal access to resources and, hence, allow the performance of each of the processes to degrade. On the otherhand, it may be decided that some applications have priority over the others i.e., a differentiated resource allocation is allowed. Such constraints must also be realized within the overall management of the distributed system.

3 Realizing the Strategy

In order to realize this strategy, we must be able to a) enforce the QoS requirements and b) specify QoS requirements and any additional constraints as policies.

3.1 Enforcement

The services needed to make a system comply with QoS requirements are referred to as *enforcement* services. Enforcement consists of three logical phases: detecting the violation of a QoS policy, determining the cause of the violation and taking steps to take to adapt the system to bring it back into a state of compliance.

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 characterising run-time behaviour and evaluating it relative to the QoS requirements.

Violation location takes symptoms and identifies the cause and location of the fault in the system. In the earlier example, a violation would occur if the video application was not receiving frames at a frame rate of between 23 and 27 frames per second. This could be caused by several situations, e.g., the video application might not be getting enough local processor cycles, the server process might not be 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.

Adaptation refers to the 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 process or rerouting traffic around a congested network switch. The action(s) to be taken depend not only on the cause of the violation, but also depend on the constraints imposed on how to achieve the QoS requirement. For example, one possible corrective action is to adjust the CPU priority of the video application

receiving the video stream. However, if there are several multimedia applications on the same host, then perhaps attaining the desired level of service for all is not possible and a different action is necessary, e.g. adjust the priority based on the user of the video application.

These constraints are a second category of requirements (hence referred to as administrative requirements). However, these are not user QoS requirements (hence referred to application 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 Requirement Specification and Distribution

Application QoS requirements for a particular application will change. 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. Administrative requirements will also change since the constraints on the possible adaptations will also change during the lifetime of the system.

As described earlier, an application QoS requirement may be violated. It must be possible to specify an action which in many cases will be to send a notification to another entity that is doing the diagnostics and determining the adaptation to be taken.

The implication of these two observations is that it should be possible to store both application QoS and administrative requirements. The application QoS requirements should accessible by an application when that application is started up. It must also be possible to specify the action(s) to be taken if the application QoS requirement is not satisfied.

This suggests that we allow the requirements to be expressed as *policies*. A *policy* can be defined [20] as a rule that describe the action(s) to occur when specific conditions occur. A policy for an application QoS requirement has as its condition the negation of the QoS requirement which means that the specified action(s) is to take place when it has been detected that the QoS requirement has been violated. Policies can also be used for administrative requirements (which we will mostly refer to this as *rules*).

4 Specifying Application QoS Policies

There already exists formalisms for specifying policies. Our goal is not to invent a new formalism, but rather use (as much as possible) existing formalisms. We use a formalism defined in [6].

Example 1.
An example of an application QoS policy is the following.

```
oblig NotifyQoSViolation {
    subject (...)/VideoApplication/qosl_coordinator
    target fps_sensor,jitter_sensor,buffer_sensor,(...)QoSHostManager
    on not (frame_rate = 25(+2)(-2) AND jitter_rate < 1.25)
    do fps_sensor->read(out frame_rate);
       jitter_sensor->read(out jitter_rate);
       buffer_sensor->read(out buffer_size);
       (...)/QoSHostManager->notify(frame_rate, jitter_rate, buffer_size);
}
```

The subject is the actual application that the policy applies to, since it will have responsibility for the policy (this will be made clearer in the next section). Basically, VideoApplication refers to the name of an executable, qosl_coordinator refers to an instrumentation component that evaluates the conditions stated in policies at run-time (more on this in the next section) and (...) includes other identitying information such as hostname, the application that the executable belongs to, etc; (more on this in a later section).

In Example 1, frame_rate and jitter_rate are attributes of a video application. The constraint on the frame_rate attribute is that its value must be 25 give or take a couple of frames. 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. The do component of the policy specifies which actions are to be carried out when the policy has been violated. (...)/QoSHostManager refers to the process (discussed in the next section) that receives notification of the violation. The constructs fps_sensor → read(frame_rate), jitter_sensor → read(jitter_rate), and buffer_sensor → read(buffer_size) refer to operations on sensors (explained in the next section) that monitor the frame rate, jitter rate and communication buffer sizes, respectively. The (...) notation includes other identifying information such as hostname, etc; The actions to be taken are to read the frame rate, the jitter rate and the buffer size and send this information to the (...)/QoSHostManager.

5 Enforcement Architecture

The overall architecture for enforcing QoS requirements, specifying QoS requirements using policies and distributing them is described in this section and the next section. This section describes the specific architectural components to support services needed for enforcement. The description assumes that a running application knows its QoS requirements. How it knows its QoS requirements will be discussed in the next section which focusses on the part of the architecture that deals with policy management i.e., the specification of QoS requirements using policies and their distribution.

The approach used in this work 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 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. The example policy used throughout this section is the one stated in Example 1.

We also assume that as part of the instrumentation each application will have a coordinator component. The role of the coordinator is to oversee the policies associated with the particular instantiation of the application and to communicate with a QoS Host Manager.

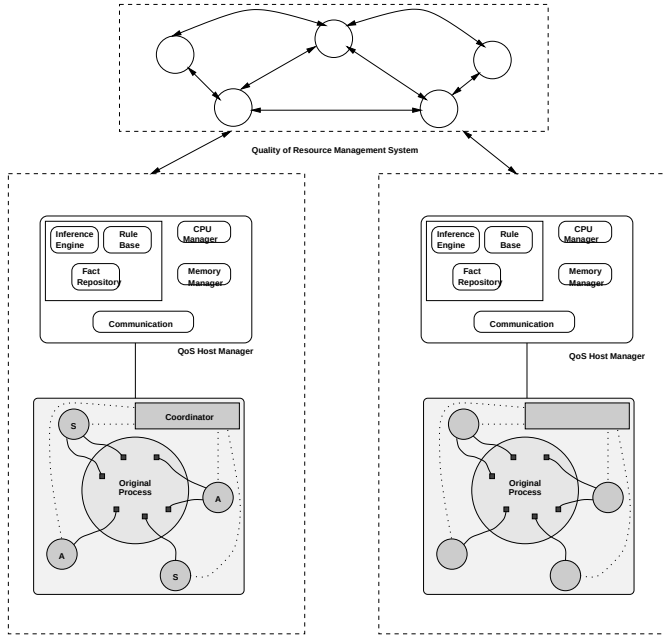

Fig. 1. Enforcement Architecture

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 in the following subsections.

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.

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 2.
As an example, consider a video playback application that has the policy stated in Example 1. This QoS requirement is translated into initial thresholds for sensor s_1, capable of measuring the frame rate. The target, upper and lower thresholds are 25, 27 and 23 frames per second, respectively. This QoS requirement also translates into an initial threshold for another 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 the constraints on the values of the frame rate are not satisfied, it will inform the coordinator *c*. s_2 has similar probes. □

How does a sensor relate to policies that express application QoS requirements? A sensor collects values for an attribute of the process. This attribute is part of a policy (or policies) being applied to the application. 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 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. An application may also have more than one policy and it is possible that these policies share attributes. We note that not all sensors measure attributes that are directly used in the specification of a policy (see Example 5).

5.2 Coordinator

The coordinator is responsible for tracking adherence to the policies associated with the application process and maintains a list of policy objects. A coordinator receives the following information for each policy: (i) The policy identifier. (ii) A **condition list** in which a condition is represented by an attribute identifier, the identifier of a sensor that monitors that attribute, a comparison operator and value that the attribute is to be compared to using the comparison operator. (iii) An **action list** in which each element of the list is a pair which represents a target object and an action to be taken on that target object. A policy, then, is represented as a conjunction or disjunction of constraints on attributes.

The coordinator takes each policy from the set of policies received and creates a policy object. For each policy, the coordinator extracts the condition list, the action list and the boolean operator. For each condition in the condition list, a variable is generated to represent the condition that must hold between an attribute identifier and the value. This is added to a boolean expression based on the boolean operator to be used.

Example 3.
The policy in Example 1 has a condition list consisting of the following component parts: $frame\_rate > 23$, $frame\_rate < 27$ and $jitter\_rate < 1.25$. Boolean variables x_1, x_2 and x_3 are generated for each one. The boolean expression is x_1 AND x_2 AND x_3. □

For the sake of simplicity, we assume that each sensor is associated with one attribute and thus only needs one init method which can be given a value and a comparison operator. The init method can take a threshold value (represented as a character string) and convert it to the appropriate type. We also assume that each sensor has a read method that returns the value of the attribute in character form. The sensor is able to do the appropriate conversion. For each comparison retrieved from the comparison list, an internal identifier generated for that comparison which was passed to the sensor using init.

The algorithm for what the coordinator does when it receives an alarm from a sensor is described as follows. 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 (based on the internal identifier generated for the comparison) to a variable that is used in the boolean expression. This variable is set to false and the boolean 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 4.
This example describes a coordinator that has the policy of Example 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 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 evaluation is false, the coordinator reads the frame rate, jitter rate and buffer size, puts them into a message report and sends this off to the QoS Host Manager. □

Example 5 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.

Example 5.
A socket provides for interprocess communication. In UNIX, a socket is a file descriptor. The kernel allocates an entry in a private table in the process area, called the user file descriptor table and notes the index of this entry. The index is the file descriptor that is returned to the process. The entry allocated is a pointer to the first inode. The operations *read()* and *write()* are done through the memory associated with the inodes of the process (which we will refer to as buffer). We can create a sensor s_3 that has a probe that given a file descriptor for a socket returns the length of the buffer. This length can be compared to a specified threshold. If the length is smaller then this threshold then this is taken to be an indication that the problem may not be local to the process. □

5.3 Quality of Service Host Manager and Domain 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 then determining 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.

Consider the policy in Example 1. One rule for the QoS Host Manager is informally stated as follows: 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.

Another rule for the QoS Host Manager is informally stated as follows: if the communication buffer size is below some threshold then send a notification to the QoS Domain Manager, which can then locate the source of the problem, perhaps by interacting with the QoS Host Managers. The implication of a small

buffer size is that the video client is able to process the received frames fast enough and that the problem is either a network problem or a server problem. The QoS Domain Manager also has a rule set which is used to drive the location process and guide the formulation of corrective actions. One such rule for the QoS Domain Manager is informally stated as follows: Upon receiving an alarm report from the client-side QoS Host Manager, ask the corresponding server-side QoS Host Manager for CPU load and memory usage. Another QoS Domain Manager rule states that if the CPU load exceeds some predefined threshold or the memory usage exceeds some threshold then an the alarm report is sent to the server-side QoS Host Manager.

This is a relatively simple set of rules[1]. A more complex set of rules would include rules that reflect administrative requirements.

6 Policy Specification and 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 polices/rules to the relevant management components. In this work, this means that policies about an application should be able to change each time the application runs and that the rules in the QoS Host Manager should also be able to change. These changes should not require recompilation (unless, of course, there is a change in the actual attributes needed). In this section, we describe the applications and services needed to support policy distribution. This is illustrated in Figure 6. The components (e.g., Policy Agent, Policy Repository, applications) in Figure 6 fit in the "Quality of Resource Management System" depicted in Figure 2. Our current implementation focusses on policies.

6.1 Information Model

We begin with a partial description of the information model used to represent the data needed to support policy distribution. We have identified the following different types of data needed. The data and the relationships between the data are described in this section.

An *executable* is instantiated on a host as a process. More information about the attributes of an executable class can be found in [14].

Sensors represent code that is instrumented into a program. A sensor collects values for an attribute of the process and is part of an executable. In the model, sensors are associated with the executable of the process. A sensor may be used for more than one executable and an executable may have more than one sensor (hence the many to many relationship). A sensor class includes a sensor identifier and a list of attribute identifiers.

An *application* ([14]) defines the application to be managed. An application is composed of at least one executable.

[1] Due to space considerations, we have not included the complete set of rules.

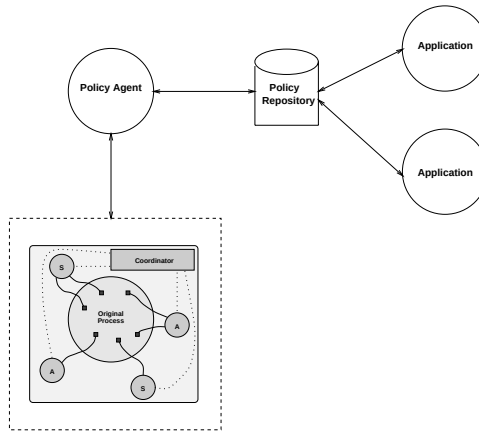

Fig. 2. Policy Distribution Architecture

A *policy* states that the application QoS requirement of an executable of an application and the actions to be taken if that QoS requirement is violated. We note that in one session an executable may have a different policy then one used in a second session simply because the application that is using that executable is different. Policies also differ depending on the user of the application. A policy is composed of policy conditions and policy actions. A *policy condition* can be reused in other policies as can a *policy action*. Hence, the reason for their separate class representation. One attribute of a policy represents how the policy conditions are to be evaluated e.g., conjunctively or disjunctively. Policies can also be subclassed e.g., QoS policies.

6.2 Description of Components

We will now describe the components needed for policy distribution.
Repository Service. The Repository Service allows for the storage and retrieval of the data specified in the previous section.
Management 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. Another management application can be used for checking information integrity e.g., an application can ensure that an application has sensors that collect the attributes specified in the policies (note that in Figure 6 the management application is just referred to as an application).
Policy Agent. This agent provides an interface that includes a method that is used by a process to register with an agent. When a process starts up, it registers with the policy agent. The process passes information about the process that is relevant in determining the policies that are applicable to that process. This includes a process identifier, an application identifier, an executable identifier

and a role identifier. The Policy Agent uses this information and maps it to the appropriate policies. This is sent to the coordinator component that creates a list of policy objects.

7 Prototype

To explore and evaluate the architecture, we developed a prototype system based on the architecture presented in this paper. This prototype has been implemented for Solaris 2.8.

All of the sensors described in Section 5 were implemented. 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 a collection of resource managers that each manage a single system resource. To date, we have resource managers capable of adjusting CPU allocations (through manipulating time-sharing priorities, or by allocating units of real-time CPU cycles) as well as memory (through adjusting the number of resident pages each process has in physical memory). The inference engine, rule set and fact repository are implemented using CLIPs [18].

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 through a QoS Domain Manager) 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).

Each of the classes defined in the information model were mapped to LDAP classes.

We have implemented a simple application that provides an interface that asks the user to input policy information based on the notation presented earlier. The possible values are retrieved from the repository server. Very simple information integrity checking is done which basically is making sure that that policy is being applied to an executable that has the sensors that can monitor the attributes specified in the policy. Another check is to make sure that the actions are either method invocations on sensors or a notification to the QoS Host Manager and that the notification is based on data returned by sensors (must be non-empty). This gets translated into an LDIF file which can be easily uploaded into LDAP.

Figure 3 compares the mean video playback throughput, in frames per second, for the MPEG video player [17] 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

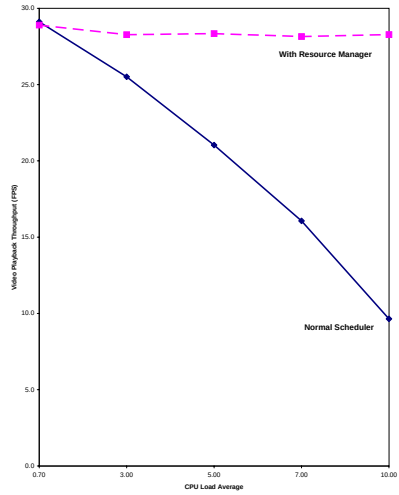

Fig. 3. Video Playback Throughput Comparison

per second – well within the acceptable limits set by the quality of service policy for the video player.

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. More detailed discussion can be found in [8, 15].

8 Related Work

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

8.1 Dynamic QoS Management

The work closes to ours is found in [1, 4] is very similar philosophically to our notion of QoS management without knowledge of detailed resource requirements. However, they have yet to address the issues related to developing an architecture that can deal with resource adjustments and fault location. There are also differences in the way CPU adjustments are done. Some work (e.g., [3]) allows for the adaptation of the application's operations (as opposed to resource adjustments) to accommodate violations of QoS requirements.

8.2 Policies

We divide our discussion into the following: Policy specification, architectures and vendor support.

Policy Specification.

IETF is currently developing a set of standards (current drafts found in [16, 19, 21]) that includes an information model to be used for specifying policies, a standard that extends the previous standard for specifying policies for specifying QoS policies and a standard for mapping the information model to LDAP schemas. Informally, speaking an IETF policy basically consists of a set of conditions and a set of actions. The standards focus on low-level details related to policy encoding, storage and retrieval. Our policy definition with its notion of reusable policy conditions and actions is very similar to that defined in the IETF standards. Our use of "userole" is similar to the intent of "role combination". One of our conclusions is that it is possible to use the draft standards for defining application QoS policies even though most of the work has been motivated by security or management of network bandwidth policies.

However, there are some problems with current policy specifications. For example, an alarm report from an instrumented process triggers one or more rules. The alarm report is an event that starts a collection of actions. Not all of these actions are necessarily taking place at one QoS Host Manager. Currently, none of the draft standard definitions nor higher level specifications of policies like [6] make it easy to represent this. We are intrigued by the suggestions made in [13] and we will be examining this work in more detail.

Architecture. The IETF has also defined a general architecture (a good description can be found in [20]) for a policy management system. The general architecture is very high-level and there is nothing to suggest that our more specific architecture does not fall within the proposed general architecture.

Vendors. Many network vendors such as 3Com, Cisco Systems, Lucent and Nortel are offering "primitive" versions of policy-based network management systems for control of their network devices.

8.3 Fault Isolation and Diagnosis

A literature survey (which includes [7, 2, 12, 11, 9, 10]) has shown that appropriate techniques exist for specific areas of fault management especially at the network layer. We use 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.

9 Discussion

Work to date has provided a number of insights several into QoS management. that we will discuss in this section.

Changing QoS Requirements. The separation of specifying QoS requirements using policy formalisms from applications allows for the following: (i) The use of the "UserRole" allows for different users to have different QoS requirements for the same application.(ii) Although not described, the sensors provide an interface that allows for a threshold to be changed. Thus, we are able to change QoS requirements while an application is executing.

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 application QoS requirements seem relatively easy to specify. However, the administrative policies (or rules) 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 and evaluation. As discussed in the Related Work section, we also had a difficult time with the specification of rules. This is a subject for future work.

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 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.

Interconnecting QoS Domain Managers. A QoS Domain Manager is primarily responsible for locating sources of problems involving applications distributed across multiple hosts, as well as determining actions required for solving the problem. Each QoS Domain Manager is assigned to a collection of hosts (its *domain*); it interacts with QoS Host Managers on these hosts to enforce policies as necessary. At times, problems may span multiple domains, which introduces several interesting issues, as the relationship between these management entities is not clear. Should it be hierarchical or the will optimal relationship between the managers 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 and the configuration services needed to support these 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.

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] G. Beaton. A Feedback-Based Quality of Service Management Scheme. In *HIPPARCH Workshop, Uppsala*, June 1997.

[2] A. Bouloutas, S. Calo, A. Finkel, and I. Katzela. Distributed Fault Identification in Telecommunication Networks. *Journal of Network and Systems Management*, November 1995.

[3] S. Brandt, G. Nutt, T. Berk, and M. Humphrey. Soft Real-time Application Execution with Dynamic Quality of Service Assurance. In *Proceedings of the 6th IEEE/IFIP International Workshop on Quality of Service (IWQoS '98)*, pages 154–163, 1998.

[4] H. Cho and A. Seneviratne. Dynamic QoS Control without the Knowledge of Resource Requirements. *Submitted to IEEE Transactions on Computing*, 1999.

[5] 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.

[6] N. Damianou, N. Dalay, E. Lupu, and M. Sloman. Ponder: A language for specifying security and management policies for distributed systems: The language specification (version 2.1). Technical Report Imperial College Research Report DOC 2000/01, Imperial College of Science, Technology and Medicine, London, England, April 2000.

[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] M. Katchabaw, H. Lutfiyya, and M. Bauer. Driving resource management with application-level quality of service specifications. *Journal of Decision Support Systems*, 28:71–87, 2000.

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

[10] S. Katker and H Geihs. A Generic Model for Fault Isolation in Integrated Management Systems. *Journal of Network and Systems Management*, 1997.

[11] 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.

[12] 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.

[13] M. Kohli and J. Lobo. Policy based management of telecommunications systems. *1st Policy Workshop, HP Labs, Bristol*, November 1999.

[14] H. Lutfiyya, A. Marshall, M. Bauer, W. Powley, and P. Martin. Configuration maintenance for distributed application management. *Journal of Network and Systems Management*, 8(2):219–244, 2000.

[15] G. Molenkamp, M. Katchabaw, H. Lutfiyya, and M. Bauer. Managing soft qos requirements in distributed systems. *Accepted to Appear Multimedia Systems Workshop (ICPP)*, August, 2000.

[16] B. Moore, J. Strassmer, and E. Elleson. Policy core information model – version 1 specification. Technical report, IETF, May 2000.

[17] 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.

[18] G. Riley. Clips: A tool for building expert systems. Technical report, http://www.ghg.net/clips/CLIPS.html, 1999.

[19] Y. Snir, Y. Ramberg, J. Strassner, and R. Cohen. Policy framework qos information model. Technical report, IETF, April 2000.

[20] Startdust.com. Introduction to qos policies. Technical report, Stardust.com, Inc., July 1999.

[21] J. Strassner, E. Ellesson, B. Moore, and Ryan Moats. Policy framework ldap core schema. Technical report, IETF, November 1999.