Managing Soft QoS Requirements in Distributed Systems

Gary Molenkamp, Michael Katchabaw, Hanan Lutfiyya, and Michael Bauer
Department of Computer Science
The University of Western Ontario
London, Ontario, Canada N6A 5B7
gary,katchab,hanan,bauer@csd.uwo.ca

Abstract

We address the problem of soft 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.

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 meet 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 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 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 require users or application developers to have detailed knowledge of the resources needed and resource scheduling and allocation techniques in use.

This paper is organised as follows. In section 2, we describe our strategy. Section 3 describes the architecture needed to support the strategy described in Section 2. Sections 4 and 5 describes our prototype and experiments. Section 6 describes related work. Sections 7 and 8 describe our insights and future work.

2 Strategy

Our approach focuses on using user or developer expectations as operational run-time requirements that

Appeared in the Proceedings of the Multimedia Systems Workshop (ICPP), Toronto, Canada, August 2000

can be captured in the form of *policies* which state the specific constraints of the desired QoS behaviour of an application or collection of applications. For example, the developer of an application with a video component might identify a policy such as: "The number of video frames per second displayed to the user must be at least 25" as opposed to specifying the number of CPU cycles needed to get the 25 frames per second. We dynamically adjust resources as needed by the applications. This requires the ability to detect that a QoS requirement is not being satisfied, determine why it is not being satisfied and then determine corrective adaptive step(s). This is referred to as *policy enforcement.*

Assuming that we have a policy that states "A given video application is to deliver video at a frame rate of 25 frames per second, plus or minus 2 frames", we can illustrate the steps of policy enforcement as follows.

Violation Detection. Violation detection occurs when 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.

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.

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 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 (policies). 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 Architectural Design

To support our strategy, we developed the architecture shown in Figure ??. This architecture is derived from our earlier work in application instrumentation [?]. 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 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 following example policy is used throughout this section to illustrate concepts.

Example 3.1. An example policy that defines a QoS requirement can be 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". □

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

3.1 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.

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

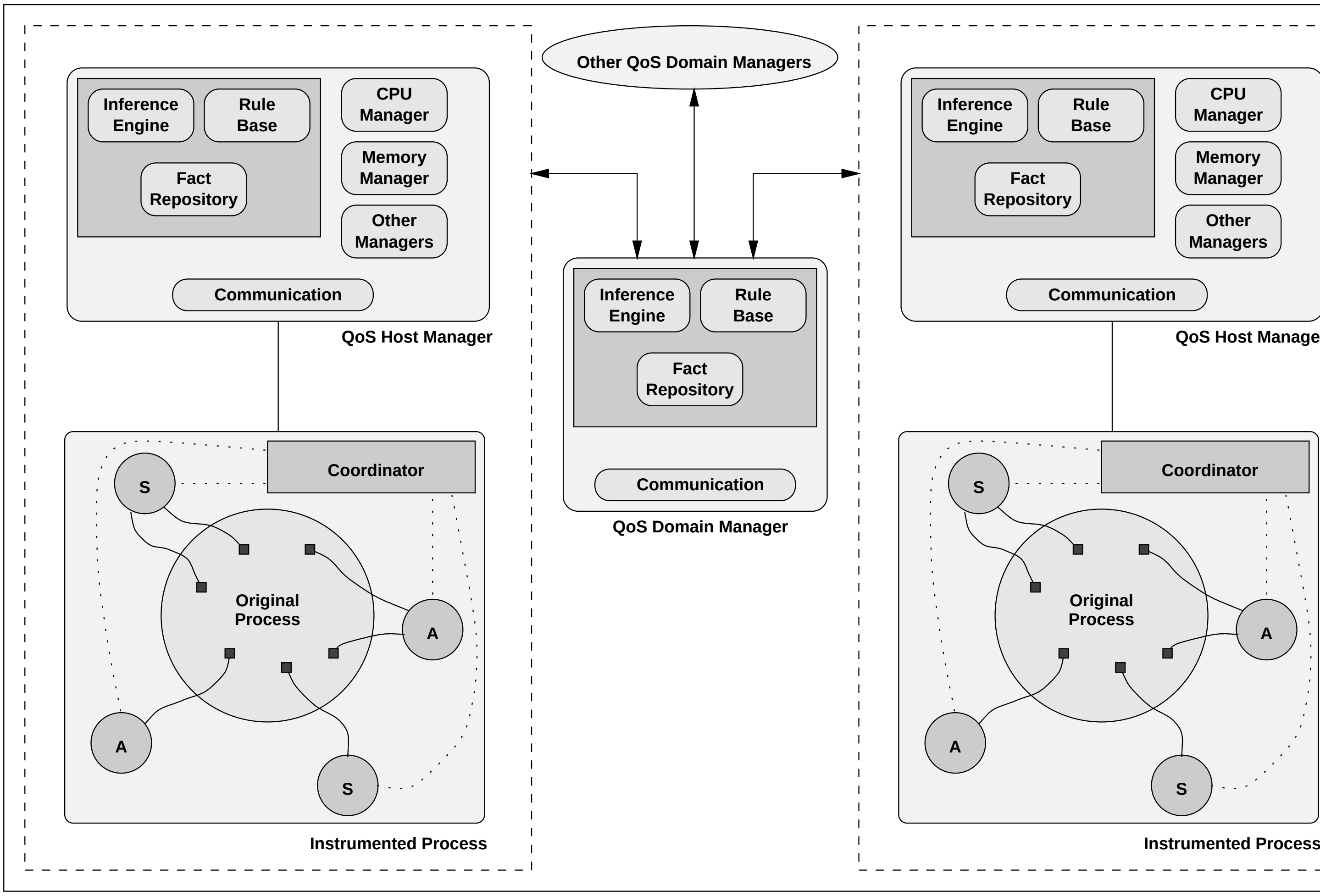

Figure 1: Application-Driven Resource Management Architecture

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 3.2. As an example, consider the sample pseudo-code for a video playback application in Figure ?? that has the policy stated in Example 3.1. 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 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 there is a problem, it will inform the coordinator c. s_2 has similar probes. □

How does a sensor relate to policies that express application-level 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. 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 3.1, 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 have more than one policy and it is possible that these policies share attributes. Hence, a sensor may provide values for more than one policy. We note that not all sensors measure attributes that are directly used in the specification of a policy (see Example 3.4).

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 constraints of the attributes of a policy. 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 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 3.3. This example describes a coordinator that has the policy stated in Example 3.1. 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 3.1. 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 in Example 3.1 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 reads the frame rate, jitter rate and buffer size, puts 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_1, s_2 with e.
while (v **not done**) **do:**
    Retrieve next video frame f.
    Decode and display f.
    Execute $s_1 \rightarrow$ *probe*, $s_2 \rightarrow$ *probe*
endwhile

Figure 2: Instrumentation Example

Example 3.3 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 3.4. 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. □

3.2 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 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 3.1. 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.

One rule for the QoS Host Manager can be 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 can be 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 through 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. We have begun to look at such rules in the context of electronic commerce applications.

4 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. 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 3. 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. 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).

5 Experimental Results

To demonstrate the utility of our work, we have instrumented a client-server MPEG video-on-demand system. In this system, clients connect to the video server, selects the video to watch, and receive streamed video

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

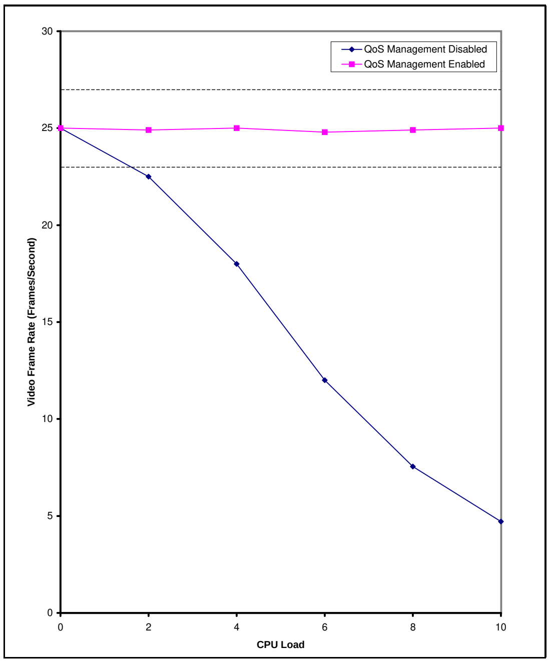

Figure 3: Video-On-Demand Throughput

to display to their users. To monitor delivered quality, we added a sensor to measure its video playback throughput, in frames per second. The server was executed on an UltraSparc 5 workstation, with a client executing on an UltraSparc 1 workstation, connected by 10 megabit switched Ethernet. Both workstations run the Solaris 2.6 operating system, with our quality of service management system for support.

We instructed the client to select a 320×240 8-bit video file from disk, and we gave it a quality of service policy dictating that it play the video with a mean frame rate of 25 frames per second, allowing for a deviation of 2 frames per second from this target (giving a range of 23 to 27 frames per second). Our experimentation consisted of executing the instrumented client several times under various constant processor loads generated by a CPU load generator (for example, generating a CPU load of 5.0 implies that there is a mean run-queue length of 5.0). These experiments were repeated with our quality of service management enabled and disabled, and replicated five times for statistical confidence.

Figure ?? compares the mean video playback throughput, in frames per second, for the video client with quality of service management enabled and disabled. From this figure, we can see that video throughput dropped dramatically under an increasing CPU load when our managers were disabled. With our managers in place, however, throughput remained reasonably consistent around 25 frames per second – well within the acceptable limits set by the quality of service policy for the video player. The problem was correctly identified, located and dealt with by the rules given to our managers.

Similar experiments have been carried out stressing other resources, such as memory, and with other applications, including a video conferencing system, and an electronic commerce application. Experiments in these cases produced similar results. [2]

6 Related Work

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

6.1 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 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 [?, ?, ?, ?, ?, ?, ?, ?, ?, ?]. Some of these ideas and concepts have been used in developing new systems such as Real-Time Mach [?]. We should note that our framework allows us to build our QoS mechanisms over any operating system. As seen in Section 4, we have implemented our framework on a Solaris platform.

6.2 Policies

The term *policy* is not unique to this work. The work described in [?] describes how to define and manage policies and analyse the policies to find conflicts. Later work [?] defines a policy specification language that we modified for our purposes. Other work includes policy distribution [?, ?]. 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.

6.3 Fault Isolation and Diagnosis

A literature survey [?, ?, ?, ?, ?, ?, ?, ?] 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.

[2]Due to space considerations, we have not included these results. These results can be made available upon request.

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

7 Discussion

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

Overhead. 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.

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 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 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.

8 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.

9 Acknowledgements

This work is supported by the National Sciences and Engineering Research Council (NSERC) of Canada, the

IBM Centre for Advanced Studies in Toronto, Canada, Communications and Information Technology Ontario (CITO) and Canadian Institute of Telecommunications Research (CITR).