Diagnosing Quality of Service Faults in Distributed Applications

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

Abstract

QoS management refers to the allocation and scheduling of computing resources. Static QoS management techniques provide a guarantee that resources will be available when needed. These techniques allocate resources based on worst-case needs. This is especially important for applications with hard QoS requirements. However, this approach can waste resources. In contrast, a dynamic approach allocates and deallocates resources during the lifetime of an application. In the dynamic approach the application is started with an initial resource allocation. If the application does not meet its QoS requirements, a resource manager attempts to allocate more resources to the application until the application's QoS requirement is met. While this approach offers the opportunity to better manage resouces and meet application QoS requirements, it also introduces a new set of problems. In particular, a key problem is detecting why a QoS requirement is not being satisfied and determining the cause and, consequently, which resource needs to be adjusted. This paper investigates a policy-based approach for addressing these problems. An architecture is presented and a prototype described. This is followed by a case study in which the prototype is used to diagnose QoS problems for a web application based on Apache.

Key Words: Quality of Service (QoS), Policies, E-commerce, Apache

1 Introduction

An increasing number of distributed applications involve data or processing which is time-sensitive – such as video-on-demand, distance eduction, telemedicine, teleconferencing and electronic commerce applications. These applications typically involve some form of client-server communication. Application components might consist of several processes running on multi-purpose servers as well as client components co-existing with other applications on general-purpose workstations. Information is exchanged across several networks, including, possibly, the Internet. Users expect a high level of quality of service (QoS) from many of these applications. By quality of service, we are referring to non-functional, run-time (or operational) requirements, such as performance or availability requirements. An example of a QoS requirement for a multimedia application that receives a video stream might be: "The number of video frames per second displayed to the user must be at least 25 plus or minus 2 frames". Another example of a QoS requirement for an electronic commerce application that has a web server as the front end might be: "The time it takes to service an HTTP request should be less than 1.5 seconds". These are examples of *soft* QoS requirements in that these applications are still considered functionally correct if the QoS requirement is not satisfied. This is opposed to *hard* QoS requirements where applications, e.g., flight control systems, chemical process control systems, patient-monitoring systems, must meet QoS requirements to avoid disastrous consequences.

The allocation and scheduling of resources is referred to as *QoS management*. Many QoS management techniques (e.g., [8, 9, 11, 14, 21, 10, 20, 12, 6] provide a guarantee that resources will be available when needed by statically allocating resources based on worst-case needs. This is particularly important for applications that have hard QoS requirements. Some techniques provide for soft QoS requirements by initially allocating resources in a similar fashion, often based on optimistic estimates of resource needs and assuming that the resources 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. In either case, much of the work is based on the assumption that the user is aware in advance of the resource needs of the applica-

Appeared in Proceedings of the 21st IEEE International Performance, Computing, and Communications Conference, Phoenix, Arizona, April 2002

tion. Moreover, much of this work focusses on either the network or host resources to satisfy QoS, but does not look at the combination of the network and the host resouces. That is, they do not address *end-to-end* QoS. Managing QoS for a distributed application requires an end-to-end approach where possible.

For many applications these techniques are not appropriate. First, resource allocation based on worst-case needs could lead to the allocation of resources that are wasted. This suggests that it must be possible to adapt resource allocation and/or application behaviour. Second, determining precise resource needs requires intricate knowledge of the hardware architecture, network and system software in the target environment, which is distributed and heterogeneous. This tends to be very difficult in practice. Users and application developers cannot be expected to have this knowledge. Third, as noted, end-to-end solutions are needed; it must be possible to determine what particular resource allocation is inadequate, e.g., the problem could be that there are not enough local CPU cycles, it could be network congestion, or it could be a lack of cycles on the server.

In contrast, a dynamic approach allocates and deallocates resources during the lifetime of the application. Unlike the static approach, resources are not always requested from a resource manager when the application starts. If the application does not meet its QoS requirements, a resource manager would attempt to allocate more resources to the application until the application's QoS requirement is met. If the application exceeds its QoS requirments, the resource manager could deallocate resources from the application in order to provide these resources to other applications. This approach does not guarantee that the resources will be available, but attempts to best utilise resources to meet the QoS needs of the applications. However, the use of dynamic allocation introduces new problems:

1. The application needs to be monitored in order to determine if the QoS requirements are being met. While some of this can be done from outside of the application (i.e. bandwidth), most monitoring needs to be done from within the application (e.g. buffer levels, service rates). Determining that an application is not meeting its QoS requirements is referred to as *fault detection*.

2. Once a performance degradation has been detected by the monitoring system, the cause of the degradation must be determined. *Fault diagnosis* techniques hypothesise as to a probable cause of the degradation, with subsequent testing used to determine the validity of the hypothesis.

3. The resource allocation must be adjusted so as to correct the diagnosed fault. The system *adapts* the allocation of resources to achieve the required QoS. In some cases, if it is not possible to adapt the allocation of the resource, the behaviour of the application adapts to the resource allocation.

For instance, our e-commerce application might fail to meet a QoS requirement specifying response time per request. Failure to do so might occur for a number of reasons: too few web server processes processing requests, the database is not getting enough CPU cycles or enough pages or the network between the web server and the database is congested.

This paper presents an approach for detection and diagnosis of QoS faults and experience with a prototype system. The approach is policy-based and enables dynamic control of resources for providing QoS.

2 Policies

We make the following two observations. First, application QoS requirements for a particular application will change. For example, the requirements may very well depend on the user who invoked the application or may change over time, such as the time to service an HTTP request. Second, a QoS requirement may be violated. It must be possible to specify an action, that in many cases, will be to send a notification to another entity that is doing the diagnosis and determining the adaptation to be taken.

This suggests that we allow the requirements to be expressed as *policies*. Broadly, a *policy* can be defined [19] as a rule that describes 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 the QoS requirement has been violated. This separates what is to be done from the mechanisms to perform the actions. There already exists formalisms for specifying policies [16, 17, 7]. Our goal is not to invent a new formalism, but rather use (as much as possible) existing formalisms.

As noted, application QoS requirements are really non-functional requirements of the run-time behaviour of the application, e.g., performance or availability requirements. This run-time behaviour can be characterised by several attributes of the application. An example of an application QoS policy is the following, using the notation presented in [7].

Example 2.1

```
POLICY apache_service_rate {
MODE      0
SUBJECT   (...)/ApacheApplication/qosl_coordinator
EVENT     service_rate>0.015, service_rate<0.0
ACTION    service_rate_sensor->read(out sensor_rate);
 service_rate_sensor->read(out complexity);
 (...)/QoSHostManager->notify(sensor_rate,complexity);
DESCRIPTION "monitor apache service rate
          with complexity adjusting"
}
```

The subject is the application component responsible for the policy. For example, in our architecture, the qosl_coordinator is an instrumentation component embedded within the application that evaluates policies at runtime. The ellipsis in the SUBJECT field denotes other identifying information, such as hostname, etc. The rate at which requests are serviced is represented by the attribute service_rate which is an attribute of the Apache Web server. The constraint on the service_rate is that it must take less than 1.5 seconds to process a request. If this is not the case, then the ACTION field 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 service_rate_sensor→read(out sensor_rate) and service_rate_sensor→read(out complexity) refer to operations on sensors (sensors are instrumentation components; the meaning of the attributes will be described in more detail later). The actions specify that the values of the attributes specified in the read operations on sensors are to be read and sent to the (...)/QoSHostManager.

We should note that there are also constraints on adaptations allowed. These are a second set of requirements that we call *administrative* requirements. These can also be expressed using sets of policies; see [15] for a further discussion of these types of policies.

3 Architecture

The overall architecture for enforcing QoS requirements is described in this section. The description assumes that a running application knows its QoS requirements (for more details on this see [15]).

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 data that can be collected and control available.

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 (as shown in Figure 1) of the application and to communicate with a QoS Host Manager.

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. This instrumentation is provided by several components.

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.

Probes. Probes are embedded in process code to facilitate interactions with sensors and actuators. Probes allow sensors to collect data 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 defined as classes. Probes can either be methods of the sensors and actuators or be functions that call these methods.

Example 3.1
As an example, consider a web server that has the policy of Example 2.1. This QoS requirement is translated into initial thresholds for sensor s_1, capable of measuring the service rate.

Sensor s_1 includes at least the following 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 is inserted in the part of the code where a request is received. (3) A probe that does the following when a request has been processed. (i) Determines the elapsed time between arrival of the request and the finished processing of the request. (ii) Checks to see if this falls within a particular range defined by the lower and upper acceptable thresholds, filtering out unusual spikes. (iii) If the constraints on the values of the service rate are not satisfied, it will inform the coordinator c.

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

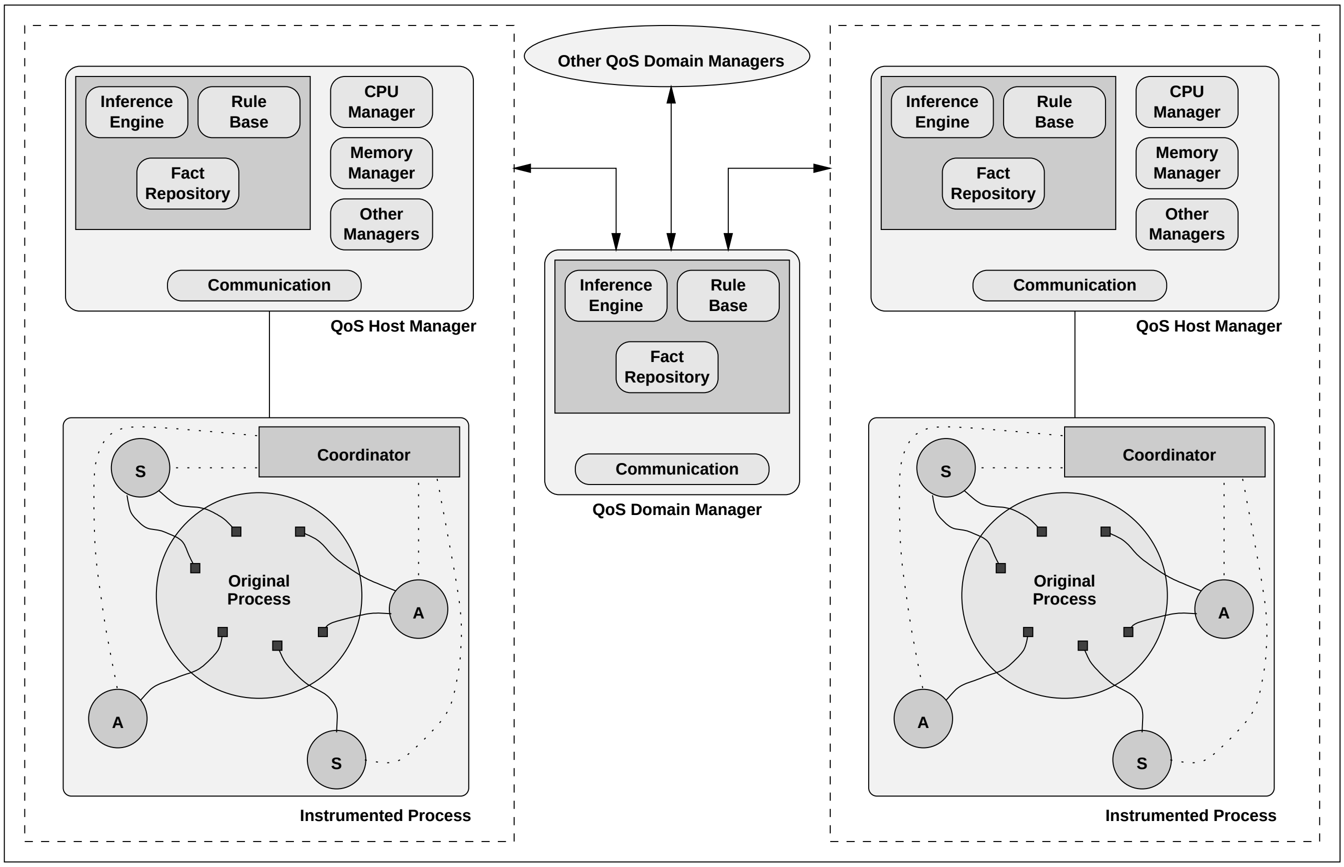

Figure 1: Enforcement Architecture

that attribute is violated, it reports this to the coordinator as an *alarm report*. For Example 2.1, it is assumed that one sensor is needed for measuring the service rate.

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. Sensors may also measure attributes that do not occur explicitly as part of a policy specification. These attributes may be used as an intermediate result.

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:

- The policy identifier.
- 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.
- 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 boolean expression (a conjunction or disjunction) of constraints on attributes.

If the coordinator receives an alarm report from a sensor, it determines which policy objects 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.

3.2 Quality of Service Host Manager and Domain Manager

The QoS Host Daemon listens for violation reports, determines the cause of the violation (if its a local resource problem) and takes corrective action by invoking the appropriate resource manager. It uses three main components for fault location: a fact base, an inference engine and a rule base. The fact base includes registration information about the managed process environment, the policy under examination and the data that is provided by the violation report. The inference engine is an expert system that applies the fact base to the loaded rule set. The rule set is key in diagnosing the cause of a violation. The adaptation of resources are manipulated within the QoS Host Daemon through the use of a generic interface. Each resource manager (or adapter) has this interface.

4 QoS Management Prototype

To evaluate the approach, 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 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) and adjusting memory allocations (through adjusting the number of resident pages each process has in physical memory). The QoS Host Daemon was implemented to permit multiple violations reports to be examined concurrently. The inference engine, rule set and fact repository are implemented using CLIPS [18]. There is 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 adjust CPU and memory allocations where appropriate.

5 Case Study using Apache

This section describes a case study of using the prototype QoS management system for an electronic commerce (e-commerce) application, which was an electronic bookstore, with the policy described in Example 2.1. E-commerce applications commonly consist of a web server that uses application programs (implemented as CGI scripts or Java applets) that may have access to one or more databases. E-commerce applications are typically multi-tiered, heterogeneous and often co-exist with other applications on large servers. The database is likely on another server and access may traverse several intervening networks. Browsers are used to access the web server and, of course, there can be many in diverse locations.

A test e-commerce application is depicted in Figure 2. It consists of an Apache web server, PostgreSQL database and client emulators. The Apache web server and the PostgreSQL database were placed on different machines. The Apache web server used CGI scripts or PHP3 for accessing the PostgreSQL database.

Policies were made available to the Apache web server process through the use of a file. The policy is parsed by the instrumentation coordinator.

5.1 Issues in Instrumentation

The policy described in Example 2.1 states a requirement on the rate that requests are processed. However, different requests need different processing times e.g., the time to complete a request for a 300 kilobyte file would be longer than the time needed to complete a request for a 3 kilobyte file. One approach is to have individual sensors and policies (stating a requirement for a particular type of request) for each type of HTTP request. This increases the complexity of the instrumentation, adds to the overhead, makes the instrumentation less scalable, and requires more effort to prevent policy conflicts. The approach taken is based on measuring the complexity of requests. This measurement is used to adjust the threshold specified in the policy. The complexity of a request is based on factors such as the file size of the HTML page request or the MIME type of the request (e.g., static, PHP3, CGI). An average complexity value is computed over a time period and is applied to the threshold specified in the policy using the following formula:

$$\text{new threshold} = \text{policy specified threshold} \;/\; (\,1 - \text{monitored complexity value})$$

A complexity of 0 does not alter the threshold, and a complexity of .9 alters the threshold by a factor of 10. A complexity of 1 is undefined. The new threshold is compared to the value of the reported sensor to determine if the monitored value is within policy limits. If the value remains within the threshold, no violation occurs. There are two approaches to providing and using complexity values. In the first approach, the sensor measuring the frame rate also measures and averages the complexity sensor. If the frame rate doesn't satisfy the threshold values specified in the policy then the average complexity value is used to determine a new threshold. If the frame rate doesn't satisfy that threshold then a violation report is issued, otherwise, it is assumed that there are no problems. In the second approach, there is an additional sensor (a *complexity* sensor) that determines the complexity of each request and determines the average. When the sensor measuring the frame rate determines that the frame rate doesn't satisfy the condition specified in the policy then the frame rate and a reading of the complexity sensor is sent to the QoS Host Manager which applies the complexity value to the threshold specified in the policy using the above formula. The first approach is more efficient, but the second approach allows the sensors to be more reusable. We currently have an implementation based on the second approach.

The Apache process is a client of the database server process. This means that the Apache process can be impacted by the performance of the database server process. Hence, the database server must be instrumented to assist in the diagnosis of a QoS violation. The PostgreSQL database was therefore instrumented to gather service times based on its ability to service requests. These service times were also adjusted through the use of a complexity sensor based on the complexity of the SQL request.

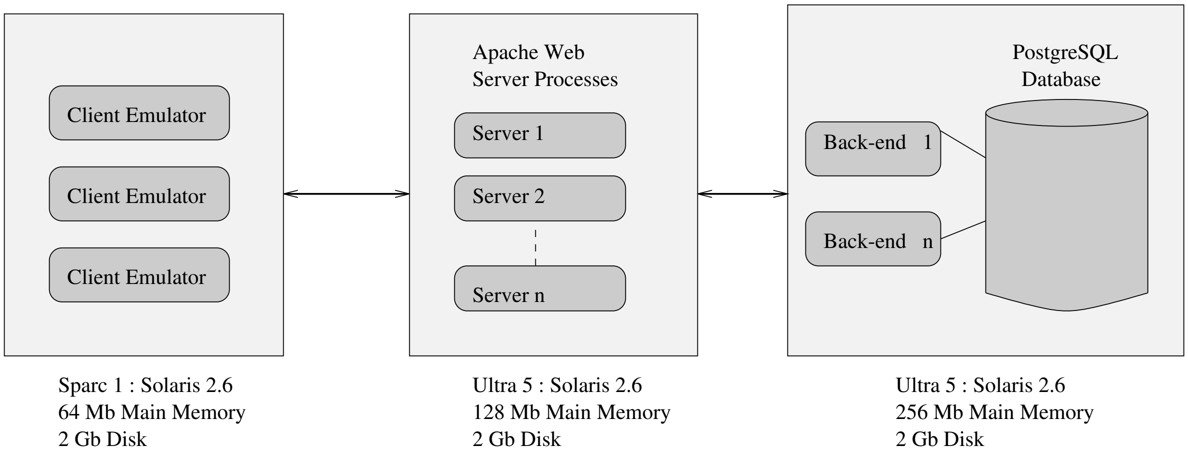

Figure 2: E-commerce Testbed

In PostgreSQL, an SQL query is passed through a planning stage, which creates one or more plans to execute the request. Each plan has a cost associated with the plan, and the optimiser selects the plan with the lowest cost. This cost is used by the complexity sensor to select the complexity value to be used by that request.

5.2 Developing Rules for Diagnosis

We developed a set of rules to be used in diagnosing problems. The rules are briefly summarised in Table 1. We conducted a number of controlled experiments in which CPU, memory, and disk resource loads were varied. This helped us identify resource dependencies of the application which were used to prioritise the resource requirements of the application based on the percentage of contribution to the affected QoS metric.

	If	*Then*
1	Service_time_high(web_pid, x), DataReferrals(<= 10%), I/O_available(web_pid)	Increase_I/O(web_pid)
2	Service_time_high(web_pid, x), DataReferrals(<= 10%), I/O_unavailable(web_pid)	Action(adjust apache config)
3	Service_time_high(web_pid, x), DataReferrals(> 10%)	Forward_Violation(V)
4	Service_time_high(web_pid, x), Service_time_high(database_pid, x), I/O_available(database_pid)	Increase_I/O(database_pid)
5	Service_time_high(web_pid, x), Service_time_high(database_pid, x), CPU_available(database_pid)	Increase_CPU(database_pid)
6	Service_time_high(web_pid, x), Service_time_low(database_pid, x)	Network_Cause( )

Table 1: Inference Engine Rules for the e-commerce Application

The rule presented in Row 1 states that if the Apache service time is high, (Service_time_high(web_pid, x)) and the occurrence of database accesses is less than 10 percent (DataReferrals(<= 10%)) and there are I/O resources (this refers to either disk or network bandwidth) to spare, (I/O_available(web_pid)), then increase I/O resource allocation (e.g. either disk or network bandwidth) to the Apache process that caused the policy violation. A low value of DataReferrals implies that most HTTP requests are requests for static pages. The results of the resource experiments we conducted indicate that the most likely cause of the variation is the Apache host's I/O allocation. I/O resource allocations account for 12.50% of the variation to the Apache service time when we only examine resources on the Apache web server host.

For rule 2, the conditions are similar to those in rule 1, but I/O resources are not available. The inference engine has determined at this point that it is not possible to adjust the system any further to satisfy the QoS requirements. We must then attempt to adjust the application so as to meet the QoS requirements with the resources currently available. This allows the management system to attempt to provide the specified QoS under overload conditions. For the e-commerce application, we instruct the Apache server to adjust its own internal configuration (Action(adjust apache config)). For example, the Apache web server could alter its QoS requirements by changing to a set of HTML files which are smaller, have fewer images or images of lower quality.

The rule in row 3 states that if the database referral occurrence is greater than 10 percent (DataReferrals(> 10%)), then the violation report is forwarded to the Global QoS Daemon, since the allocation of resources to the web server will not increase database server performance or the performance of the network between them.

The rule in row 4 is a global rule which uses both the service time of the web server (Service_time_high(web_pid, x)) and the service time of the database server (Service_time_high(database_pid, x)). When both are high and I/O resources are available, the I/O resource allocation to the database server (Increase_I/O(database_pid)) is to be increased.

Row 5 shows a rule similar to row 4 with the exception that we are adjusting CPU resources.

If the service time is high on the web server but low on the database server, the rule in row 6 states that there must be an allocation that must be done on the network between the two services (Network_Cause( )).

We note that the resource adjustments described here are rather simplistic since our current goal is to focus on diagnosis. Currently, but not presented here, we are working on more sophisticated rules for adjustment.

5.3 Experiments

A set of experiments was executed to evaluate the prototype's ability to correctly detect QoS policy violations and diagnose the cause of that violation.

The experiments were run in the environment depicted in Figure 2. Each experiment was replicated three times for statistical confidence.

Two separate sets of experiments were conducted to test the correctness of the management system's diagnosis. For each experiment, a set of web request were sent to the e-commerce application while artificial loads were imposed on either the Apache host or the PostgresSQL host. All occurances of QoS violations and the subsequent diagnosis, a fault on the Apache host or a fault on the database host, by the management system were recorded.

A summary of the results of the experiments are presented in Table 2. In the experiments in row labelled "Apache Host I/O" the I/O resources of the Apache web server artificially increased from 0 % to 90%. A total of 5 experiments were run and in each experiment 460 requests from the clients were processed. On average, a total of 70 QoS violations were detected. Of these, approximately 90% were correctly diagnosed as faults with Web I/O resources.

The experiments in row labelled "PostgresSQL Host I/O" were executed in a similar fashion, but with the load on the I/O resources of the database server. The same number of experiments were run with the same requests. On average, a total of 5.2 QoS violations were detected and approximately 90% (4.6) were correctly diagnosed as faults with the database resources.

To increase the I/O load on a host, we created a utility which performs I/O requests. The utility allows us to specify parameters, such as the data block size for reading and writing. We used this utility to obtain an I/O subsystem utilisation of 90%.

Injected Fault	Web resources	Database Resources
Apache Host I/O	89.0%	10.9 %
PostgreSQL Host I/O	11.5%	88.5 %

Table 2: Apache Diagnosis results

As seen from Table 2, when a resource load is artificially increased, the prototype is able to determine the most likely location of the policy violation.

6 Related Work

6.1 QoS Management

The little work in dynamic QoS management includes [3, 5]. This work 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., [4]) allows for the adaptation of the application's operations (as opposed to resource adjustments) to accommodate resource allocations.

6.2 Policy Architecture

The IETF has also defined a general architecture (a good description can be found in [19]) for a policy management system that includes a policy console (for entering policies), a policy repository, a policy decision point (PDP) and a policy enforcement point (PEP). The policy enforcement point is said to enforce policy while the PDP makes decisions based on policies it retrieves from the policy repository. The suggestion for the repository is to use the Lightweigth Directory Access Protocol (LDAP). 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.

6.3 Fault Isolation and Diagnosis

A literature survey (which includes, but is not limited to [13]) 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.

6.4 Web Servers

Work in using policies to manage web servers can be found in [2]. In this work, the authors decompose a web site into its component services and apply policies to specifying the desired behaviour of the component services. The relationship between the component services is abstractly represented by a tree. Since each service has a policy associated with it, each node of the tree has a policy associated with it. A graphical tool can be used to browse the tree and its policies. The authors claim that the diagnosis algorithm found in [1] can be applied to the tree structure, however, there is no presentation of the results of doing so. The diagnosis algorithm in [1] focusses on availability issues. QoS requirements need a focus on the relationship between the services and the computing resources which is what our work does.

7 Conclusion

We have presented an architecture that enables dynamic control of resources for the purpose of ensuring QoS with a specific focus on diagnosing the cause of the QoS violation. Policies were used to specify a set of QoS requirements. This allows us to separate the specification of

QoS requirements from an application and enables us to change QoS requirements while an application is executing. This provides a good deal of flexibility in managing applications over time.

Although our fault diagnosis approach is similar to that in other work, we apply the approach to determine overloaded resources among a non-trivial set of software and hardware components with complex relationships. We also have developed an integrated architecture for detection, diagnosis and adaptation.

We have identified a number of areas for future work. (i) More sophisticated fault diagnosis techniques are needed to support this approach. (ii) We need the means to handle additional resources, e.g. network bandwidth, memory, on hosts and the means to analyze them across multiple hosts. (iii) The work we have done to date is reactive – when QoS violation occur, we correct them. Another approach that we are investigating is a proactive one where potential problems are dected and handled before they actually occur. (iv) The development of tools to aid in determining rules. (v) More work is needed on the ways and means of adapting applications to change their behaviour to accommodate changing QoS requirements.

References

[1] A. Baldwin, C. Bartolini, G. Di Vitantonio, and K. Eshghi. "a novel algorithm for fault diagnosis in internet-based services". *4th Workshop of the HP OVUA, Madrid Spain*, 1997.

[2] A. Baldwin and M. Mont. Policy Based Monitoring of a Web-Based Service. *Hewlett Packard Laboratories, Technical Report HPL-98-76*, 1998.

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

[4] 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.

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

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

[7] 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.

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

[9] 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.

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

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

[12] 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.

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

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

[15] H. Lutfiyya, G. Molenkamp, M. Katchabaw, and M. Bauer. Issues in Managing Soft QoS Requirements in Distributed Systems Using a Policy-Based Framework. *Submitted*, 2000.

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

[17] 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.

[18] Lyndon B. Johnson Space Center Software Technology Branch. CLIPS. http://www.ghg.net/clips/CLIPS.html.

[19] Startdust.com. Introduction to QoS Policies. Technical report, Stardust.com, Inc., July 1999.

[20] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. *Proceedings of the Real-Time Systems Symposium*, Washington, DC, December 1996.

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