Administrative Policies to Regulate Quality of Service Management in Distributed Multimedia Applications

Michael Katchabaw, Hanan Lutfiyya and Michael Bauer
Department of Computer Science
The University of Western Ontario
London, Ontario, Canada N6A 5B7

Abstract

With the growth in adoption of next generation distributed multimedia applications, there is a corresponding growing need to adequately manage these applications. At the top of users' lists of requirements is high Quality of Service (QoS), including performance, reliability, availability, and a variety of other operational requirements. Managing quality of service according to these user requirements, however, is no simple task. In addition to satisfying user requirements, QoS management must also respect administrative requirements imposed by higher authorities. Such administrative requirements are necessary to balance user requirements against the capabilities and resources of the distributed computing environment supporting their applications, as well as the needs and broader goals of the enterprise.

Properly capturing and handing such administrative requirements is a challenging, infrequently studied, problem. In this paper, we formalize administrative requirements as administrative policies, and use policy-based management techniques for enforcing them. Doing so adds a great deal of flexibility and power to QoS management. We discuss our general policy-based approach to QoS management, provide several examples of administrative policies, and present highlights from the use of a prototype policy-based QoS management system that uses administrative policies in a variety of experiments with a distributed multimedia application.

Key Words: Administrative policies, quality of service management, distributed multimedia

1 Introduction

Distributed multimedia applications, such as video-on-demand, distance education, tele-medicine, video conferencing, electronic commerce, and several others [6], place more emphasis on the importance of defining an application's Quality of Service than traditional distributed applications such as e-mail. An application's Quality of Service (QoS) refers to its non-functional, run-time requirements. An example QoS requirement for an application receiving a video stream is the following: "The number of video frames displayed per second must be 25, plus or minus 2 frames". In addition to performance requirements, an application's QoS may include availability and reliability requirements.

Technical Report #596, Department of Computer Science, The University of Western Ontario
London, Canada, July 2003

A QoS requirement is *hard* for an application if the application is not functionally correct if the QoS requirement is not satisfied at run-time. Otherwise, the QoS requirement is said to be *soft*. Examples of hard QoS requirements can be found in flight control systems and patient monitoring systems. Most distributed multimedia applications, however, have *soft* QoS requirements. Application QoS requirements for multimedia applications are also often dynamic in that they may vary for different users of the same application, for the same user of the application at different times, or even during a single session of the application.

The allocation and scheduling of computing resources is referred to as *QoS management*. QoS management techniques (for example, [10, 22, 26]), such as resource reservation and admission control can be used to guarantee QoS requirements, since resource reservations are based on worst-case scenarios. This is useful for applications that have hard QoS requirements but often leads to inefficient resource utilisation in environments that primarily have applications with soft or dynamic QoS requirements. Such approaches also tend to encounter difficulties in environments in which all resources are not under the same administrative control.

We have developed a QoS management system that deals with soft and dynamic QoS requirements by providing management services, implemented by a set of management processes and resource managers, that support the following:

1. Detecting that an application's run-time behaviour does not satisfy the application's QoS requirements. This is called a *symptom* and is considered a manifestation of a fault in the system;

2. Fault location, using a set of symptoms, determines a hypothesis identifying possible faults causing the violation of the QoS requirements;

3. Based on the fault and the system state, adaptation actions are chosen that may take either the form of resource allocation adjustments or application behaviour adjustments.

These adaptations depend not only on the cause of the violation, but also depend on the constraints imposed on how to achieve the QoS requirement, especially important in the case of overloaded conditions. These constraints are referred to as *administrative* requirements. For example, administrators may want to prioritize applications so that if the system is overloaded, only those higher priority applications will get the computing resources needed to ensure that their QoS requirements are satisfied. Administrative requirements can also be used to try to prevent QoS violations. For example, administrators may wish to limit access to the computing environment to protect previously admitted applications from QoS requirement violations.

Administrative requirements that are implicitly structured into the code of the QoS management system makes the system rigid, inflexible, and unable to cope when changes are needed to these requirements or when new administrative requirements are to be put in place. To address this problem, the QoS management system that we have developed is *policy-based*. A *policy* is defined [3, 15] as a rule that describes the actions to occur when a specific condition occurs. Policies can be used to semi-formally express both QoS and administrative requirements. Using policies separates the process of formulating management decisions from the management system mechanisms carrying out these decisions, thus allowing for different management decisions to be executed by the same management system mechanisms.

In our earlier work in [14], we presented a simple policy-based QoS management system capable of supporting a single administrative policy for service differentiation. This work, however, was

Technical Report #596, Department of Computer Science, The University of Western Ontario
London, Canada, July 2003

somewhat crude, limited, and ad hoc. Since then, we have demonstrated how the QoS management system [16] services can be mapped to the IETF architectural framework [15] for a more comprehensive solution. This work provided a needed upgrade to the policy distribution and handling mechanisms from our work in [14]. In our current work, discussed in this paper, we extend this previous work to handle a much richer, broader, and more powerful variety of administrative policies to meet the requirements of modern computing environments. Experimental results derived from our improved prototype system using these new administrative policies are also discussed.

This paper is structured as follows. Section 2 presents the details of our policy architecture. Section 3 provides an overview of policy specification in our work. Section 4 describes the specific administrative policies studied in this work. Section 5 outlines our prototype policy based quality of service management system, and presents results from experimentation with administrative policies to date. Section 6 presents a brief discussion of our experiences and observations from this work. Section 7 examines and analyses related work in this area. Finally, Section 8 concludes this paper with a summary, and directions for future work in this area.

2 Policy Architecture

The management architecture used in this work is considered an adaptation of the IETF policy framework [15, 21, 24] and is a generalization of our architecture described in [14]. We first describe the components of this architecture, and then illustrate its use through a sample management scenario.

2.1 Architectural Components

The following is a detailed description of the various components of our architecture. These components are depicted in Figure 1.

Coordinator. The coordinator is the interface between a process and the management system. Application QoS requirements are defined in terms of application-specific attributes. This suggests that the monitoring of the process is needed. An attribute of an application can be monitored with a Sensor and thresholds can be associated with a sensor so that a sensor will only report to the coordinator when the monitored data's value is not within the specified threshold. The coordinator will create an event report which is sent to the Event Manager.

The coordinator is instantiated upon process startup. The coordinator registers with the management system through the Name Server. The coordinator requests the QoS requirements of the application and the actions to be taken if the QoS requirements are not satisfied from the Policy Manager. The Policy Manager returns the QoS requirements as a condition list where each condition is represented as an attribute identifier, comparison, and value that the attribute is to be compared to using the comparison operator. The actions are sent as an action list.

Event Manager. An Event Manager receives and aggregates event reports and allows other management entities to register an interest in an event. The Event Manager can receive event reports from monitoring components in response to a violation of QoS requirements, or it can

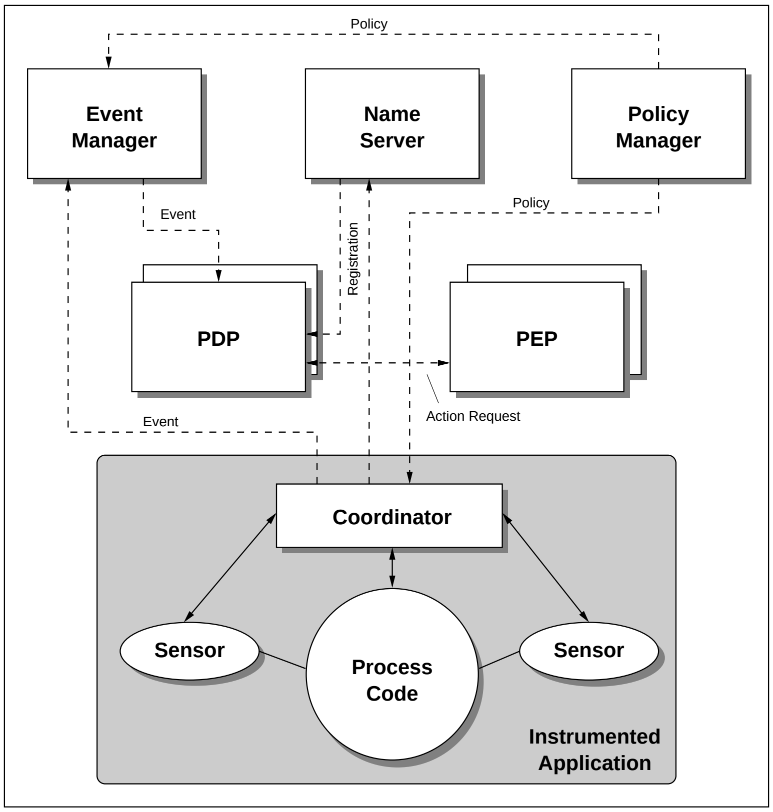

Figure 1: Policy Architecture

receive event reports that represent a regular report of monitored data; for example, resource usage information.

Name Server. The Name Server receives and maintains registration information in a repository. It assigns a unique instance identifier for each registered process, thereby enabling unique identification of each registered component. The Name Server coordinates the interaction between the QoS management components and the Coordinator during the initialization process for an application by notifying the necessary Policy Decision Points of a newly registered application process.

Policy Decision Points (PDPs). PDPs are used to make decisions on actions to be taken based on the receipt of an event which is generated from the monitoring of the environment and relayed through an Event Manager. PDPs are responsible for diagnostics, by determining the cause of the QoS requirement violation and then determining corrective actions. PDPs are also used to authorize actions on behalf of Policy Enforcement Points.

Within an administrative domain, there is one PDP that communicates with a repository where QoS and administrative policies are stored, and information about users is kept. This PDP is referred to as a Policy Manager.

Policy Enforcement Points (PEPs). PEPs apply the actions determined by PDPs. A PEP is a management process that is responsible for monitoring the device that it is on and executing

Technical Report #596, Department of Computer Science, The University of Western Ontario
London, Canada, July 2003

actions. A PEP receives requests from a PDP, verifies that the action to be executed is allowed (by consulting with another PDP and using administrative policies), and performs the requested action if the action is permitted.

2.2 Sample Scenario

The remainder of this section describes possible interactions among the various components listed above. This description is based on the following informal QoS requirement.

Example 2.1 *A video client is to receive video at a frame rate of 25 frames per second, plus or minus 2 frames.*

It is assumed that the video client process can be found on host machine strawberry.syslab.csd.uwo.ca (or just strawberry) and that the video server process can be found on farquarson.syslab.csd.uwo.ca (or just farquarson). For the purpose of this discussion, it is assumed that there is one PEP and PDP for each host machine within a domain. These PDPs authorize actions on behalf of a PEP. There is one PDP for the domain that is located on host machine vanilla.syslab.csd.uwo.ca (or just vanilla) that is used for diagnoses. The Event Manager is also located on vanilla. The administrative requirement used in the discussion is the following: "If process p is in class A and CPU usage on strawberry is less than 80%, then increase the CPU priority of p."

1. When a QoS requirement is violated, the coordinator for the video client generates an event report indicating this to the Event Manager. In this example, if the current frames rate falls below 23 then the fps_low event is generated with information such as the current_fps, target_fps, current_buffer_size and so on. This is packaged into an event report and sent to the Event Manager.

2. The Event Manager sends a notification to each PDP that has registered for the fps_low event. In our example, there is one PDP (the PDP on host machine vanilla) that has registered an interest in this event. The information in the event report constructed in the previous step is sent to the PDP.

3. The PDP uses the information sent from the Event Manager and carries out diagnostics. Based on its diagnosis, the PDP proposes an action to resolve the situation. For example, it may determine that the reason the client process is not receiving the expected average frames per second is the result of the process not getting enough CPU cycles. The PDP recommends that the CPU priority of this process is increased so that it may get enough CPU cycles. This is sent to the PEP on host strawberry.

4. The PEP on host strawberry receives the request from the PDP on vanilla. The PEP communicates with the PDP on the same machine to determine if it can carry out the request. It provides this PDP with resource availability information, including its current CPU load. In this example, the PDP authorizes the PEP to increase the priority of the process if it is in group A and there are enough CPU resources. Otherwise, the PEP on host strawberry will send a request to the video client's coordinator indicating that the client should change the resolution at which it displays video frames. The PDP will continue attempts to find a possible action until it exhausts all possibilities, which essentially implies that nothing further can done.

Technical Report #596, Department of Computer Science, The University of Western Ontario
London, Canada, July 2003

3 Policy Specification

In our current work, policies are formally described using a notation called Ponder [3]. Our use of Ponder as a specification language is intended to make it simpler for administrators to specify policies.

In this formalism, an *obligation* policy specifies the action that a subject must perform on a set of target objects when an event occurs. A *positive authorization* policy specifies permissible actions and a *negative authorization* policy specifies the forbidden actions. Actions specified are carried out by subjects on the specified targets.

The Ponder language provides constructs to define policy types and instances of the policy types. In other words, a policy can be thought of as a class that can be instantiated with specific elements; for example, host machine names or process identifiers.

Targets and subjects are specified using domains. This provides a mechanism for specifying a policy that can be applied to a collection of objects or for specifying the collection of objects that can carry out the actions specified in the policy.

Administration requirements can be specified using a mixture of Ponder obligation and authorization policy constructs. An authorization policy associated with the example administrative requirement stated in Section 2.1 is the following:

Example 3.1

```
type auth+ authCPUIncreaseT (subject s, target t)
{
  action  CPUIncrease(ProcessIdentifier,normalizedvalue)
  when belongs(GroupA, ProcessIdentifier) and CPUResourcesAvailable(normalizedvalue);
}
```

This can be instantiated as follows:

```
inst auth+ allocateResourcePolicy=
  authCPUIncreaseT(/ca/uwo/csd/syslab/strawberry/PEP,
   /ca/uwo/csd/syslab/strawberry/ClientVideoProcess);
```

This policy states that /ca/uwo/csd/syslab/strawberry/PEP is allowed to increase the CPU priority for a video client process if the video client process belongs to GroupA and if there are enough CPU resources. This is calculated by CPUResourcesAvailable. The parameters to this action include the process identifier (ProcessIdentifier) and a normalised value (normalizedvalue) representing the difference between the attribute's current value and the expected value.

4 Policies Studied

This section presents several administrative policies that can be used to influence resource allocation. Each policy is described informally, and also defined formally using Ponder.

4.1 Admission Control

Policies can be used to limit the number of violations of QoS requirements by limiting the number of applications executing in the computing environment. Admission control refers to the process of comparing projected application resource needs with available resources to determine if an application should be allowed to consume computing resources. Upon application registration, policies are used to determine if an application may continue. This can be application specific.

Example 4.1

```
type auth+ authAdmissionControl (subject s, target t)
{
  action  admitProcess(ProcessIdentifier)
  when CPUResourcesAvailabibility() > 5 and FreeMemory > 100;
}

inst auth+ authAdmissionControlstrawberry =
  authAdmissionControl(/ca/uwo/csd/syslab/strawberry/PEP,
                       /ca/uwo/csd/syslab/strawberry/ClientVideoProcess);
```

In this case, the PEP is authorized to admit a process whose identifier is ProcessIdentifier when at least 5% of the CPU and 100 pages of free physical memory are available.

4.2 Differentiated Services

It is not always possible to satisfy quality of service requirements for all applications at the same time. There are a number of possible ways to deal with this scenario, depending on the answers to a variety of questions: Should all applications receive equal service or should some application receive preferential service to meet its requirements? If the latter, how would the application to favour be selected? What kind of favouring would be provided? These decisions can be formulated as service differentiation policies. There are two types of service differentiation policies: uniform service policies and priority-based differentiated service policies.

An example of a priority-based differentiated service policy was given earlier in Example 3.1. Another policy is needed to handle the case when the CPU resources are not available. This is often application specific. An example is the following:

Example 4.2

```
type auth+  authChangeApplicationResolution (subject s, target t)
{
  action ChangeResolution(ProcessIdentifier);
  when not(CPUResourcesAvailable(normalizedvalue));
}
```

This can be instantiated as follows:

```
inst auth+ authChangeApplicationResolutionInstance=
  authChangeApplicationResolution(/ca/uwo/csd/syslab/strawberry/PEP,
                                  /ca/uwo/csd/syslab/strawberry/ClientVideoProcess);
```


Technical Report #596, Department of Computer Science, The University of Western Ontario
London, Canada, July 2003

This policy specifies that /ca/uwo/csd/syslab/strawberry/PEP requests that the video client process's resolution is to be changed if there are not enough CPU resources available. This should allow for the application to maintain the desired frame rate, but at the expense of a lower picture quality. The implementation of of this is done through the use of an actuator, which is an instrumentation component of an application [11].

In addition to the policy stated in 3.1, one may have an additional policy statement that reduces the CPU priority of all processes that are in GroupB. This is only done when CPU availability is low.

Example 4.3

```
type auth+ authCPUDecrease (subject s, target t)
{
  action  CPUDecrease(ProcessIdentifier,normalizedvalue);
  when belongs(GroupB, ProcessIdentifier) and (CPUResourcesAvailability() < 5);
}
```

This can be instantiated as follows:

```
inst auth+ authCPUDecreaseDecreaseInstance =
   authCPUDecrease(/ca/uwo/csd/syslab/strawberry/PEP,
                   groupBProcesses);
```

In this case t is defined as the set of processes that belong to the class with the label groupBProcesses.

Example 4.4

A uniform service policy basically states that any requesting application will receive an increased CPU priority if the resources are available.

```
type auth+ authCPUIncreaseT (subject s, target t)
{
  action  CPUIncrease(ProcessIdentifier,normalizedvalue)
  when CPUResourcesAvailable(normalizedvalue);
}
```

4.3 User Hints

So far, all the events we have been dealing with are QoS requirement violations. User hints are events that indicate that the user's interest or focus of activity has changed, as discussed in detail in [12]. Hints such such as minimizing and restoring windows, the covering and uncovering of windows, and the activation and deactivation of screen savers and locks are all excellent indicators of the relative interest users have in their applications. A user hint can be used to reduce the amount of host computing resources consumed by an application process that is no longer useful or of interest to the user by reducing its CPU priority, which in turn causes it to consume fewer CPU cycles.

Example 4.5

```
type oblig user_hint {subject s, target t)
    on UserHint(ProcessInfo)
    do CPUDecrease(ProcessInfo,normalizedvalue};
}
```

This can be instantiated as

```
inst oblig user_hint_on_strawberry =
   user_hint(/ca/uwo/csd/syslab/strawberry/PEP,
            /ca/uwo/csd/syslab/strawberry/ClientVideoProcess);
```

The PEP is obligated to reduce the CPU priority of a process whose identifier is ProcessIdentifier when a user hint event, as described above, has occurred. This policy is assumed to be used by the PDP on vanilla which has registered an interest in the event User_Hint. The action is sent to the PEP on strawberry. This is validated through an authorization policy used by the PDP on strawberry.

5 Experience

Using the policy based management techniques and sample administrative policies from the previous sections, we have implemented a prototype policy-based QoS management system and have used this prototype to conduct a wide variety of experiments to investigate the effects of administrative policy on quality of service delivery. In this section, we discuss our experience with this to date.

5.1 Prototype

We have developed a prototype policy-based QoS management system for the Solaris 2.6 environment. This prototype was based on work discussed in [14] and extended in [11] and again in [16]. It is capable of managing and regulating access to multiple computing resources, including CPU cycles and memory, with integrated support for networking resources nearing completion. It also supports a wide variety of distributed multimedia applications, as well as more traditional applications.

The policy engine in our prototype, used for formulating and executing management decisions, is built on the Java Expert System Shell (JESS) [5], which is compatible with the C Language Integrated Production System (CLIPS) [7] used in our earlier work. Policies are specified using Ponder, and then mapped into JESS rules for use in our prototype system.

Further implementation details can be found in [11, 14, 16].

5.2 Experimental Results

For experimentation demonstrating the effectiveness of administrative policies, we chose to use a distributed client-server video-on-demand application. In response to a request from a client, the server streams a given MPEG video over a TCP connection back to the client, which decodes the video and presents it to the user. Three machines from our laboratory environment were used in experimentation: farquarson, a Sun Microsystems UltraSparc 1 workstation with a 140 Mhz processor, 64 megabytes of physical memory, and 300 megabytes of virtual memory; strawberry, a Sun Microsystems UltraSparc 5 workstation with a 269 Mhz processor, 128 megabytes of physical memory, and 480 megabytes of virtual memory; and vanilla, a Sun Microsystems UltraSparc 5 workstation with a 269 Mhz processor, 256 megabytes of physical memory, and 760 megabytes of virtual memory. All three machines use Solaris 2.6 as their operating system.

5.2.1 Admission Control

For these experiments, the host farquarson was used to serve multiple video feeds to clients launched on strawberry. Every 10 seconds, a new instance of the video-on-demand application client was launched on this host, and instructed to play the same small 80×60 8-bit video from a file at a mean frame rate of 25 frames per second, plus or minus 2 frames (yielding an acceptable quality range of 23 to 27 frames per second). The video was small enough to be pre-loaded into memory once, and fed to clients without further disk access at run-time.

Figure 2 shows a representative run from the first instance of the video-on-demand client started during experimentation. As noted in the figure, additional instances of the application were started roughly 10 seconds apart (sometimes more, depending on the load on the system induced by other instances of the application). As can be seen from the figure, quality of service is quite good until the fifth instance of the player is added. Variation in quality deliver begins to increase at this point with each subsequent player added, and mean quality delivered drops.

Figure 3 shows a representative run of this experiment, with the admission control policy of Section 4.1 in place to limit access to the system to maintain quality of admitted applications. In this case, our prototype system admitted only four instances of the application at a time, which allowed quality to be delivered in accordance with QoS requirements. (The spikes in quality observed at the launch of each application was found to be the result of the initial entrance of the application to the system before our admission checks were made, providing a brief drop in service during this period.)

5.2.2 Differentiated Services

For experimentation with service differentiation, we chose to use the same application, with a different workload. The application workload assigned was a 10 minute 320×240 8-bit video of footage recorded by our network operations counter cameras stored in a local file. The expectation policy in place dictated that this workload be delivered at a mean frame rate of 24 frames per second, plus or minus 2 frames per second (yielding an acceptable quality range from 22 to 26 frames per second). In this case, the file was too large to cache in memory, so we used two different servers, on farquarson and on vanilla to stream the video to two clients on strawberry.

Figure 4 shows a representative run of this scenario using the uniform services policy discussed in Section 4.2, in which both instances of the video-on-demand application compete for resources on a first-come, first-serve basis. Neither instance of the application is able to meet its QoS requirement, as both instances evenly share available resources.

Figure 5 shows a representative run in this scenario with the service differentiation policies from Example 3.1 and Example 4.3 discussed in Section 4.2 in place to favour one application instance over the other. In this case, service differentiation was enacted as soon as the supply of resources

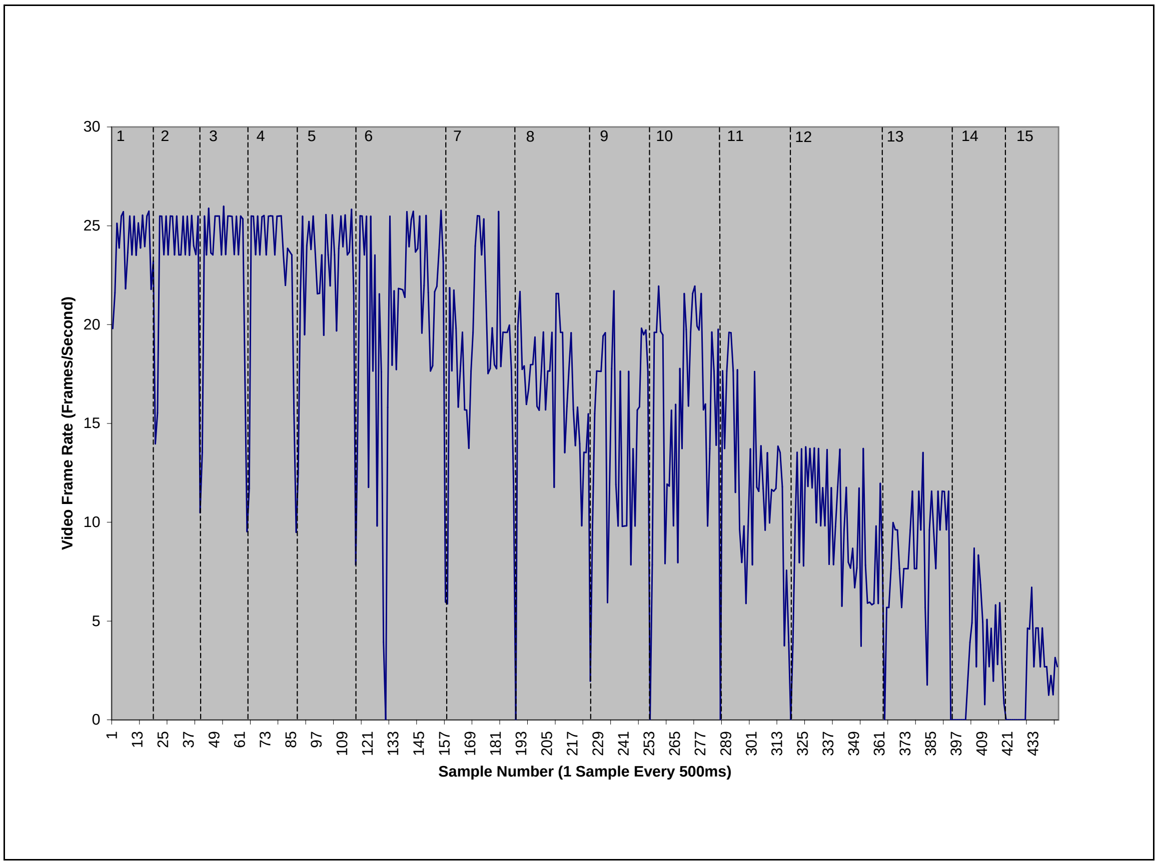

Figure 2: Video-on-Demand without Admission Control

on the host was exhausted, yet demand for the resources was still high. Our prototype system scaled back resources allocated to the second instance of the application and reallocated them to the first. Consequently, the first instance was able to converge and deliver stable quality meeting requirements, while the second was still able to deliver quality at 50% of its target rate.

5.2.3 User Hints

For experimentation with user hints, we used the same scenario used in the previous section for differentiated services experiments. This includes the same application, workload, and host configurations as before.

Figure 6 shows a representative run of this scenario with the user hints policy from Section 4.3 in place. In this case, any user hints indicating a change in user focus or attention would result in a corresponding shift of computing resources. When the application instances started, the video window of one was placed over the other, generating a user hint event, and adjusting resource allocations in its favour. At time A, the ordering of windows was reversed, shifting resources from one instance to the other. At time B, the ordering was reversed once again, resulting in a shift back

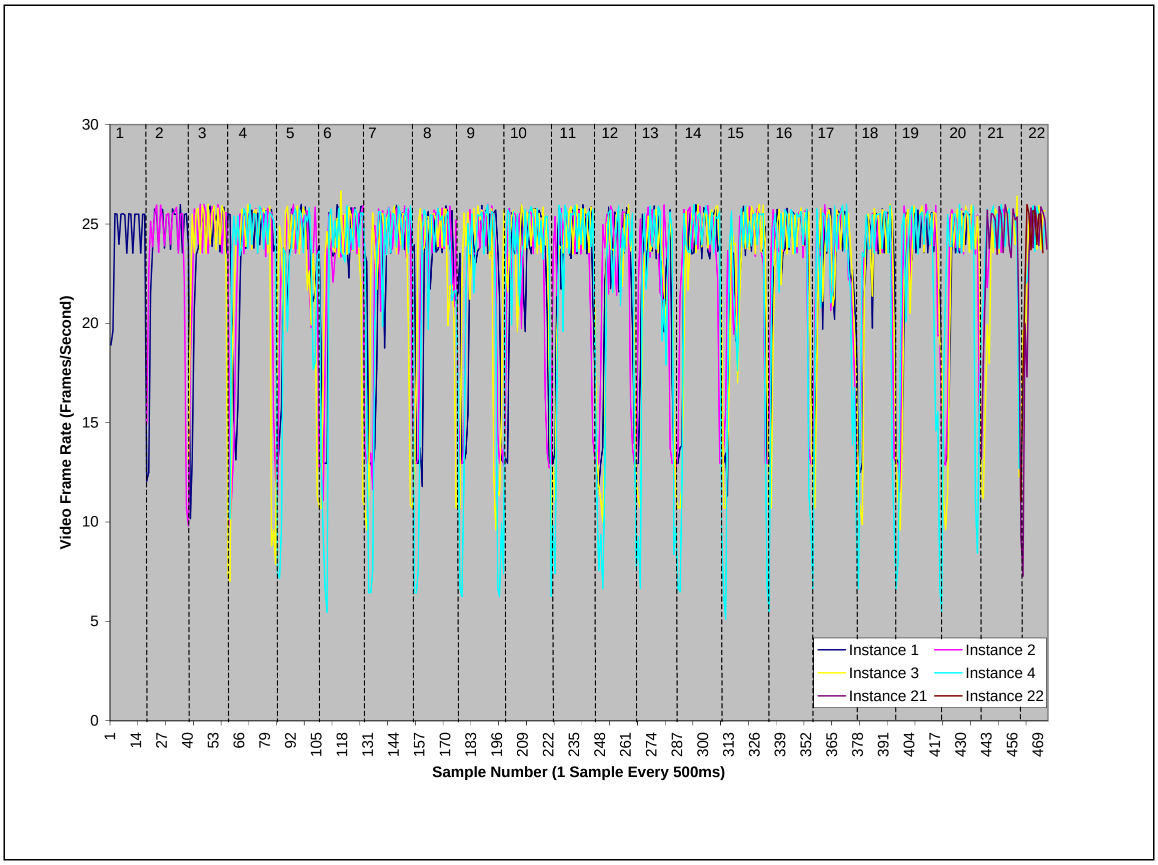

Figure 3: Video-on-Demand with Admission Control

to the original allocation.

5.2.4 Multiple Administrative Policies

The last scenario examined in experimentation involved the use of our admission control, differentiated services, and user hints policies all at the same time. In this experimentation, we used the same application, workload, and host configuration as in the previous two sections of experiments.

Figure 7 shows a representative run from this experimentation. Because sufficient resources were deemed available, both instances of the video-on-demand application were admitted. Windows from both applications were placed in non-overlapping positions at the top of the window stack. Both application instances competed for resources until the supply was exhausted, at which time service differentiation kicked in to favour one instance over the other. When windows were repositioned so that the favoured application was covered, the user hints policy overrode service differentiation, and reallocated resources to the uncovered application instance. When this instance terminated, the remaining instance was reallocated resources and completed its execution.

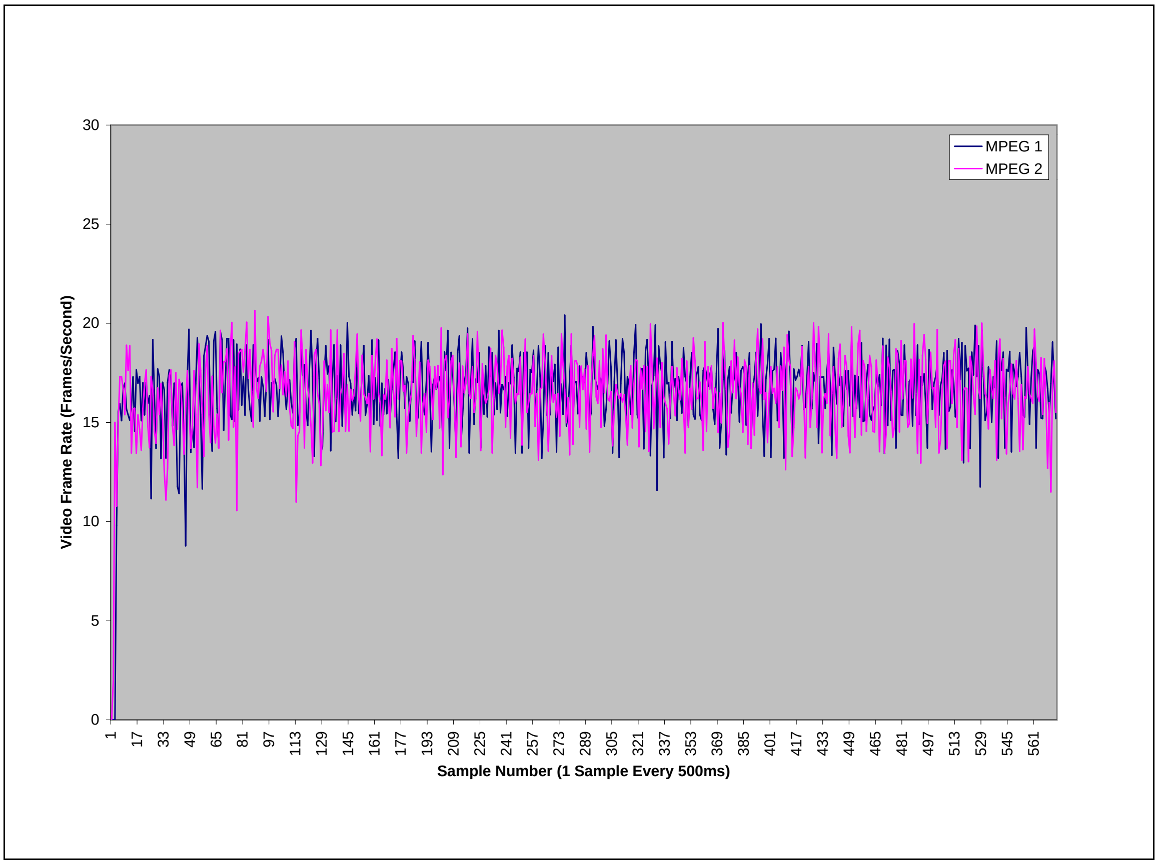

Figure 4: Video-on-Demand with Uniform Services

6 Discussion

This section describes our observations and experiences.

1. In each of the cases discussed in the previous section, we found that the application of administrative policies had a profound difference in the way QoS requirements were met by the distributed multimedia application under study. We could quite effectively control resource allocation mechanisms to direct resources to applications according to violations of QoS requirements or administrative goals for the computing environment.

2. The use of administrative policies adds considerable flexibility and power to management decisions. For example, priority assignment and group classification of application processes, as used in policies defined in Section 4, can be entirely driven by policies. As another example, QoS requirements may be made time-dependent, so that QoS requirements for an application are in effect only at certain times of day, or periodically.

3. We also found that potentially a large number of policies are required to fully define the behaviour of QoS management. For example, the service differentiation policy in Example 4.3

Technical Report #596, Department of Computer Science, The University of Western Ontario
London, Canada, July 2003

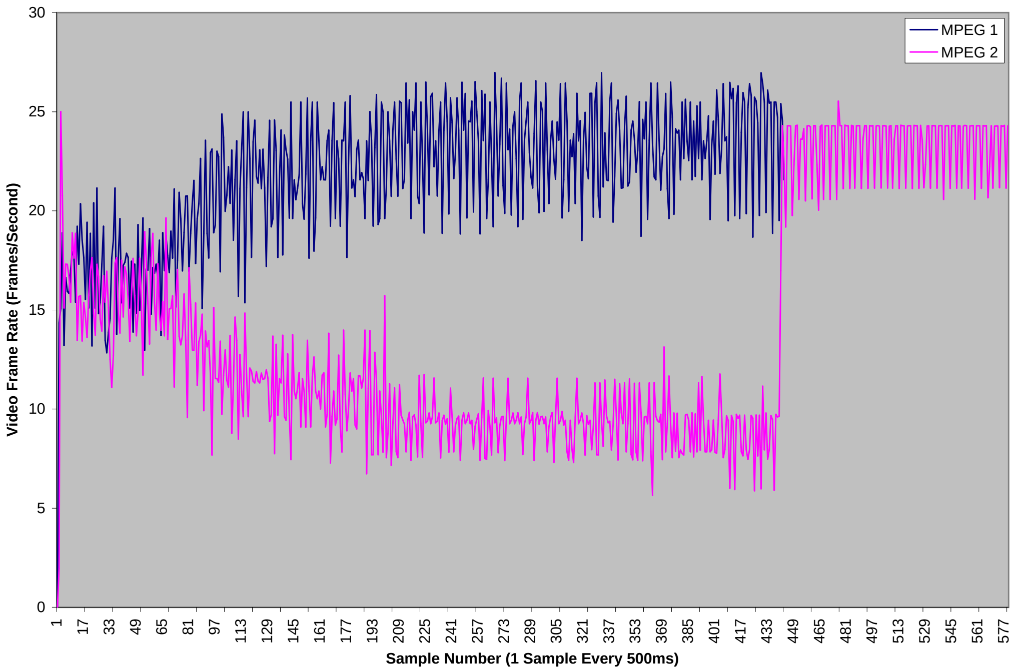

Figure 5: Video-on-Demand with Differentiated Services

will take away resources from all application processes belonging to a certain group when resource supplies dips below a given threshold. This approach is likely overkill; additional policies are required to provide more refined behaviour.

As another example, we currently have no policies in place for specifically reallocating computing resources as they become available on the termination of application processes. At the moment, they are simply added to the pool of resources for later allocations. A more proactive approach may reallocate such freed resources immediately, but this would again require more administrative policies to be put in place.

4. Diagnostics can be centralized or decentralized. We have implemented this both ways in our current prototype. With centralized diagnoses, there is the possibility of a PDP failing or becoming a performance bottleneck. With decentralized diagnoses, organization and distribution of QoS requirements and administrative policies becomes significantly more complex.

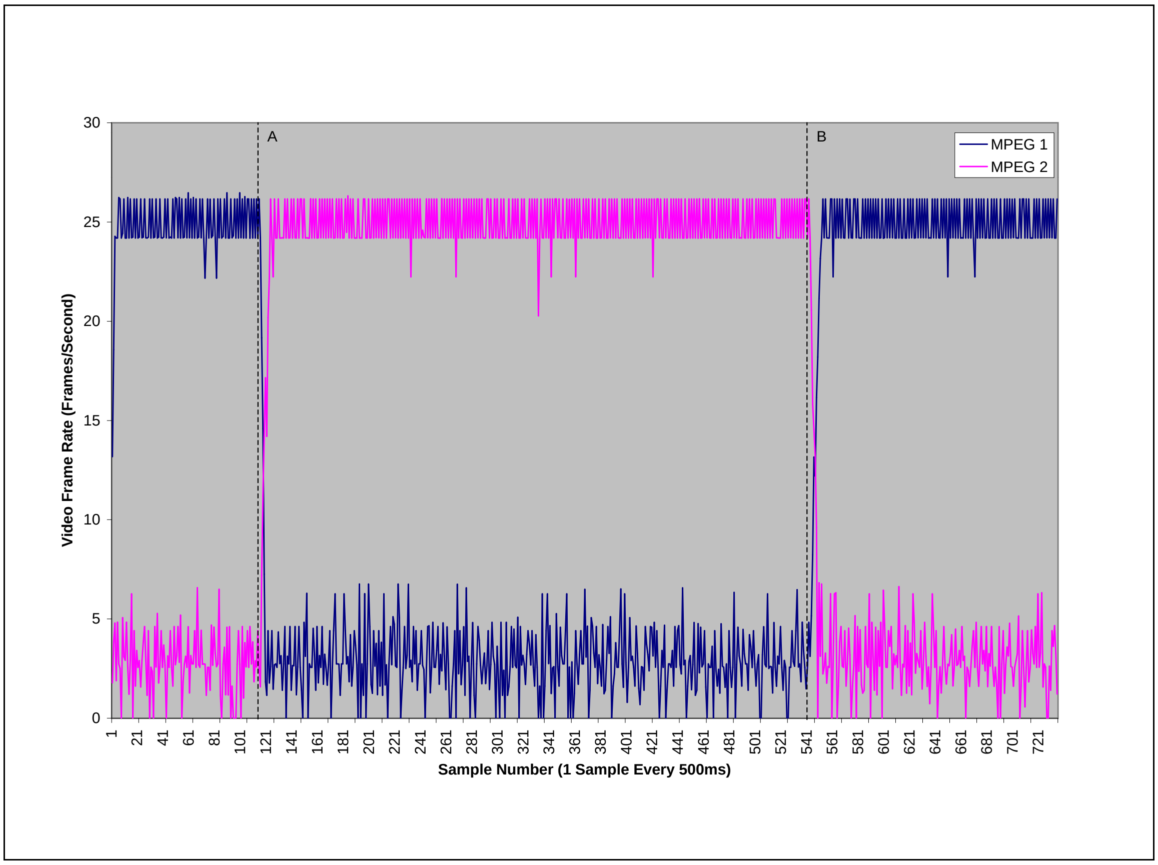

Figure 6: Video-on-Demand with User Hints

7 Related Work

In this section, we describe the related work in the areas of policy specification and policy architectures. We describe how our current work differs and how it relates.

7.1 Policy Specification

There has always been the use of policies in some form to manage networking resources in most enterprises. For example, system administrators often define policies regarding what types of traffic a firewall will accept or reject. However, these policies are rarely formalized, and hence the use of tools to automate the management of network resources based on these policies is relatively primitive. With the number and complexity of network devices increasing, and as more network services are added, there is now considerable interest in formalizing policies and using these formalized policies for managing network resources, especially bandwidth management and access management. This interest also includes the ability to automatically distribute these policies.

The IETF is currently developing a set of standards (current drafts found in [15, 21, 24]) that

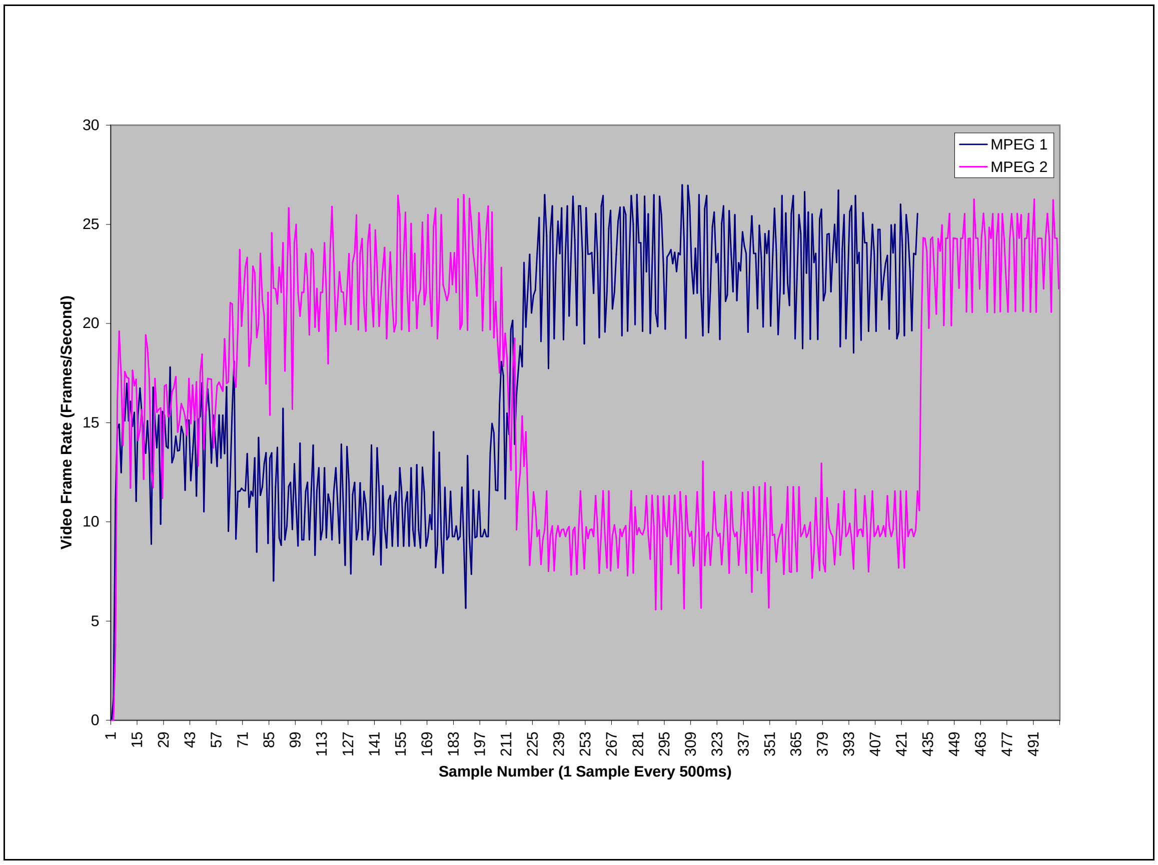

Figure 7: Video-on-Demand with Usage Based Differentiated Services

includes an information model to be used for specifying policies, a standard that extends the previous standard for specifying policies to specifying QoS policies, and a standard for mapping the information model to LDAP schemas. Informally speaking, an IETF policy essentially consists of a set of conditions and a set of actions. The standards tend to focus on low-level details related to policy encoding, storage, and retrieval. We have experience in using these IETF standards, and have examined the use of IETF rules for specifying policies. IETF rules are reasonable for specifying QoS requirements, but we have found that it was much more difficult to specify administrative and diagnostic policies. Generally, we found it easier to specify policies using Ponder and then translate to the IETF format as necessary.

Other language standards by the IETF, as well as other network policy languages (summarized in [23]), include RPSL for describing routing policy constraints, PAX for defining pattern matching criteria in policy-based networking devices and SRL for creating rule sets for real-time traffic flow measurement. These languages primarily focus on policies applied to a network device. None of this work, however, addresses the use of policies applied to applications.

Other policy specification languages [2, 8, 9, 17] focus on features that only enable the specification of security related policies. The Ponder Policy Specification language [3] has a broader scope

Technical Report #596, Department of Computer Science, The University of Western Ontario
London, Canada, July 2003

than most of the other languages, in that it not only was designed with specifying security as the primary objective, but also with management policy in general. Ponder allows the administrator to use declarative statements, and is independent of underlying systems.

7.2 Architectures, Policy Distribution and Enforcement, and Experiences

There has been some recent work in service level management [18, 19] that describes an architecture and policies for the management of a differentiated services network so that users receive good quality of service and fulfill the service level agreements. The policies are primarily applied to network devices and calculate the drop rate for an incoming line. Traffic is mapped into the available traffic classes according to rules configured into packet classifiers. These rules are similar to the notion of policies used in this paper. Experiments were carried out to test the effectiveness of different policies for different loads.

There has also been significant work (for example, [1, 13, 20, 25]) that has looked at the translation of policies to network device configurations which are then sent to the PEPs so that the PEPs can configure the network devices. The focus, in these cases, is typically on network devices. In most of this work, policy distribution is initiated by an administrator and focusses on one device. The emphasis was on the framework and the effectiveness of the translation as opposed to the effectiveness of different policies.

A deployment model was developed in [4] for Ponder. Each policy type is compiled into a policy class by the Ponder compiler. The instantiation of a policy class is a policy object. Each policy object has methods that allow a policy object to be loaded to an enforcement agent and unloaded from an agent. Additional methods include the enabling and disabling of objects. Agents register with the event service to receive relevant events (as specified by the policies) generated from the managed objects of the system. There is no discussion on the configuration of devices or applications. The roles of PDPs and PEPs are similar to that of an enforcement agent. We decided, however, that the use of JESS would make certain tasks easier and more efficient. For example, Ponder assumes that all policies are independent, in the sense that the order in which they are presented in does not matter. We assumed, however, that order of policies mattered when mapping to JESS rules, allowing our system to be more selective in its examination and evaluation policies. In the Ponder deployment model, all policy objects that register for a specific event will receive notification that the event has occurred and conditions are checked, which is not always necessary. Ponder and the deployment model are used to implement the IETF DiffServ notion of Per Domain Behaviour (PDB). This is used to describe the behaviour to be experienced by a particular set of packets as they cross a DiffServ domain. Policies are used to assign users to different PDBs based on their class.

Generally, we found that very little related work focusses on applying policies at the application level. This, however, is an oversight we are addressing through our work in this area.

8 Concluding Remarks

The work we describe here is part of ongoing work on addressing issues in the quality of service management of distributed multimedia applications. This paper focusses on the use of administrative policies to provide flexible and dynamic control over the quality of service management process to balance the individual needs of users against the broader goals of administration. We developed a general policy architecture for facilitating this, and presented a variety of administrative policies that address several quality of service issues. Using a prototype system based on this approach, we have been able to carry out a number of experiments that show the positive effects administrative policies can have on the delivery of quality of service, and demonstrate the effectiveness of our approach.

We have identified several avenues for continuing and future work in this area. We need to investigate other administrative policies that are suitable for quality of service management. We need to examine and refine the administrative policies we have already defined to further improve their effectiveness and applicability. To ease adoption and simplify tasks for administrators, we need to investigate more graphical and user friendly approaches to policy specification, which can then be translated to Ponder or JESS directly. With an increasing number of administrative policies to activate at the same time, we need to also increase efforts looking into techniques for detecting and resolving conflicts between policies automatically. We also need to complete on-going work integrating network resource management into our management system, as well as facilities for I/O and disk resource management. We need to continue porting efforts to other platforms, including the Microsoft Windows family, other variants of the Unix environment, and Java.

Through this work, we have gained valuable insight into the quality of service management problem. With this experience, we can further pursue solutions that completely address the variety of quality of service issues that occur in modern computing environments.

References

[1] M. Brunner and J. Quittek. MPLS Management using Policies. *Proceedings of the 7th IEEE/IFIP Symposium on Integrated Network Management (IM'01)*, Seattle USA, May 2001.

[2] F. Chen and R. Sandhu. Constraints for Role-Based Access Control. *Proceedings of 1st ACM/NIST Role Based Access Control Workshop*, Gaithersburg, Maryland, USA, ACM Press, 1995.

[3] 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.

[4] N Dulay, E. Lupu, M. Sloman, and N. Damianou. A Policy Deployment Model for the Ponder Language. *Proceedings of the 7th IEEE/IFIP Symposium on Integrated Network Management (IM'01)*, Seattle USA, May 2001.

[5] E. J. Friedman-Hill. *Jess, The Rule Engine for the Java Platform.* Sandia National Laboratories Report (SAND98-8206 (revised)), 2003.

[6] N. D. Georganas. Multimedia Applications Development: Experiences. *Journal of Multimedia Tools and Applications, Volume 4, Number 3*, May 1997.

[7] J. Giarratano and G. Riley. *Expert Systems: Principles and Programming.* PWS Publishing Company, 1998.

[8] J. Hoagland, R. Pandey, and K. Levitt. Security Policy Specification Using a Graphical Approach. Technical Report Technical Report CSE-98-3, UC Davis Computer Science Department, UC Davis Computer Science Department, July 22, 1998 2002.

[9] S. Jajodia, P. Samarati, and V. Subrahmanian. A Logical Language for Expressing Authorisations. *Proceedings of the IEEE Symposium on Security and Privacy*, 1997.

[10] M. Jones, D. Roşu, and M. Roşu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. *Proceedings of the Sixteenth ACM Symposium on Operation Systems Principles*, St. Malo, France, October 1997.

[11] M. J. Katchabaw. *Quality of Service Resource Management.* PhD thesis, The University of Western Ontario, June 2002.

[12] M. J. Katchabaw, H. L. Lutfiyya, and M. A. Bauer. Using User Hints to Guide Resource Management for Quality of Service. *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, June 1999.

[13] P. Martinez, M. Brunner, J. Quittek, F. Strauss, J. Schoenwaelder, S. Mertens, and T. Klie. Using the Script MIB for Policy-Based Configuration Management. *Proceedings of the IFIP/IEEE Symposium on Network Operations and Management Symposium (NOMS'02), Florence, Italy, 2002*, pages 461–468, January 2001.

[14] G. P. Molenkamp, M. J. Katchabaw, H. L. Lutfiyya, and M. A. Bauer. Distributed Resource Management to Support Distributed Application-Specific Quality of Service. *Proceedings of the Fourth IFIP/IEEE International Conference on the Management of Multimedia Networks and Services*, Chicago, Illinois, October 2001.

[15] B. Moore, J. Strassmer, and E. Elleson. Policy Core Information Model – Version 1 Specification. Technical report, IETF, May 2000.

[16] N. Muruganantha and H. Lutfiyya. Issues in Policy Specification, Distribution and Architecture for Quality of Service Management. *Integrated Network Management, Volume VIII*, March 2003.

[17] R. Ortalo. A Flexible Method for Information System Security Policy Specificatio". *Proceedings of 5th European Symposium on Research in Computer Security (ESORICS 98)*, Louvain-laNeuve, Belgium, Springer-Verlag, 1998.

[18] P. Pereira, D. Dadok, and P. Pinto. Service Level Management of Differentiated Services Networks with Active Policies. *3rd Conferencia de Telecomunicacoes.*, Rio de Janeiro, Brazil, December 1999.

[19] P. Pereira and P. Pinto. Algorithms and Contracts for Network and Systems Management. *Proceedings of the 1st IEEE Latin American Network Operations and Management Symposium (LANOMS99)*, Rio de Janeiro, Brazil, December 1999.

[20] Alberto Gonzalez Prieto and Marcus Brunner. SLS to DiffServ Configuration Mappings. *Proceedings of the 12th International Workshop on Distributed Systems: Operations and Management DSOM'2001*, Nancy France, October 2001.

[21] Y. Snir, Y. Ramberg, J. Strassner, and R. Cohen. Policy Framework QoS Information Model. Technical report, IETF, April 2000.

[22] 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.

[23] G. Stone, B. Lundy, and G. Xie. Network Policy Languages: A Survey and New Approaches. *IEEE Network*, 15(1):10–21, January 2001.

[24] J. Strassner, E. Ellesson, B. Moore, and Ryan Moats. Policy Framework LDAP Core Schema. Technical report, IETF, November 1999.

[25] P. Trimintzios, I. Andrikopoulos, G. Pavlou, and C. Cavalcanti. An Architectural Framework for Providing QoS in IP Differentiated Services Networks. *Proceedings of the 7th IEEE/IFIP Symposium on Integrated Network Management (IM'01)*, Seattle USA, May 2001.

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