Administrative Policies to Regulate Quality of Service Management in Distributed Multimedia Applications[1]

Michael Katchabaw, Hanan Lutfiyya and Michael Bauer

Department of Computer Science
The University of Western Ontario
London, Ontario, Canada N6A 5B7

Abstract. Properly capturing and handing administrative requirements for Quality of Service (QoS) management 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, QoS management

1 Introduction

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. 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 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

[1] An extended version of this paper can be found as Technical Report #596, Department of Computer Science, The University of Western Ontario.

Proceedings of the 6th International Conference on Management of Multimedia Networks and Services, Belfast, Nortern Ireland, September 2003

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 that detect when an application's run-time behaviour does not satisfy the application's QoS requirements, determine the possible causes of the violation of requirements, and formulate corrective actions to resolve the situation. These actions depend not only on the cause of the violation, but also depend on the constraints imposed on how to achieve the QoS requirement. These constraints are referred to as *administrative* requirements. For example, administrators may want to prioritize applications so that if the system is overloaded, only high priority applications get the computing resources needed to ensure their QoS requirements are satisfied. Administrative requirements can also attempt 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 [1, 7] 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 mechanisms.

In our earlier work in [6], we presented a simple policy-based QoS management system capable of supporting a single administrative policy for service differentiation. This work, however, was somewhat crude, limited, and ad hoc. Since then, we have demonstrated how the QoS management system [8] services can be mapped to the IETF architectural framework [7] for a more comprehensive solution. This work provided a needed upgrade to the policy distribution and handling mechanisms from our work in [6]. 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.

2 Policy Architecture

We first describe the components of our policy architecture, depicted in Figure 1, and their interactions using the following informal QoS requirement.

Example 1 *A video client is to receive video at a frame rate of 25 frames per second, plus or minus 2 frames.*

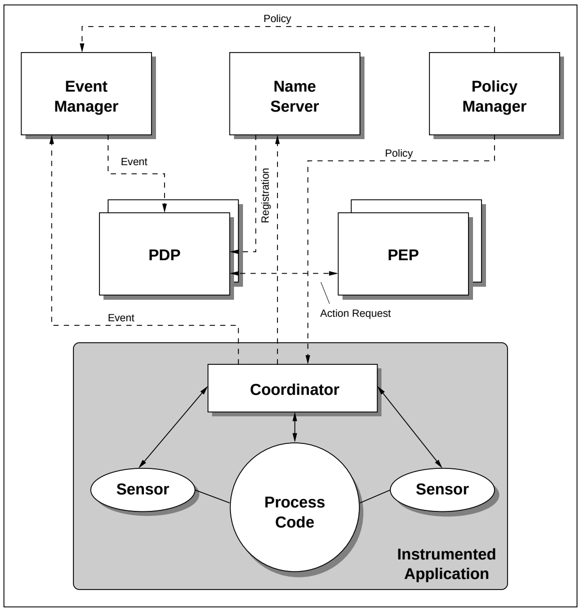

Fig. 1. Policy Architecture

Coordinator. The coordinator is the interface between a process and the management system. QoS requirements are defined in terms of application specific attributes. 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. For Example 1, the attribute to be monitored by a sensor is the video frame rate. If the frame rate is below 23 or above 27, then a report of this is sent to the Coordinator, which in turn forwards an event report to the Event Manager.

Event Manager. An Event Manager receives and aggregates event reports (from violations of QoS requirements or other monitored data) and allows other management entities to register an interest in an event. For Example 1, the Event Manager receives the report from the Coordinator and passes this information on to a Policy Decision Point that has registered interest in this kind of event to resolve the situation.

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 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 monitoring 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. For Example 1, the PDP that received the report from the Event Manager carries out diagnostics to determine how to resolve the situation and notifies the appropriate Policy Enforcement Point of the actions necessary. If the video client in this example has a frame rate below 23, the PDP may determine that the action to be taken is to increase the CPU resources of the client. PDPs are also used to authorize actions on behalf of Policy Enforcement Points. Furthermore, 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 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 permitted. For Example 1, when the PEP receives the instruction to increase CPU resources for the video client from the PDP, it will verify this action with an authorizing PDP, and then add CPU resources if the action is permitted.

3 Policy Specification

In our current work, policies are formally described using a notation called Ponder [1]. Our use of Ponder as a specification language is intended to make it simpler for administrators to specify policies. 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.

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. Targets and subjects are specified using domains, providing a mechanism for applying a policy to a collection of objects or for specifying the collection of objects that can carry out given actions. Administration requirements can be specified using a mixture of Ponder obligation and authorization policy constructs.

Example 2 This policy, a formalization of the requirement in Example 1, states that the PEP is allowed to increase the CPU priority for a video client process if the 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 (ProcessId) and a normalised value (normvalue) representing the difference between the attribute's current value and the expected value.

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

4 Policies Studied

This section presents several different administrative policies. 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 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 3 In this policy, the PEP is authorized to admit a process ProcessId when at least 5% of the CPU and 100 pages of free physical memory are available.

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

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 receive preferential service to meet 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 such policies: uniform service policies and priority-based differentiated service policies.

An example of a priority-based differentiated service policy was given earlier in Example 2. Another policy is needed to handle the case when the CPU resources are not available. An example is given below.

Example 4 This policy specifies that the 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 [5].

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

Example 5 In addition to the policy stated in Example 2, 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.

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

Example 6 This 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(ProcessId,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 [5]. 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 7 In this policy, the PEP is obligated to reduce the CPU priority of a process whose identifier is ProcessId when a user hint event, as described above, has occurred. This policy is assumed to be used by the PDP which has registered an interest in the event UserHint. The action is sent to the appropriate PEP, and is validated through an authorization policy used by the appropriate PDP.

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

5 Experience

Using the policy based management techniques and sample policies from the previous sections, we have implemented a prototype policy-based QoS management system. In this section, we discuss our experience with this system to date.

5.1 Prototype

We have developed a prototype policy-based QoS management system for Solaris 2.6. This prototype was based on work discussed in [6] and extended in [5] and again in [8]. 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) [3], which is compatible with the C Language Integrated Production System (CLIPS) [4] used in our earlier work. Policies are specified using Ponder, and then mapped into JESS rules for use in our prototype system.

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 Sun Microsystems UltraSparc workstations with Solaris 2.6 were used in experimentation: farquarson, strawberry, and vanilla.

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 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 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.

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.

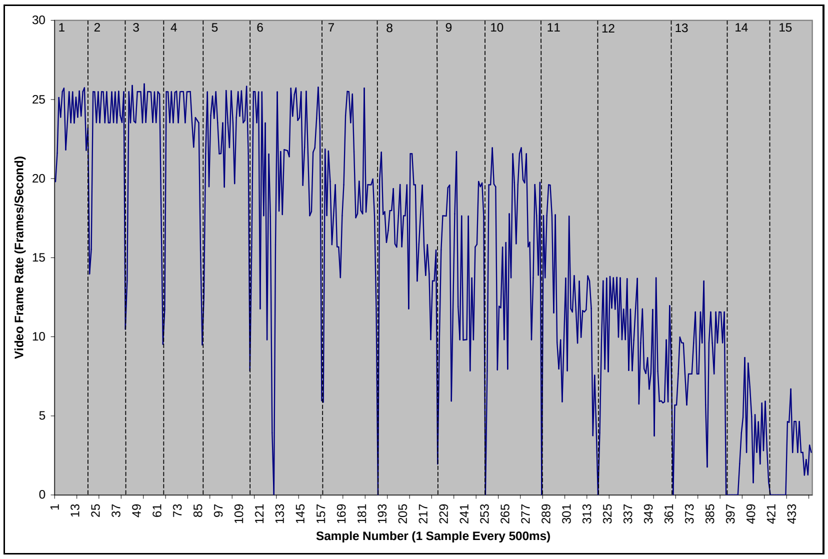

Fig. 2. Video-on-Demand without Admission Control

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 2 and Example 5 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 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.

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

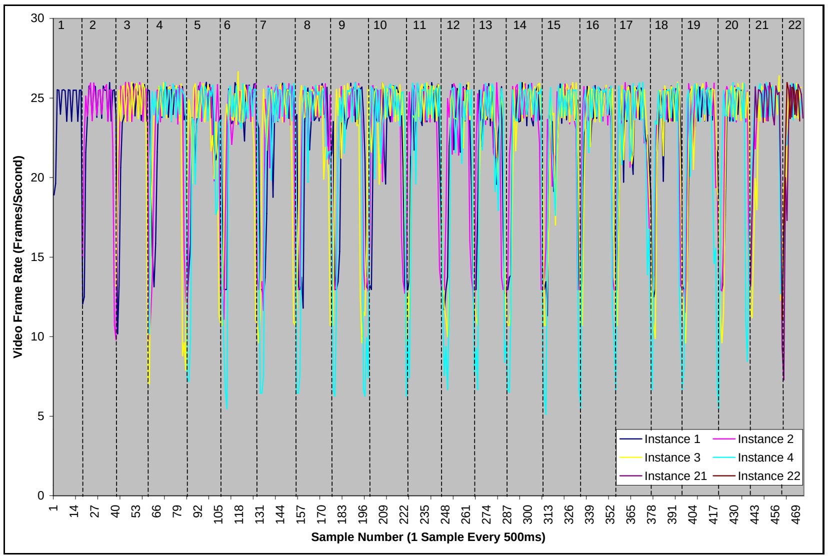

Fig. 3. Video-on-Demand with Admission Control

from one instance to the other. At time B, the ordering was reversed once again, resulting in a shift back to the original allocation.

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.

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.

6 Related Work

The IETF is currently developing a set of standards (current drafts found in [7, 11, 13]) that 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

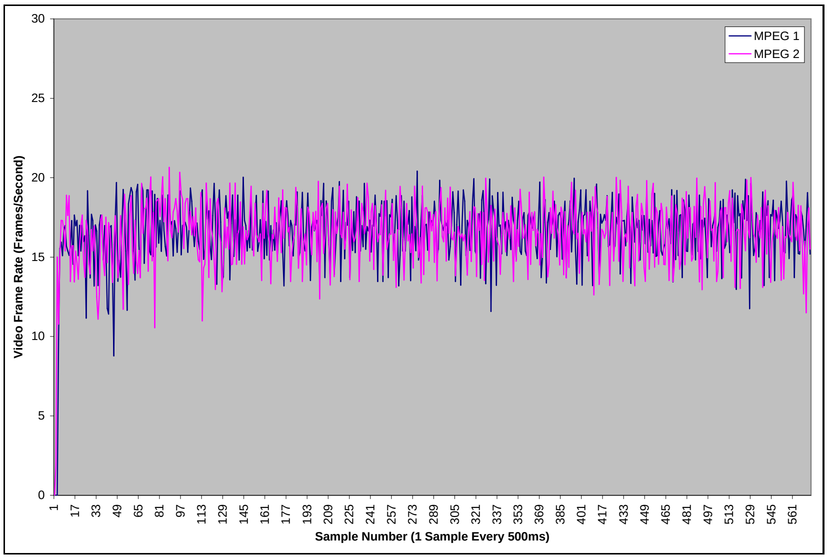

Fig. 4. Video-on-Demand with Uniform Services

schemas. IETF policies 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 have been proposed by the IETF and others, as summarized in [12]. These languages primarily focus on policies applied to a network device; none of this work addresses the use of policies applied to applications. Other policy specification languages (for example, [9] and others) focus on security related policies. The Ponder Policy Specification language [1] has a broader scope than most of the other languages, in that it not only was designed with specifying security, but also with management policy in general.

There has been some recent work in service level management [10] 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.

There has also been significant work (for example, [14] and others) that has looked at the translation of policies to network device configurations. 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 [2] for Ponder. Each policy type is compiled into a policy class by the Ponder compiler, and instantiated as policy objects. Each policy object has methods that allow a policy object to be loaded to

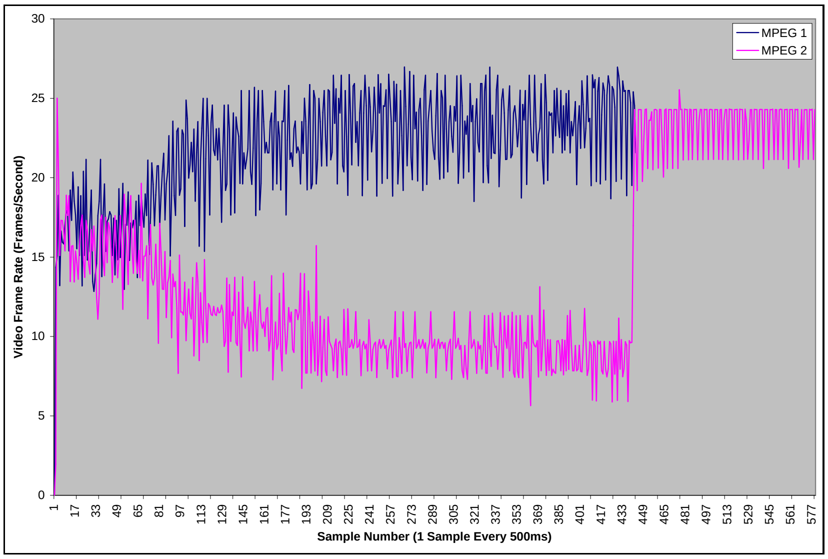

Fig. 5. Video-on-Demand with Differentiated Services

an enforcement agent and unloaded from an agent. 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. Furthermore, this approach has its drawbacks; we found that the use of JESS would make certain tasks easier and more efficient.

Generally, we found that little related work focusses on policies at the application level. We are addressing this oversight through our work in this area.

7 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 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

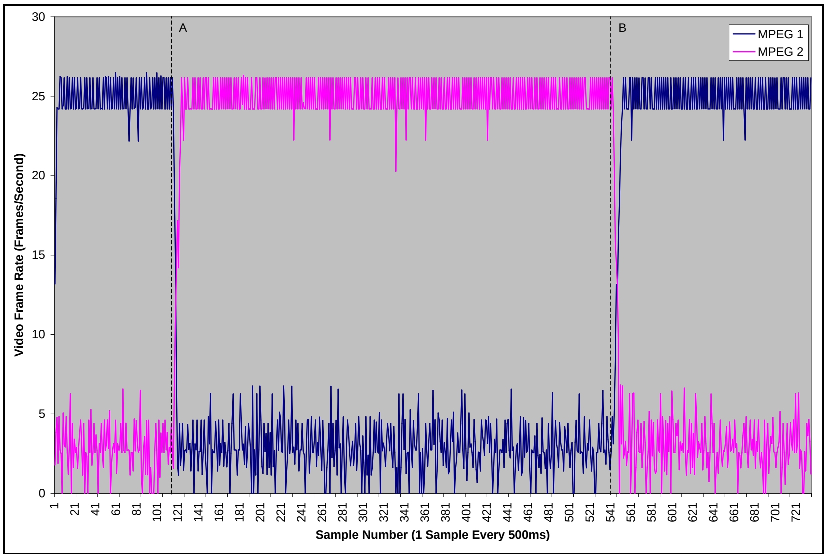

Fig. 6. Video-on-Demand with User Hints

already defined to further improve their effectiveness and applicability. To ease adoption and simplify tasks for administrators, we need to investigate more 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 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.

References

1. N. Damianou, N. Dalay, E. Lupu, and M. Sloman. Ponder: A Language for Specifying Security and Management Policies for Distributed Systems: The Language Specification. *Imperial College Research Report DOC 2000/01, Imperial College of Science, Technology and Medicine*, April 2000.
2. 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.
3. E. J. Friedman-Hill. *Jess, The Rule Engine for the Java Platform.* Sandia National Laboratories Report (SAND98-8206 (revised)), 2003.
4. J. Giarratano and G. Riley. *Expert Systems: Principles and Programming.* PWS Publishing Company, 1998.

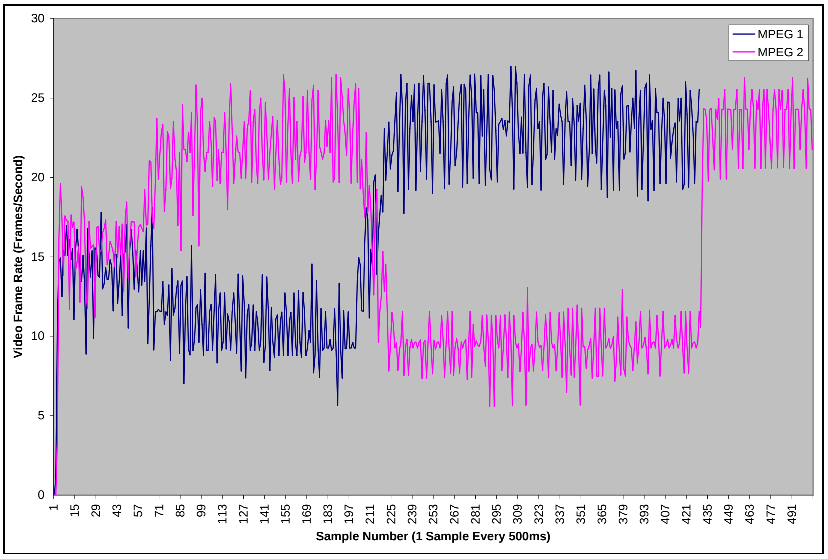

Fig. 7. Video-on-Demand with Usage Based Differentiated Services

5. M. J. Katchabaw. *Quality of Service Resource Management.* PhD thesis, The University of Western Ontario, June 2002.
6. 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.
7. B. Moore, J. Strassmer, and E. Elleson. Policy Core Information Model – Version 1 Specification. Technical report, IETF, May 2000.
8. N. Muruganantha and H. Lutfiyya. Issues in Policy Specification, Distribution and Architecture for Quality of Service Management. *Integrated Network Management, Volume VIII*, March 2003.
9. 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.
10. 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.
11. Y. Snir, Y. Ramberg, J. Strassner, and R. Cohen. Policy Framework QoS Information Model. Technical report, IETF, April 2000.
12. G. Stone, B. Lundy, and G. Xie. Network Policy Languages: A Survey and New Approaches. *IEEE Network*, 15(1):10–21, January 2001.
13. J. Strassner, E. Ellesson, B. Moore, and Ryan Moats. Policy Framework LDAP Core Schema. Technical report, IETF, November 1999.
14. P. Trimintzios, I. Andrikopoulos, G. Pavlou, and C. Cavalcanti. An Architectural Framework for Providing QoS in IP Differentiated Services Networks. *Proceedings of the 7th Symposium on Integrated Network Management*, Seattle USA, May 2001.