Intelligent QoS Support for an Adaptive Video Service

Kyungkoo Jun, Ladislau Bölöni, David Yau, and Dan C. Marinescu
Computer Sciences Department, Purdue University
West Lafayette, IN, 47907, USA
Email: [junkk, boloni, yau, dcm]@cs.purdue.edu

October 18, 1999

Abstract

In this paper we present an adaptive video service architecture. Software agents provide feedback regarding the desired and the attained quality of service at the client side. Server agents respond by reconfiguring the server and reserving communication bandwidth and/or CPU cycles according to a set of rules. The adaptation mechanism is controlled by an inference engine running as one of the strategies of the server agent. The agents are assembled dynamically from reusable components using the Bond Agent Framework. QoS reservation is supported by two native resource managers in Solaris 2.5.1, for network bandwidth and CPU cycles respectively.

Contents

1 Introduction

Data streaming requires that enough communication bandwidth and CPU cycles be dedicated to an audio or video application and there is a consensus that QoS for multimedia applications cannot be guaranteed without reservation of resources. QoS guarantees require: (a) the characterization of an application in terms of overall resource consumption for various levels of service, (b) a mechanism to negotiate the level of service between the provider and the consumer, (c) a mechanism to reserve resources, and (d) a mechanism to enforce reservations.

In this paper we propose an architecture supporting server reconfiguration and resource reservations for a video application. Software agents provide feedback regarding the desired and the attained quality of service at the client side. Server agents respond by reconfiguring the server and reserving communication bandwidth and/or CPU cycles according to a set of rules. The adaptation mechanism is controlled by an inference engine running as one of the strategies of the server agent. The agents are assembled dynamically from reusable components using the Bond Agent Framework. QoS reservation is supported by a native bandwidth scheduler and a CPU scheduler in Solaris 2.5.1.

Adaptation using active networks [2], and other methods, [3], have been proposed in the past. Yet the agent approach discussed in this paper is more convenient, it does not require that the software be reinstalled every time a change of resource allocation policy takes place and it is capable of handling complex negotiations between the parties involved.

The contributions of this paper are an agent-based architecture for QoS adaptation and a set of rules for reconfiguring an MPEG server based upon the information about resources and needs of a video service application. We also report measurements performed on our test bed system.

This paper is organized as follows. Section 2 introduces our resource managers for network bandwidth and CPU cycles. Section 3 provides an overview of the Bond Agent Framework. Section 3 introduces the multimedia agents, the data streaming modes the inference strategy and the rules for server reconfiguration and for resource reservations. Measurements are reported in Section 5 and conclusions are presented in Section 6.

2 Operating System Support for QoS

Middleware multimedia agents interface with native OS resource managers to secure resources for an assured level of service. This section describes our implementation of two resource managers in Solaris 2.5.1 for network bandwidth and CPU cycles, respectively.

2.1 Bandwidth scheduling

The bandwidth manager in our system is called *Tempo.* Tempo allows application flows to reserve for guaranteed network bandwidth. Its software architecture consists of a stream *driver* for control operations and a stream *module* for flow classification and scheduling [7]. Standard Solaris applications can benefit from Tempo services without any modifications.

Tempo supports very flexible resource sharing according to Internet "flow specifications". An Internet *flow* is defined by the five-tuple:

```
<ip_src, ip_dst, proto, src_port, dst_port>
```

i.e. as source IP address, destination IP address, transport protocol (usually TCP or UDP), protocol source port number, and protocol destination port number. A *flow specification* then gives a set of flows by allowing any number of fields in the five tuple to be *partially* specified: IP addresses can be wildcarded to different degress (e.g. * or 128.*), port numbers can specifiy a range (e.g. "don't care" or 1050–1100), etc. When all fields of the five tuple are completely unspecified, we have a *default* flow specification, i.e. one that matches all network packets.

The default flow specification is at the root of each (per-interface) Tempo sharing hierarchy. This root specification can be partitioned into a number of *child* flow specifications that are all disjoint. Each of these children can then be further partitioned in a recursive manner. To ensure that sharing relationships are well formed, we further require that if C is a child flow specification of its parent P, then the set of flows that match C must be a subset of the flows that match P.

For real-time performance, each flow specification in the sharing hierarchy can be given a *resource specification.* This resource specification is in the form of a *service curve* [8], which defines a monotonically non-decreasing function of time $S(.)$. Intuitively, $S(t)$ (in bits) specifies the minimum amount of service that a flow specification with the reservation should receive by time t (in μs), provided that the set of flows matching the flow specification has "sufficient" aggregate demand. Notice that $S(.)$ for the default flow specification of an interface with bandwidth B (in Mbps) should be linear with slope B. To guarantee service curves, we need to prevent resource over-subscription. This can be achieved by ensuring that for each non-leaf flow specification P, the sum of $S(.)$ for all of P's children does not exceed $S(.)$ of P itself. Figure 1 illustrates an example Tempo sharing hierarchy.

While $S(.)$ is generally defined to be a monotonically non-decreasing function of time, in practice we represent it as a sequence of linear segments. Each segment is of the form $< t_1, m, t_2 >$, meaning

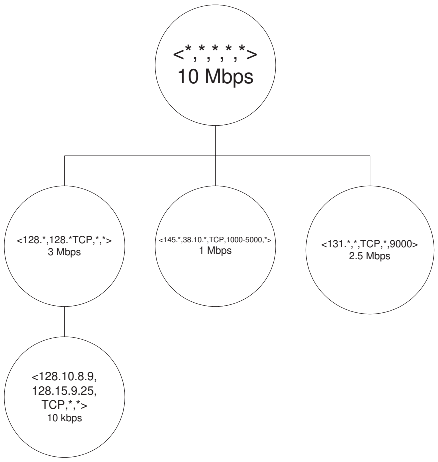


Figure 1: Example Tempo sharing hierarchy by flow specification.

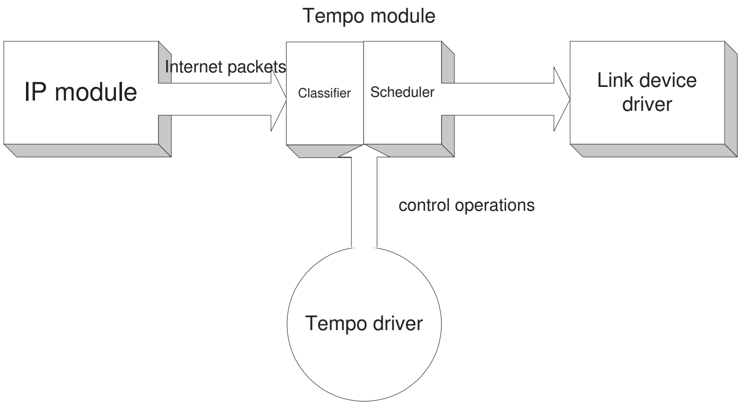


Figure 2: Tempo module/driver software architecture.

that $S(.)$ has slope m during time $[t_1, t_2 >$. It can be seen that service curve generalizes the widely used notion of service *rate* (in that a rate defines a linear service curve). By doing so, it allows delay and rate performance to be decoupled [8] [10]. This benefits an important class of flows, such as interactive audio, which have low long term rate but nevertheless desire low delay guarantees.

Software architecture. Each network interface configured to use Tempo will have a Tempo module "pushed" between the device driver for the interface and a common IP multiplexor module (see Figure 2). Packets sent by standard Internet applications are routed by IP to go out of a certain network interface. If the interface is Tempo-enabled, these packets will be processed by the corresponding Tempo module. Each of them will be classified to a *most specific* flow specification. For the sharing hierarchy in Figure 1, for example, packet `<128.211.1.10,128.180.3.10,TCP,7500,15000>` will be classifed to `<128.*,128.*,TCP,*,*>` and packet `<128.10.8.9,128.15.9.25,TCP,10000,60000>` will be classified to `<128.10.8.9,128.15.9.25,TCP,*,*>`, which is a more specific match than the alternative `<128.*,128.*,TCP,*,*>`. Given the classified flow specifications, Tempo implements the *fair service curve earliest deadline first* scheduling algorithm in [10]. The algorithm guarantees the service curve of each flow specification while minimizing progress unfairness between flows.

Bandwidth scheduling interface. Applications manage resource reservations for a send interface by opening the Tempo device driver for that interface. An open returns a standard file descriptor which serves as a handle for various `ioctl` calls to control the interface. In particular, the following control operations are supported:

1. Create a flow specification with a given resource reservation. The create request will fail if it will result in a sharing hierarchy that is not well formed, or if there is insufficient available bandwidth. Otherwise, the call will return a handle that can be used by subsequent operations on the flow specification.
2. Delete a flow specification given its handle.
3. For the flow specification identified by a given handle, modify its resource specification.
4. List the current sharing hierarchy.

Notice that the sharing hierarchy of a Tempo-enabled interface is global to all applications. As a result, resource reservations used by an application need not be managed by the application itself. Instead, a *third-party* control program can be used to manage reservations on behalf of diverse user applications. This allows resource management functions to be localized in "specialist" modules such as our middleware multimedia agents. These agents can then export resource management services to a wide range of real-time applications such as video servers, Web based media streaming, teleconferencing, etc.

2.2 CPU scheduling

Our CPU manager allows Solaris applications to reserve for guaranteed CPU time. It defines a CPU sharing hierarchy that is very similar to that used by Tempo. Similar to Tempo, for example, it allows CPU capacity to be recursively partitioned into *configured* service classes with specified resource specifications. Threads can then be admitted to any configured service class with sufficient capacity, and we view them as *leaf* service classes (which of course cannot admit other threads). A resource specification for a (configured or leaf) service class is in the form of a service curve $S(.)$. In this case, $S(t)$ (in μs) gives the minimum amount of CPU time that a service class with the resource specification should receive by time t (in μs), provided that the service class has sufficient demand.

Unlike Tempo, however, the sharing hierarchy cannot be defined by way of flow specification relationships. Instead, any service class created to join a sharing hierarchy is given a system wide integer identifier. A class with id p can then be configured explicitly as a child of another class with id q using the call

```
join_node(p, q);
```

Threads are likewise created to be children of specified configured service classes. A created thread can later change its resource specification, including leaving its original service class and joining a new one. This is provided by the SVR4 `priocntl(2)` API call discussed below.

CPU scheduling interface. Solaris comes from the lineage of Unix SVR4. As such, it supports different *scheduling classes*. Our system retains the use of the SYS class for interrupt processing (in Solaris, interrupts have thread context). Our service curve based CPU scheduling is implemented as a new scheduling class called SC. At system startup time, we partition CPU capacity into three default service classes: SC_C0, SC_S1, and SC_S2, that are allocated 70%, 20% and 10% of the CPU respectively. Once the system is booted up, a privileged user can repartition the CPU into other configurations.

We change the Solaris kernel to run the `init` process (process number 1 and an ancestor of all Unix user processes) in SC_C0 with the linear service curve of slope 0.001. In Unix SVR4, children processes forked by a parent process inherit the scheduling class and parameters of the parent by default; therefore, all user processes (and their associated threads) also run in SC_C0 with the linear service curve of slope 0.001 by default.

We provide two principal ways to change the default scheduling parameters of an SC thread. First, we support the `priocntl(1)` command for SC scheduling. Using this command, any standard Solaris application can be started with specified parameters from a Unix shell, as follows:

```
priocntl -e -c SC -r1 <m1> -x <d> -r2 <m2> -n <class> <program>
```

where <program> is the name of the application to run, <class> identifies the service class (e.g. SC_C0) that is to be the parent of the application, and the service curve for <program> is specified to have slope <m1> (in unit of 0.1%) during time $[0, d >$ (<d> is in μs) and slope <m2> during time $[d, \infty >$ (i.e. we restrict the specification to only two piecewise linear service curves).

Second, application threads once started can change their scheduling parameters using the SVR4 `priocntl(2)` system call. For the `PC_SETPARMS` entry point of `priocntl`, which sets the scheduling paramters of a specified set of threads (e.g. all threads belonging to a given process, the thread for the current LWP, all threads belonging to a given user, etc), the SC class specific parameter structure is defined as follows:

```
struct {
     int   m1;     /* in 0.1 % */
     int   m2;     /* in 0.1 % */
     int   d;      /* in microseconds */
     int   class;  /* id of parent service class */
};
```

3 Bond Agent Framework

Bond [4] is a Java-based distributed object system and agent framework, with an emphasis on flexibility and performance. It is composed of (a) a core containing the object model and message oriented middleware, (b) a service layer containing distributed services like directory and persistent storage services, and (c) the agent framework, providing the basic tools for creating autonomous network agents together with a database of commonly used strategies which allow developers to assemble agents with no or minimal amount of programming.

3.1 Bond Core

At the heart of the Bond system there is a Java Bean-compatible component architecture. Bond objects extend Java Beans by allowing users to attach new properties to the object during run-time, and offer a uniform API for accessing regular fields, dynamic properties and JavaBeans style `setField/getField`-defined virtual fields. This allows programmers the same flexibility like languages like Lisp or Scheme, while maintaining the familiar Java programming syntax.

Bond objects are network objects by default: they can be both senders and receivers of messages. No post-processing of the object code as in RMI or CORBA-like stub generation, is needed. Bond uses *message passing* while RMI or CORBA-based component architectures use *remote method invocation.*

The system is largely independent from the message transport mechanism and several communication engines can be used interchangeably. We currently provide TCP-based, UDP-based, Infospheres-based, and, separately, a multicast engine. Other communication engines will be implemented as needed. The API of the communication engine allows Bond objects to use any communication engines without the need to change or recompile the code. On the other hand, the properties of the communication engine are reflected in the properties of the implemented application as a whole. For example the UDP based engine offers higher performance but does not guarantee reliable delivery.

All Bond objects communicate using an agent communication language, KQML [9]. Bond defines the concept of *subprotocols*, highly specialized, closed set of commands. Subprotocols generally contain the messages needed to perform a specific task. Examples of generic Bond subprotocols are *property access* subprotocol, *agent control* subprotocol or *security* subprotocol. An alternative formulation would be that subprotocols introduce a *structure in the semantic space of the messages.*

Subprotocols group the same functionality of messages which in a remote method invocation system would be grouped in an *interface.* But the larger flexibility of the messaging system allows for several new techniques which are difficult to implement in the remote method call case:

- The subprotocols implemented by objects are properties of the object, so two objects can use the property access subprotocol implemented by every Bond object, to find the common set of subprotocols they can use to communicate.

- An object is able to control the path of a message and to delegate the processing of the message to subcomponents called *regular probes*. Regular probes can be attached dynamically to an object as needed.
- Messages can be intercepted before they are delivered to the object, thus providing a convenient way to implement security by means of a fire wall, accounting, logging, monitoring, filtering or preprocessing messages. These operations are performed by subcomponents called *preemptive probes* which are activated before the object in the message delivery chain.
- Subprotocols, like interfaces, are grouping some functionality of the object, which may or may not be used during its lifetime. A subcomponent called *autoprobe* allows the object to instantiate a new probe, to handle an incoming message which can not be understood by the existing subcomponents attached to an object.
- Objects can be addressed by their unique identifier, or by their *alias*. Aliases specify the services provided by the object or its probes. An object can have multiple aliases and multiple objects can be registered under the same alias. The latter enables the architecture to support *load balancing* services.

These techniques can be implemented through different means in languages which treat methods as messages, e.g. Smalltalk. In Java and C++ they can be implemented at compile time, not at runtime, e.g. using the delegation design pattern. Techniques from the recent CORBA specifications e.g. the simultaneous use of DII, POA, trading service and others, also allow to implement a similar functionality, but with a larger overhead, and significantly more complex code.

3.2 Bond Services

Bond provides a number of services commonly used in distributed object systems, like directory, persistent storage, monitoring and security. Event, notification, and messaging services, which provide message passing services in remote method invocation based systems are not needed in Bond, due to the message-oriented architecture of the system.

Some of Bond services perform differently than their counterparts in other middleware systems, like CORBA. For example, Bond never requires explicit registration of a new object with a service. Finding out the properties of a remote object, i.e. the set of subprotocols implemented by the object, is done by direct negotiation amongst the objects. The directory service in Bond combines the functionality of the naming and trading services of other systems and it is implemented in a distributed fashion. Objects are located by a search process which propagates from local directory to local directory. The directories are linked into a virtual network by a transparent *distributed awareness* mechanism, which transfers directory information by piggybacking on existing messages.

Compared with the naming service implementations in systems like CORBA or RMI, which are based on the existence of a name server, this approach has the advantage that there is no single point of failure, and the distributed awareness mechanism reconstitutes the network of directories even after catastrophic failures. However, a distributed search can be slower than lookup on a server, especially for large networks of Bond programs. For these cases, Bond objects can be registered to external directories, either to a CORBA naming service through a gateway object, or to external directory services using LDAP access.

3.3 Bond Agents

The *Bond agent framework* is an application of the facilities provided by the Bond core layer to implement collaborative network agents. Agents are assembled dynamically from components in a structure described by a multi-plane state machine, [5]. This structure is described by a specialized language called `blueprint`. The active components (*strategies*) are loaded locally or remotely, or can be specified in interpretive programming languages embedded in the blueprint script. The state information and knowledge base of the agents are collected in a single object called *model of the world* which allows for easy checkpointing and migration of agents. The multiplane state machine describing the behavior of agents can be modified dynamically, thus allowing for *agent surgery*.

The *behavior* of the agent is described by the *actions* the agent is performing. The actions are performed by the strategies either as reactions to external events, or autonomously in order to pursue the *agenda* of the agent. The current state of the multiplane state machine (described by a *state vector*) is specifying the strategies active at a certain moment. The multiple planes are a way of expressing parallelism in Bond agents. A good technique is to use them to express the various facets of the agents behavior: sensing, reasoning, communication/negotiation, acting upon the environment and so on. The *transitions* in the agent are modifying the behavior of the agent by changing the current set of active strategies. The transitions can be triggered by internal events or from external messages - these external messages form the *control subprotocol* of the agent.

Strategies, having limited interface requirements are a good way to provide code reuse. The Bond agent framework provides a strategy database, for the most commonly used tasks, like starting and controlling external agents or legacy applications. A number of base strategies for common tasks like dialog boxes or message handlers are also provided, which can be sub-classed by developers to implement specific functionality. External algorithms, especially if written in Java are usually easily portable to the strategy interface.

4 Adaptive MPEG Video Server Architecture

In this section we describe the basic architecture of a system using agents to negotiate the level of service between an MPEG video server and a community of video clients.

The Moving Picture Experts Group, MPEG, is a set of standards used for coding digital audio-visual information in a compressed format [1]. MPEG-1 was developed for storing video data and its associated audio data on digital storage media and intended for data rates on the order of 1.5 Mbit/sec. MPEG-1 specifies an algorithm for compressing video pictures and audio and then provides the facility to synchronize multiple audio and multiple video bit streams in an MPEG-1 system.

The MPEG-1 video stream distinguishes between I-frames, P-frames, and B-frames, which differ in the coding scheme providing three levels of compression by exploiting similarities within the picture or with neighboring pictures. I-frames are the most complete, P-frames contain difference from former I or P-frames, and B-frames are encoded with differences from both preceding and following I or P-frames, thus contain the least amount of data.

4.1 Multimedia Agents and Streaming Modes

We present an adaptive MPEG system where multiple video clients connect to a video-server and negotiate the level of service using Bond agents as shown in Figure 3. The *MPEG video client* is a process responsible to display a video stream. This process is started by means of a *client agent* that monitors the reception of the video stream. Whenever an MPEG client agent requests a video stream, the *MPEG video server* spawns an *MPEG server* processes to deliver the video stream and an *MPEG server agent* to control the streaming modes. There are control channel between the server agent and the client agent for feedback and command, and video streaming channal between the MPEG server and the MPEG video client.

The MPEG server agent can adapt with different stream modes. Initially the MPEG server agent is configured to deliver a compressed video stream. However as the resource states changes, the agent can select other streaming modes. The affecting resources are network bandwidth and the CPU loads on the server and client sides. The server agent currently supports four streaming modes:

- *Compressed Video Stream.* The MPEG server reads the video stream from the Video Data Base or from a local file and transmits it to a client. The MPEG client decodes the video stream and displays the frames. Decoding the video stream is a CPU intensive operation.

- *Drop B,P Frame.* The MPEG server partially decodes the video stream to identify the frame types and drops certain type of frames. The sever selects the frames of which type affects the video quality less than other types, for example, B-type and P-type. This mode is suitable for low bandwidth.

- *Server Decode.* The MPEG server transmits the decoded frames to a client. It allows the client to use less CPU cycle by avoiding decoding process, whereas the transmission bandwidth increases because of the frames size, for example, in some cases the frame size increases ten

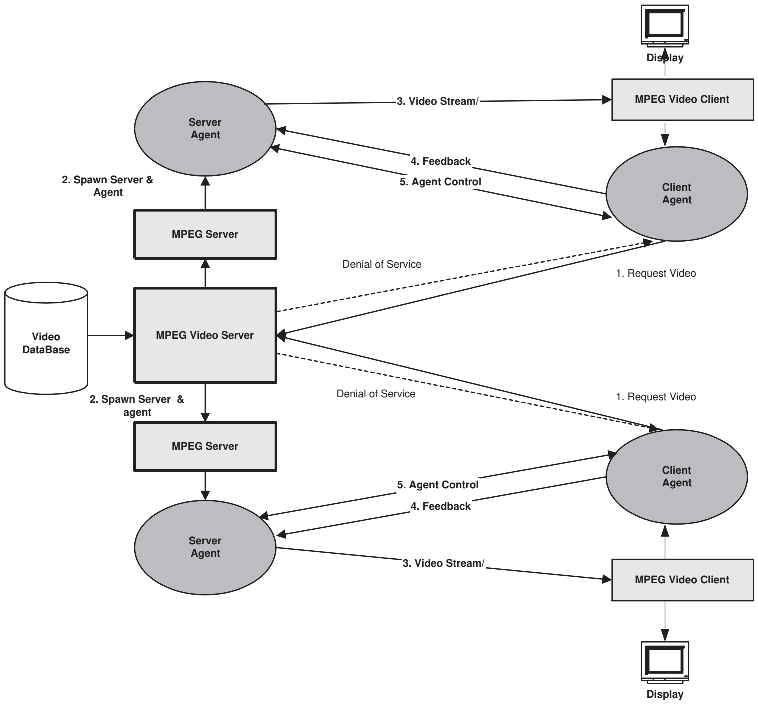


Figure 3: The MPEG system consists of a server and a set of server agents and client agents. The server agents and the client agents are for video streaming and display respectively, and the server agents are created by the server at the request of the client.

times after decoding. Thus this mode is useful to the clients on highly-loaded systems with CPU intensive programs, but connected with high-bandwidth network.

- *Server Decode and Drop.* This mode as the combination of the `Server Decode` and the `Drop B,P Frame` modes is suitable for overloaded clients connected with moderate bandwidth.

Figure 4 shows the streaming modes and the possible transitions among them. The transitions between some modes are bi-directional, thus the server agents are able to changes modes repeatedly, for example, after changing into the drop mode, the server agent might change back to the compressed mode after reserving the bandwidth. When the server agent is initialized, it can reject the service if the service capacity reaches the limit, for example, the video server limits the total number of current clients.

Figure 5 shows the structure of the server agent with two planes: an MPEG plane with a set of strategies corresponding to the streaming modes, and a control plane with the reasoning capability for deciding a streaming mode. The reasoning is performed by the *inference strategy*, which will be discussed in the next section. Using the parallelism supported by the multiplane structure, the MPEG plane is dedicated to the video streaming, while the control plane is continuously reasoning to decide whether to change modes. Figure 5 shows an example of reconfiguration by changing from the compressed mode to the decoding mode. To trigger the state change of the MPEG plane, the control plane invokes the external transition specified in the blueprint. The server agent has the clear separation between the functional parts and decision parts, thus resulting in easy modification, maintenance, and improved reusability.

4.2 The Inference Strategy and the Rules for Server Reconfiguration

The inference strategy using Java Expert System Shell, JESS, inference engine [6] is one of the generic strategies available with the Bond system. The most significant benefit of the inference strategy is that the reconfiguration algorithms can be modified without recompiling the whole agent whereas, in other applications, even small changes in the adaptive algorithms end up with causing large modifications and even re-programming in other parts. Once installed, the inference

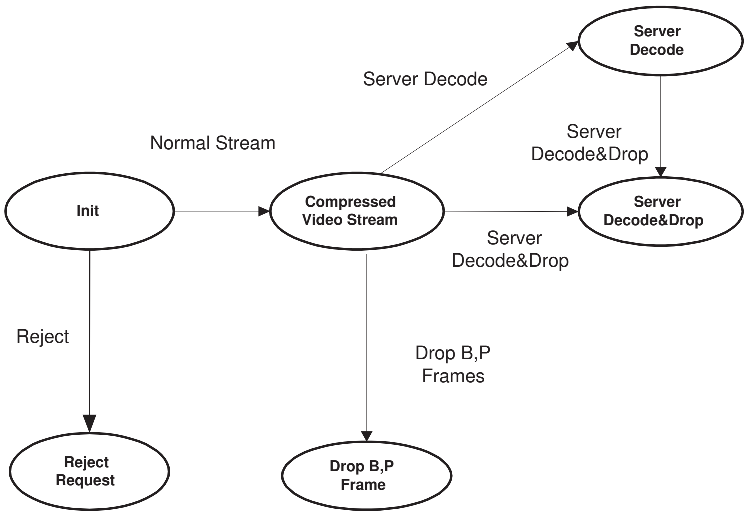


Figure 4: The server streaming modes. Initially the video server is configured to deliver a compressed video stream. The server may by reconfigured to decode frames and send uncompressed frames, to drop B and/or P frames but send compressed frames, or to decode and drop frames.

Frames/sec.	Transmit Rate (bytes/sec.) Bandwidth
5	4000
10	7000
15	10000
20	13000
25	16000
30	20000

Table 1: Video profile listing the frame rates with the required transmit rate/bandwidth

strategy creates a Bond inference engine which wraps the JESS inference engine and the inference engine is capable of communicating with other objects.

In this application, the inference engine uses *facts* and *rules* to reconfigure the video-server. The facts are the performance data of the server, the feedback information of the client, and other state information, e.g. desired frame rate, current streaming mode, current video file name. A set of facts forms the *knowledge base* inside the inference engine. Since the inference engine is the network object like any Bond objects, the engine can gather the feedback information using the KQML messages from the clients. The inference engine exports a set of APIs to insert, delete, modify, and list the facts. The performance and feedback facts are:

- *Transmit rate.* The server logs the transmit rate as bytes/sec. This rate can be affected by the system load on the server side. Each video file has its own profile giving an estimate of the data rate corresponding to a given frame rate. Table 1 shows a sample profile of one of the video files we used for testing. The profile is obtained by summing up each frame size.

- *Packet loss rate.* The current video streaming implementation is based upon UDP and the video quality is affected by lost packets. Each UDP packet contains one frame, thus it is easy for the client to re-synchronize the stream. By comparing the unique frame numbers of the arriving frames, we can detect lost packets. Although each UDP packet contains one frame, the packet loss rate is not the same as the frame loss rate, because P-frames and B-frames are dependent on I-frames. If an I-frame is lost, the depending frames are considered to be lost. The packets arriving out of order are rearranged.

- *Inter-frame time.* The client needs to decode the frame in compressed mode. The inter-frame

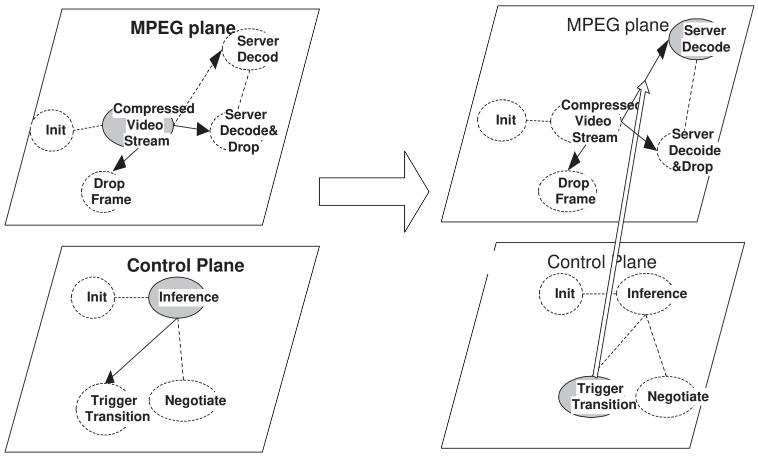


Figure 5: The server agent consists of two planes: the MPEG plane and the control plane. Each plane consists of a state machine, each state has a strategy associated with it. After reasoning, the control plane decides to reconfigure the server

time shows the time between two displayed frames. This time shows reflects the CPU cycles used for decoding for different types of frames. Since I and P frames are larger than B frames, the time to decode them is larger. If the system on the client side is overloaded, the inter-frame time increases.

- *Receving rate.* The client logs the arriving packet sizes as bytes/sec. The receiving rate is affected by a congested network or high system load on the client side. To distinguish two cases, we reserve a bandwidth as specified in the profile. If the rate is still low, it means the client is overloaded.

Rules are sets of conditional statements to control the system configuration and resource reservation. The inference engine load the rules and apply them to the knowledge base. The rules has the following form:

```
Rule:
   [Condition_1]
   [Condition_2]
   ..
   [Condition_N]
   =>
   [Action]
```

As mentioned, the semantics of a rule is: `if [Condition_1] AND [Condition_2] AND ... AND [Condition_N] then [Action]` statement of a procedural language, but it is not intended to be used in a procedural way. Rather than being executed in a specific order by which the rules are listed, any rules with all the conditions satisfied are executed. The conditions of the rules are pattern-matched against the knowledge base and they are *activated* when all the conditions are met. The *firing* of the activated rule is delayed until we invoke a `run` method on the inference engine. Besides the `run` method, the inference engine exports APIs to load the rules, clear or save the current rules, list the activated rules. Since the JESS is the interpreter-based, the rules can be loaded or modified in run time. In this application, the actions of the rules can be the external transition or resource reservation. As shown in Figure 3, the control plane has a *Trigger-Transition* state for invoking external transitions, and a *Negotiate* state for the negotiation with the resource managers about the reservation.

We present the rules for the resource reservation and reconfiguration:

Bandwidth Reservation Rule. The objective of this rule is to reduce the packet loss rate by reserving bandwidth when the network is congested. The profile also has the maximum packet loss rate allowed to maintain a certain frame rate. By comparing the packet loss rate with the maximum rate, we can find out the network is congested. The rule is

```
(packet-loss-rate ?lr)
(desired-frame-rate ?fr)
(maximum-loss-rate ?mr)
(test (> ?lr ?mr))
=>
(reserve-bandwidth ?fr)
```

After this rule is fired, the strategy of the negotiate state looks up the profile to check the required bandwidth to achieve fr, and reserve that amount of bandwidth using the bandwidth reservation interface.

CPU Reservation Rule. This rule is fired when a CPU-intensive program running on either the server or the client side affects the transmit rate of the server or the inter-frame time of the client. The transmission rate of the server is compared with the profile and the inter-frame time is compared to the desired inter-frame time, which can be easily calculated from the desired frame rate. The video profile does not include a CPU component because the CPU requirements are system dependent. Thus this rule is repeatedly fired, and raises the reservation level gradually, until the desired rate is achieved. The rules are:

```
(transmit-rate ?tr)
(required-transmmit-rate ?rtr)
(test (< ?tr ?rtr))
=>
(increase-cpu-reservation)

(inter-frame-time ?ft)
(required-inter-frame-time ?rifr)
(test (< ?rifr ?ft))
=>
(increase-cpu-reservation)
```

Dropping Rule. This rule is fired when either the bandwidth or CPU reservation fails due to the lack of resource. As an action, the drop mode is selected to avoid more congestion or avoid sending more frames than the client can handle in time. If the reservation fails, the new facts about the failure, `(bandwidth-reservation-failed)` , `(cpu-reservation-failed)` and are added to the knowledge base. The rules are:

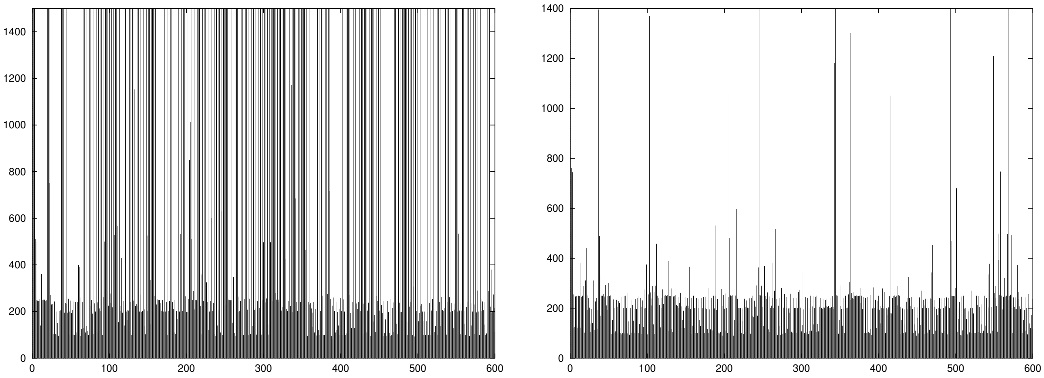
```
(bandwidth-reservation-failed)
=>
(trigger-drop-mode)

(cpu-reservation-failed)
=>
(trigger-drop-mode)
```

5 Experimental Results

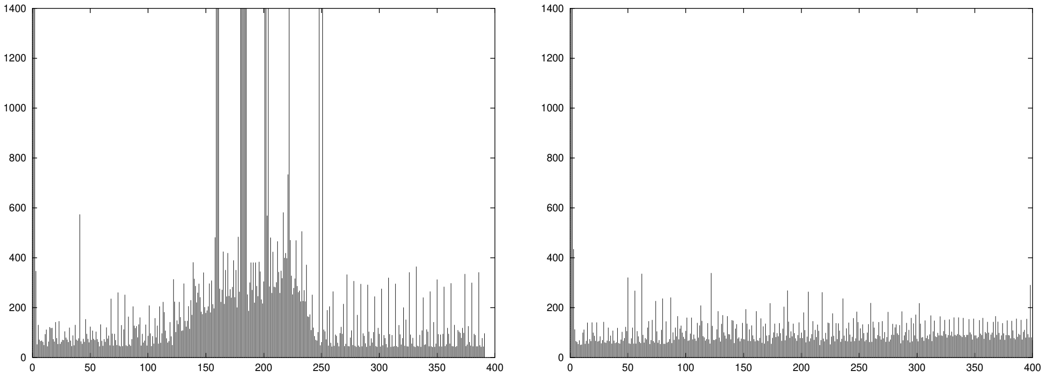
In this section we present measurements characterizing the MPEG application with and without resource reservation. As the testbed for our system, the server is an Ultra Sparc-1 machine with 128 MBytes memory running Solaris 2.5.1, the client is a Pentium II 300 MHz, with 128 MBytes memory machine running Solaris 2.5.1. To simulate increased traffic load a communication-intensive program generates a burst of UDP packets. To simulate the CPU load, a "greeedy" CPU intensive program is used.

The first experiment shows the effect of bandwidth reservation. On the server side in addition to the MPEG application we start the UDP-burst program. The traffic generated by this application interferes with the video traffic and we study the effect of this interference. The results are shown in Figure 6. In 6 (a) the video application does not reserve communication bandwidth. The graph shows the inter-frame times measured at the client site, with frame number on the horizontal axis and time in milliseconds on the vertical axis. The inter-frame time for lost frames is set to the maximum value, 60000 milliseconds, thus lost frames appear as vertical lines in the graph. Without



(a) (b)

Figure 6: The bandwidth reservation experiment.



(a) (b)

Figure 7: The CPU reservation experiment. The graphs show the inter-frame time measured at the client side with the CPU intensive greedy program. On the left, the CPU-intensive program is started while the client is running and then stopped. On the right, the CPU-intensive program could not affect the client because of CPU reservation.

reservation a large percentage of video packets are lost. Once sufficient bandwidth to support the desired frame rate is reserved, the number of lost packets is noticeably reduced , even under the heavy network traffic as shown in Figure 6.

The second experiment shows the effect of CPU reservation. We run three processes of the greedy program to compete for CPU time with the MPEG process. The experiment is repeated two times: the first time without CPU reservation, and the second with sufficient CPU reservation for the MPEG application to achieve its intended frame rate. Figure 7 shows the effect of the CPU reservation on the inter-frame time. To make more visible the effect of the CPU-intensive program, we start it in the middle of client execution at about frame 120 and then stop it at about frame 230. As shown in Figure 7 (a), the greedy program causes an increase in the inter-frame time. With CPU reservation, the CPU-intensive program does not affect the inter-frame time as shown in Figure 7 (b).

6 Conclusions

In this paper we present an agent-based adaptation scheme supporting QoS guarantees. The multimedia agents are assembled dynamically out of reusable components and interact with the client process, with the servers process and with our native OS resource managers for network bandwidth and CPU cycles.

The experiments we conducted show the effect of bandwidth and CPU reservation. Figure 6 shows that the number of lost video packets decreases significantly when sufficient bandwidth is reserved to accommodate the desired frame rate. A similar effect for CPU reservation is observed in Figure ??. In this case the variance of the inter-frame time is reduced when the CPU scheduler provides enough cycles to the video application.

The advantage of the technique described in this paper is greater flexibility and system reconfigurability. The rules governing the behavior of the agents can be modified dynamically. Moreover, the agents themselves can be assembled modified by agent surgery while running.

The Bond systems is available under an open source license from `http://bond.cs.purdue.edu`

7 Acknowledgments

The work reported in this paper was partially supported by a grant from the National Science Foundation, MCB-9527131, by the Scalable I/O Initiative, and by a grant from the Intel Corporation.

References

[1] ISO/IEC JTC 1/SC 29/N 071. *Coding of moving pictures and associated audio - for digital storage media at upto about 1.5 Mbits/s - Part1: Systems, Part2: Video,* 1992. CD11172

[2] M. Hemy, U. Hengartner, P. Steenkiste, and T. Gross. MPEG System Streams in Best-Effort Networks. In *Proceedings of Packet Video'99*, April 1999, New York

[3] J. Walpole, R. Koster, S. Cen, C. Cowan, D. Maier, D. McNamee, C. Pu, D. Steere. A player for adaptive MPEG video streaming over the Internet. In *Proceedings 26th Applied Imagery Pattern Recognition Workshop AIPR-97, SPIE, Washington DC,* October 15-17, 1997.

[4] L. Bölöni and D.C. Marinescu *An Object-Oriented Framework for Building Collaborative Network Agents* in *Intelligent Systems and Interfaces*, (A. Kandel, K. Hoffmann, D. Mlynek, and N.H. Teodorescu, eds). Kluewer Publising House, (1999), (in press).

[5] L. Bölöni and D.C. Marinescu *A Multi-Plane State Agent Model*, September 1999, (submitted).

[6] E. Friedman-Hill. *Jess, The Java Expert System Shell.* Distributed Computing Systems, Sandia National Laboratories, SAND98-8206, 1999.

[7] S. Floyd and V. Jacobson. *Link-sharing and resource management models for packet networks.* IEEE/ACM Transactions on Networking, 3(4), 1995.

[8] H. Sariowan, R. Cruz, and G. Polyzos. Scheduling for Quality of Service Guarantees via Service Curves. In *Proceedings of International Conference on Computer Communications and Networks*, September, 1995.

[9] T. Finin, et al. Specification of the KQML Agent-Communication Language, DARPA Knowledge Sharing Initiative draft, June 1993

[10] I. Stoica and H. Zhang. A Hierarchical Fair Service Curve Algorithm for Link-Sharing, Readtime and Priority Services. In *Proceedings of ACM SIGCOMM 97*, September, 1997.