Middleware QoS Agents and Native Kernel Schedulers for Adaptive Multimedia Services and Cluster Servers*

David K.Y. Yau, Dan C. Marinescu and Kyungkoo Jun
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398
{*yau,dcm,junkk*}*@cs.purdue.edu*

Abstract

Emerging operating systems allow user applications fine grain control over their own resource allocations. This provides a promising approach for user applications to meet their timing constraints. In a distributed environment, however, resource allocation decisions must also be made across machine and traditional OS boundaries. As portable and mobile software entities, middleware *agents* can manage the resource needs of a wide range of applications in this setting. We are conducting research that combines a Java based agent framework (Bond) with Solaris based native OS schedulers (Concerto) supporting performance guarantees. We discuss our application of the synthesized architecture to an adaptive MPEG video server application.

1 Introduction

Emerging operating systems support scheduling with well defined resource specifications. This allows user applications fine grain control over how system resources are being allocated to them. In turn, this provides a promising way for applications to meet their soft real-time constraints. User applications, however, are increasingly being deployed in a network or distributed environment. Resource allocation decisions must therefore also be made across machine and traditional OS boundaries. To meet this requirement, software agent technology holds much potential. As portable and mobile middleware entities, agents can be used to handle the resource needs of a wide range of applications. They can be deployed on demand in a network setting, monitoring system resources and application performance. By exchanging system information with each other and interfacing with applications and kernel schedulers, they can implement highly robust and flexible resource management strategies.

In this paper, we describe our research on agent based resource management for robust quality of service (QoS) provisioning in a distributed manner. We present the Concerto platform which implements our native OS schedulers, the Bond system which implements the middleware QoS agents, and the interface between the two. The synthesized Bond/Concerto system has been applied in building an adaptive multimedia server for MPEG video. We describe the design and implementation of the server.

1.1 Related work

QoS support for general purpose operating systems is a very active research area (see [6, 7] for just a few examples). While various works differ in target resource types, properties of scheduling algorithms, resource management abstractions, compatibility with legacy applications, etc, they all serve as evidence that in the not-too-distant future, user applications will have fine grain control over their resource allocations.

The broad concept of using middleware software objects for distributed resource management is immediately appealing and has been conceived for some time, such as the *QoS broker* architecture in [8]. Darwin [3] is one recent system building effort that employs such an architecture at the high level. While our work and Darwin share some common goals of robust and adaptive network computing, the two works are built on software infrastructures that are designed differently and have different capabilities. As mobile code that can be deployed on network nodes, agents resemble the active network paradigm, although agent programs are not expected to modify

*David Yau and the Concerto project are supported in part by the National Science Foundation under grant number EIA-9806741 and a CAREER award grant number CCR-9875742; Dan Marinescu, Kyungkoo Jun and the Bond project are supported in part by the National Science Foundation under grant number MCB-9527131, by the Scalable I/O Initiative and by a grant from the Intel Corporation.

the network infrastructure itself, thus representing a less radical departure from existing internetwork design.

2 Concerto

Concerto is a general purpose real-time OS platform capable of multiresource reservation, scheduling and signaling. It is being implemented as an extension to the Solaris 2.5.1 kernel and can run Solaris applications "out of the box", without any source code changes or need for recompilation. For the purpose of this paper, we give an overview of the CPU and bandwidth schedulers in Concerto, which have been in production use at Purdue for some time.

2.1 CPU scheduler

Our CPU manager [10, 11] 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, as *leaf* service classes. A resource specification for a (configured or leaf) service class is in the form of a service curve $S(\cdot)$ [2]. Intuitively, $S(t)$ (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 there is sufficient demand.

CPU scheduling interface. Solaris 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 HFSC. At system startup time, we partition CPU capacity into two default service classes: HFSC_C0 and HFSC_C1, that are allocated 90% and 10% of the CPU, respectively. We also changed the kernel to run the `init` process in HFSC_C0 with the linear service curve of slope 0.001. User processes inherit these scheduling paramters by default.

We provide two principal ways to change the default HFSC scheduling parameters. First, we support the `priocntl(1)` command for HFSC 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. HFSC_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 `priocntl(2)` system call. In particular, the `PC_SETPARMS` entry point of `priocntl` can set 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).

2.2 Bandwidth scheduler

The bandwidth manager in our system is called *Tempo*. Tempo allows application flows to reserve guaranteed network bandwidth. It consists of a stream *driver* for control operations and a stream *module* for flow classification and scheduling [4]. Tempo supports very flexible resource sharing according to Internet *flow specifications* based on the five-tuple: `<ip_src, ip_dst, proto, src_port, dst_port>` , i.e. as source IP address, destination IP address, transport protocol, protocol source port number, and protocol destination port number.

The default flow specification (i.e. one with all members of the five-tuple wildcarded) is at the root of each 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. A flow specification can be guaranteed bandwidth resources with a service curve resource specification. In this case, $S(t)$ (in bits) specifies the minimum amount of service that a flow specification with the reservation should receive by time t (in μs), provided that there is "sufficient" demand. Figure 1 illustrates an example Tempo sharing hierarchy.

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). Standard IP packets are routed to go out of a Tempo-enabled interface. Each of them will be classified to a *most specific* flow specification. Tempo implements the *hierarchical fair service curve* algo-

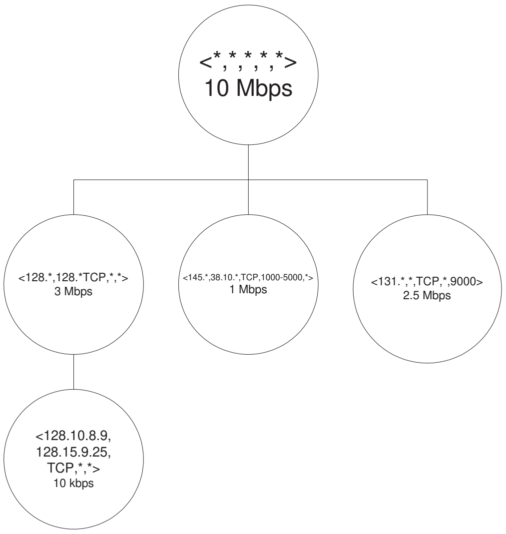

Figure 1: Example Tempo sharing hierarchy by flow specifications.

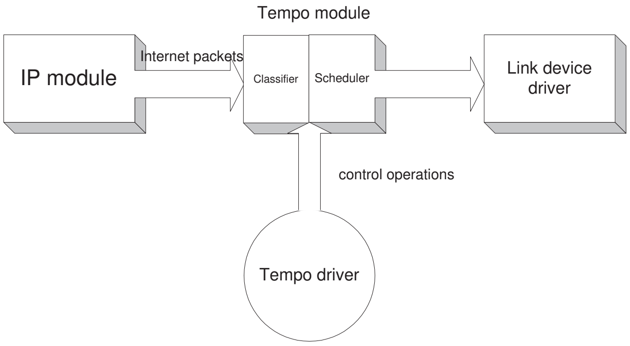

Figure 2: Tempo module/driver architecture.

rithm in [9]. The algorithm guarantees the service curve of each flow while minimizing progress unfairness between them.

Bandwidth scheduling interface. Applications can manage Tempo reservations by opening the appropriate Tempo driver. Various `ioctl` calls serve to create a flow specification with a given resource specification, delete a flow specification given the flow's handle, modify an existing resource specification, list the current sharing hierarchy, etc.

Notice that a Tempo sharing hierarchy 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 QoS agents.

3 Bond Agents

The *Bond agent framework* is an application of the facilities provided by the Bond core [1] to implement collaborative network agents. (The Bond core provides generic support for distributed agent objects, such as naming, inter-object communication, remote object invocation, security, etc.) Agents are assembled dynamically from components in a multi-plane state machine known as its *blueprint* [1], which defines an agent's program in each state as well as valid transitions between states. Programs to be executed by an agent form its active components or *strategies*. They can be loaded locally or remotely, or can be specified in an interpreted programming language, as some *script* of a blueprint. An agent's current knowledge about its execution environment is captured in a single object called its *model of the world*, which allows for easy checkpointing and agent migration. Multiplane state machines describing the behaviors of agents can be modified dynamically, thus allowing for *agent surgery*.

We design agent state machines as multiplanes because the approach can easily express execution parallelism. Different planes can be defined to express various (largely decoupled) facets of an agent's tasks: sensing, reasoning, communication, negotiation, adapting to the environment, and so on. Transitions in a multiplane state machine modify the behavior of an agent by changing its current set of active strategies. These transitions can be triggered either by internal events of computation or by external messages that form a *control subprotocol* of the agent.

4 Adaptive MPEG Video Server

We present an adaptive MPEG system in which video clients can connect to a video server and negotiate a desired level of service using Bond agents as shown in Figure 3. In the figure, an "MPEG video client" is a process responsible for displaying a video stream. This process is started by a "client agent" that monitors reception of the video stream. When a client agent requests a video, an "MPEG video server" spawns a process to deliver the video stream as well as an "MPEG server agent" to control the streaming modes. Two communication channels exist between a client agent and its corresponding server agent: a control channel for streaming commands and perfor-

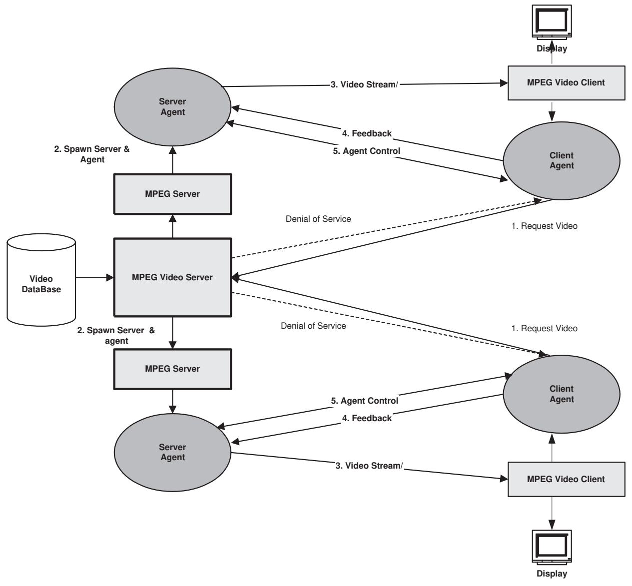

Figure 3: The MPEG agent system architecture.

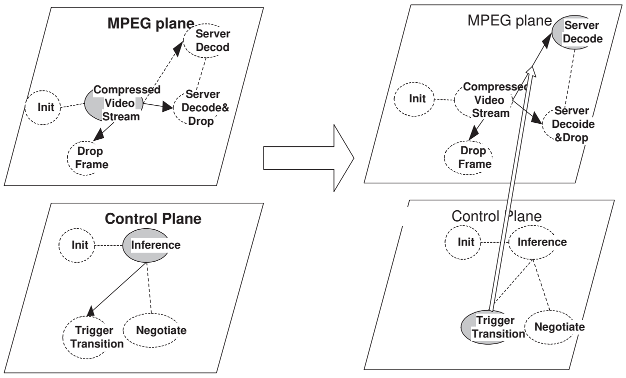

Figure 4: The server agent consists of two planes: the MPEG plane and the control plane. Each plane consists of a state machine, and each state has an associated strategy. The control plane triggers server mode changes based on output from an inference strategy.

mance feedback (from client to server), and a data channel for the streaming video contents.

An MPEG server agent is configured to deliver a video stream in MPEG compressed format when it starts up. As system resource state (measured as various CPU and network loads) changes, however, the agent may gracefully adapt to one of four supported streaming modes:

- **Compressed streaming mode.** Server transmits a stored MPEG video stream as is to a client. The receiving client is responsible for decoding the stream before display. Decoding a video stream can be CPU intensive.
- **B/P frame dropping mode.** Server partially decodes a video stream to identify the *types* of individual frames, and drops frames of certain types (e.g. some or all B frames, all B and P frames, etc). This streaming mode places reduced demand on both the network and client CPU, and can be used when these resources are overloaded.
- **Server decoding mode.** Server decodes a video stream and transmits the raw pixels to a client. This streaming mode can significantly reduce demand on a client CPU (the client has no need to perform MPEG decoding). However, it may require significantly higher network bandwidth (decoded frames can be more than ten times as large as their uncompressed counterparts). This mode is useful when client CPU resources are at a premium but the network has sufficient available bandwidth.
- **Server decoding and dropping mode.** This is simultaneous application of the "B/P frame dropping" and "server decoding" modes. It can reduce client CPU load while not placing much increased demand on network bandwidth.

We now describe the *inference strategy* that governs changes between these server streaming modes.

4.1 Inference strategy and adaptation

Figure 4 shows the structure of a server agent with two planes: an MPEG plane with a set of strategies corresponding to the streaming modes, and a control plane with "reasoning" capabilities for deciding a streaming mode based on current system state. With execution parallelism supported by the Bond multiplane structure, video streaming supervised by the MPEG plane takes place concurrently with inferences by the control plane for possible mode changes. In Figure 4, for example, the control plane invokes an *external transition* in the MPEG plane's blueprint to trigger a transition from the compressed mode to the decoding mode for the MPEG plane. The server agent thus achieves a clear separation between its functional and resource control parts, resulting in easy modification and maintenance, and improved ease of code reuse.

The application specific program that adapts the current streaming mode to system resource state is written as a set of rules for the Java Expert System

Shell (JESS) [5]. They are known as an *inference strategy* and are interpreted by a Bond *inference engine* that interfaces with JESS. The inference strategy, being JESS rules, can be changed easily and even on the fly. One implemented strategy takes as input (as JESS assertions) the following performance data, among others:

- **Transmit rate**. The server keeps a running log of its video transmit rate (in bytes/second). This rate is affected by current server CPU load.
- **Packet loss rate**. We currently implement video streaming on top of UDP, which is susceptible to packet loss. By consecutively numbering UDP packets sent from the server side, we allow the kernel of a client machine to easily detect packet losses as gaps in the received sequence numbers. In Concerto, such information can be collected in a kernel module called *Atempo*, and can be read by a client agent.
- **Inter-frame display time**. We measure the times between frames displayed by a client. Either network congestion or increased client CPU load will cause an increase in the measured inter-frame times.

Our general inference strategy is based on the following design arguments. There are three resources on an end-to-end streaming path: server CPU, network, and client CPU, any one of which is a potential bottleneck limiting end-to-end performance. The system can identify a bottleneck based on the above performance data: If the server transmit rate is low, then the server CPU is a bottleneck; if the network loss rate is high, then network bandwidth is a bottleneck; if the client display rate is low *and* the network loss rate is not high, then the client CPU is a bottleneck. Once a performance bottleneck is identified, an adaptation strategy can react accordingly. For example, one "bandwidth reservation rule" states that if the measured packet loss rate at client side is above some threshold, then the rule should trigger a request to increase bandwidth reservation. If the request did not succeed, then the system should trigger the picture dropping mode.

5 Conclusion

We described the Bond/Concerto system for adaptive and robust network computing. We presented an example application of MPEG video streaming to illustrate the utility of our system. Work is in progress to construct a Web cluster server that can perform content based load distribution, and offer performance guarantees for real-time services such as multimedia streaming.

A public binary release of our CPU and bandwidth schedulers as an extended Solaris 2.5.1 kernel is available from `http://ssal.cs.purdue.edu/concerto.html`. Reference source code will be available for institutions with Solaris 2.5.1 source code license. If interested, please send email to `ssal@cs.purdue.edu`. The Bond agent system is available under an open source license from `http://bond.cs.purdue.edu`.

References

[1] L. Boloni and D.C. Marinescu. *An Object-oriented framework for building collaborative network agents.* Kluewer Publishing House, 1999. In press.

[2] R. Cruz. Quality of service guarantee in virtual circuit switched network. *IEEE JSAC*, 13(6):1048–1056, August 1995.

[3] P. Chandra et al. Darwin: Customizable resource management for value-added network services. In *Proc. ICNP 98*, Austin, TX, October 1998.

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

[5] E. Friedman-Hill. Jess, the java expert system shell. Technical Report SAND98-8206, Sandia National Laboratories, 1999.

[6] K. Jeffay, F. D. Smith, A. Moorthy, and J. Anderson. Proportional share scheduling of operating system services for real-time applications. In *Proc. IEEE Realtime Systems Symposium*, December 1998.

[7] M. Jones, D. Rosu, and M-C Rosu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proc. ACM 16th SOSP*, Saint-Malo, France, October 1997.

[8] Klara Nahrstedt and Jonathan M. Smith. The QoS broker. *IEEE Multimedia Magazine*, 2(1):53–67, Spring 1995.

[9] I. Stoica, H. Zhang, and T.S. Eugene Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. In *Proc. ACM SIGCOMM 97*, September 1997.

[10] D. K. Y. Yau. Decoupled delay and rate guarantees for cross domain thread scheduling. Technical report, Purdue University, November 1998.

[11] D. K. Y. Yau. Performance evaluation of CPU isolation mechanisms in a multithreaded OS kernel. Technical report, Purdue University, October 1999.