End System Support for Networking with Quality of Service Guarantees*

David K.Y. Yau and Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188
{yau,lam}@cs.utexas.edu

Abstract

In part due to the emergence of continuous media applications having real-time constraints, networking with quality of service (QoS) guarantees has become an increasingly important goal. Towards this goal, we present an end system architecture with the following major components: (1) adaptive rate-controlled (ARC) scheduling for time shared resources in an end system, such as CPU and network interface, (2) a framework of *Migrating Sockets* for user level protocols that minimizes hidden scheduling in protocol processing, and (3) based on "exclusive" packet receiver information exported by Migrating Sockets, a constant overhead packet demultiplexing mechanism suitable for wide-area internetwork communication (*active* demultiplexing). We have an implementation of our architecture in Solaris 2.5.

1 Introduction

Recent explosive growth of the Internet and the increasing proportion of audio and video packets in network traffic show a strong "customer push" for continuous media (CM) applications in future computer networks. On the side of "technology pull", research on integrated services networks, which can provide deterministic or statistical quality of service (QoS) guarantees to traffic flows (through appropriate admission control and packet scheduling policies), has made great strides. In the Internet, protocols such as RSVP and IPv6 promise to avail user applications of standardized access to reserved network resources. However, network level guarantees alone are not sufficient for satisfying the real-time constraints of user applications. QoS support must similarly be provided in the operating system and communication subsystem of network hosts.

End system support for networking with QoS guarantees is a challenging problem. To meet their timing constraints, user applications must be given guaranteed access to diverse system resources, including time shared resources such as CPU and network interface, and space shared resources such as memory buffers. Morover, the run time environment for protocol processing, which provides such services as timer management, buffer management and demultiplexing table lookup, must be such that it can deliver predictable performance.

In this paper, we present an end system support architecture that addresses these and other issues for next generation networking with QoS guarantees.

*Research supported in part by National Science Foundation under grant no. NCR-9506048, a grant from AT&T Foundation, and an IBM graduate fellowship awarded to David Yau for the 1996–97 academic year.

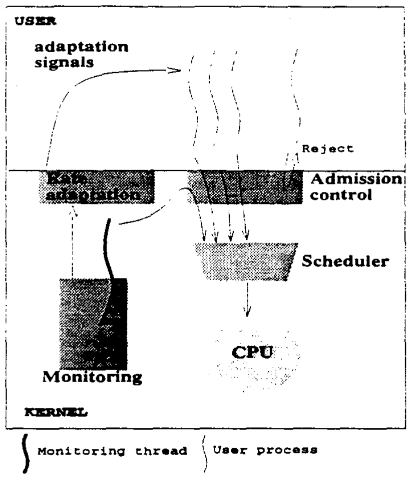


Figure 1: ARC CPU scheduling framework.

2 ARC Scheduling

We propose a family of *adaptive rate-controlled* (ARC) CPU schedulers for intergrated scheduling of CM and various other applications [9]. ARC schedulers have the following properties: (1) reserved rate can be negotiated, (2) QoS guarantees are conditional upon thread behavior, and (3) firewall protection between threads is provided. Figure 1 shows the ARC scheduling framework in a general purpose computing environment. The functional components of the framework are described next.

2.1 Reservation and admission control

We use a rate-based reservation model for ARC scheduling. Using this model, an application thread reserves CPU time using two parameters: a rate r ($0 < r \leq 1$), and a time interval p (in μs) known as period. The rate can then be viewed as a guaranteed fraction of CPU time that the thread will be allocated over time intervals determined by p. Subject to the admission control criterion that the total reserved rate in a system should not exceed one, the ARC scheduler in [9] provides the following progress guarantee: a "punctual" thread with rate r and period p is guaranteed at least krp CPU time over time interval kp, for $k = 1, 2, \ldots$.

For motivation of the rate-based reservation model, consider a video application that sends pictures to a network at a rate of 30 per second. Roughly every 33.3 ms, a video capture board digitizes and compresses a picture, and buffers the compressed picture in on-board memory for reading by the video application. The video application reads and packetizes the picture, and sends packets to the network using some protocol stack. The application has an execution profile shown in Figure 2. The vertical lines mark the times at which pictures arrive to the application. Processing time for each picture, which includes reading and packetizing picture data, as well as protocol processing of the packets, is about x ms. For minimal delay, processing of a picture should complete before the next picture arrives. The CPU requirements of the video application can be specified with $r = x/33.3$, where $x < 33.3$, and $p = 33.3 \times 1000$ in our reservation model.

Besides CM applications, the rate-based model is natural even for applications that are not "real-time" and not inherently periodic [9], such as a numerical analysis application solving a system of linear equations.

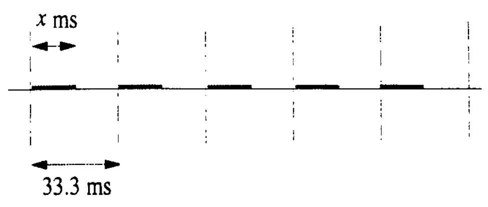


Figure 2: Execution profile of a video application.

2.2 Rate-controlled on-line scheduler

One goal in ARC scheduling is to provide progress guarantees such that the guarantee to a thread is independent of the behavior of other threads in the system. Hence, an algorithm with the *firewall property* should be used for on-line CPU scheduling in our system. A scheduler having this property, described in [9], schedules threads in increasing order of *RC values* of threads. The RC value of a thread encodes the *expected finishing time* of a previous computation performed by the thread. Informally, the expected finishing time is the time at which the previous computation would complete had it proceeded at the reserved rate. Hence, threads that have been getting more than their expected shares of the CPU will get higher RC values than threads that have been getting less than their shares.

To enforce rate reservations, RC values are recomputed for certain threads when one of three *rescheduling points* occurs in our system: (1) when a running thread becomes blocked, (2) when a system event causes one or more blocked threads to become runnable, and (3) when a periodic clock tick occurs. After RC values have been recomputed, a thread with the currently lowest RC value is chosen for execution. We note that RC value recomputation is very simple and has to be done only for the currently executing thread at a clock tick. Moreover, the RC value of the currently executing thread will not change at most clock ticks, thus avoiding excessive context switching.

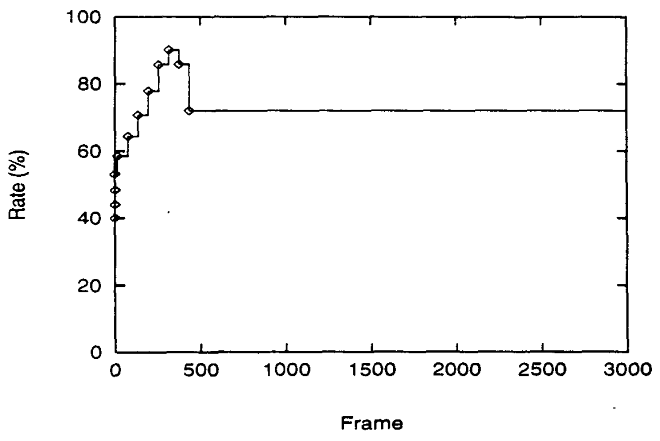
2.3 Rate negotiation

Rate negotiation in ARC scheduling is enabled by a montoring module and a *rate adaptation* interface. The montioring module measures the actual execution rates of threads in the system. For each thread that requests rate-adaptation, the monitoring module periodically computes two performance values: a *lag* (in μs) and a *lax* (in %). The lag value measures how far ahead a thread is running ahead of its reserved rate. The lax value measures the percentage of CPU time unused by a thread during the last monitoring period. A large lax value indicates that the thread may have a reserved rate that is too high.

Using information reported by the monitoring module, the rate adaptation interface allows a thread in a user process to discover and react to "significant" mismatches between its reserved and actual rates. The system delivers *signals* to a thread that has a lag or lax value exceeding some specifiable threshold. The thread may then react to the signals in an application specific manner. Rate negotiation is highly desirable for CM applications having medium to long term changes in actual rate.

Figure 3 shows the performance of rate adaptation for a video application sending pictures to a network at a rate of 30 per second. The application was started with a rate of 0.4, when in fact it needed a rate of about 0.7 to achieve the target frame rate. From the figure, the rate of the application stabilized at 0.721 after an initial *adaptation phase*, during which the application "hunted" for an appropriate rate to use. Note that, during the adaptation phase, a frame was delayed by close to one frame time about every two seconds. This is because the application had to handle a rate adaptation signal every monitoring interval, which was chosen to be two seconds.

(a)



(b)

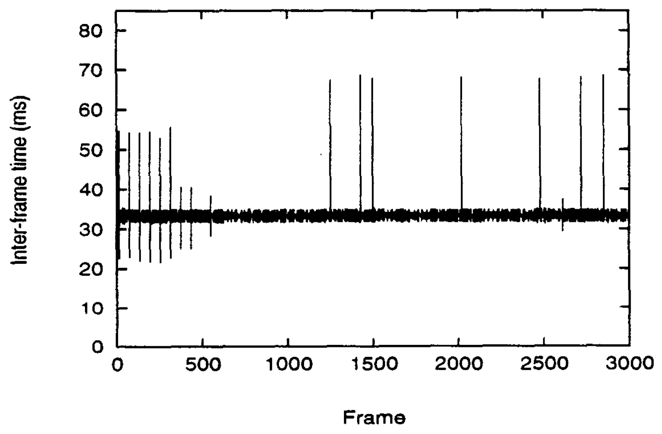


Figure 3: Profile of (a) rates and (b) inter-frame times for `video` with rate adaptation from an initial rate of 0.4.

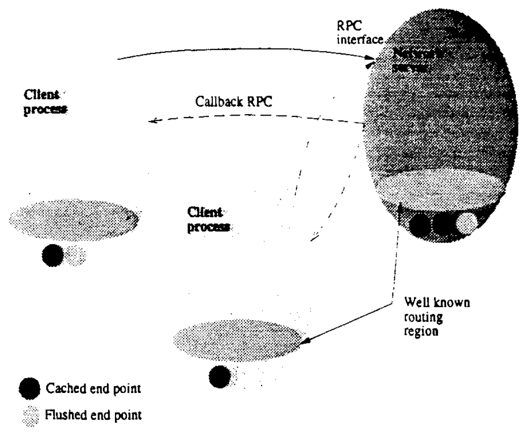


Figure 4: The protocol implementation model of Migrating Sockets.

3 Migrating Sockets

For protocol processing, our architecture adopts a user level implementation model supported by the framework of *Migrating Sockets* [8]. Migrating Sockets takes its name because the state and management right of a socket (accessing a network endpoint) in the framework can *migrate* between an application process and a network server process. The network server is used to handle "special" semantics like shared access to a socket by multiple processes, and "boundary" operations such as connection management. However, when certain conditions are met (such as when the socket is not shared and the network connection it accesses has been fully established), a socket can be managed directly by an application process for efficiency. A socket managed by an application process is said to be *cached* by the process. Otherwise, the socket is *flushed.*

Figure 4 illustrates the protocol implementation model of Migrating Sockets. Two cached and one flushed sockets are shown. A client process and the network server communicate with each other using RPC and callback RPC. The well known routing region shown in the figure enables extremely efficient sharing of dynamic routing information between client processes and the network server process [8].

Migrating Sockets is compatible with Unix semantics and the widely used Berkeley socket API. This allows us to run a large base of Unix multimedia applications (such as the mbone suite of teleconferencing tools) with minimal modifications in our system. For efficiency, Migrating Sockets runs as the protocol processing component in an OS architecture we have previously designed and prototyped [7]. The OS architecture is shown in Figure 5.

User processes in our architecture send packets by appending send requests to a *send control queue.* Conversely, a network device driver informs user processes of packets received by appending receive notifications to a *receive control queue*. Send requests and receive notifications contain only control information, but not the actual data, for send and receive. For efficient control transfer, the control queues are shared between user processes and the kernel.

For transfer of actual data, user processes may allocate network buffers. For a network interface that uses DMA, network buffers for the interface are backed by DMA resources. On the send side, a user process can put data in network buffers for direct sending to the network (i.e., without any intermediate data copies). On the receive side, a device driver can put received data in network buffers for direct access by a user process. A kernel thread (with well defined scheduling parameters) can be used to handle send requests by user processes without relying on explicit system calls. The kernel thread can also provide rate-based flow control to reserved-rate network connections in future integrated services networks.

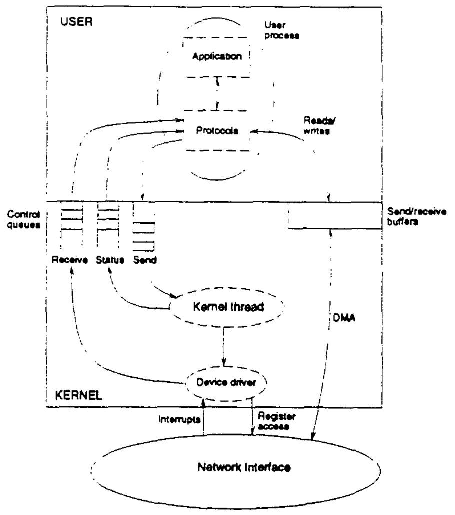


Figure 5: OS architecture for multimedia networking.

Hidden scheduling

From a quality of service perspective, Migrating Sockets works well with ARC scheduling by minimizing "hidden" scheduling. Hidden scheduling occurs when protocol processing occurs in interrupt context or in a system thread of control not belonging to an application process. To illustrate, consider protocol implementation using *streams* [3]. In streams, network send and receive can take place in *service routines* run by "background" system threads, over which applications have no scheduling control. This presents difficulty with ARC scheduling, in that it is hard to determine suitable rates of progress for the system threads, such that user applications can meet their timing constraints.

In contrast, Migrating Sockets minimizes hidden scheduling. In fact, code for network send and receive is linked with an application as a user level library. It is run by threads that are an integral part of the application process. With the bulk of receive side protocol processing performed by application threads of control, the role of network interface interrupt handling is solely to demultiplex packets to their destination processes. This significantly reduces time spent for code in interrupt context, for which it is hard to control the rate of progress.

4 Active Demultiplexing

Traditional packet filters for receive side packet demultiplexing, while highly flexible, incur processing overheads that increase with the number of network endpoints in a system. Since predictable performance is an important goal of our architecture, Migrating Sockets supports a constant overhead packet demultiplexing mechanism known as *active demultiplexing* [8]. The basic idea is that, under certain conditions, an OS handle identifying a receive process can be included in packets destined for that process. A receive end system can then make use of the OS handle to deliver packets directly to the receive process, without any searching.

Currently, active demultiplexing exploits the notion of *exclusive packet receivers* exported by Migrating Sockets. The notion, which describes network endpoints, has the following meaning: if a packet is delivered to a network endpoint that is an exclusive packet receiver, then no other endpoints in the system should receive a copy of the packet. With the notion defined, active demultiplexing, if enabled, is triggered when a socket accessing an exclusive packet receiver becomes cached. The process caching the socket *advertises* to the socket's *peer endpoints* (i.e. endpoints from which the socket has received packets) the OS handle used for receiving. The peer endpoints can then *enclose* the advertised handle in subsequent packets they send to the process, thereby enabling active demultiplexing of these packets when they are received. OS handles can later be *revoked* when they are no longer valid.

We have an implementation of active demultiplexing in the context of TCP/IP. Our implementation uses TCP options for handle advertisement and revocation. OS handles are enclosed as an IP option. Use of TCP/IP options *transparently* enables active demultiplexing if both hosts in a TCP connection support it. For security reasons, we include a *nonce* field in the OS handle used for active demultiplexing. Nonces have the property that each newly generated one should have a fresh value. They can, for example, guard against obsolete OS handles advertised for exclusive packet receivers that are no longer cached.

Table 1 shows a breakdown of the per packet active demultiplexing overhead in our current system. The dominant cost is in inserting the IP option for enclosing OS handles. This cost can be significantly reduced by an optimization that stores the option in a template header for packets sent from relevant sockets.

Machine	Insert IP option	Kernel level security overhead	Total overhead
SPARC 10	5.9	0.958	6.858
Ultra-1	2.5	0.281	2.781

Table 1: Breakdown of per packet active demultiplexing overhead (in μs).

When active demultiplexing cannot be employed or is not preferred, Migrating Sockets makes use of packet filters.

5 Related Work

Techniques from the real-time literature have been proposed for scheduling CM applications [4]. Two algorithms that are generally believed to be suitable for this purpose are the rate monotonic and earliest deadline first algorithms. We rejected a straightforward implementation of either algorithm in a general purpose computing environment because they lack the firewall property. Our proposal that threads should be shielded from each other for progress guarantees is similar in objective to *processor capacity reserves* [2], which has been implemented in Real-Time Mach. However, unlike processor capacity reserves, ARC scheduling does not require an explicit mechanism for replenishing reserves and *depressing* thread priorities on reserve depletion.

Migrating Sockets draws on previous experience in user level protocol implementation [1, 5]. For example, it uses a similar partitioning of protocol processing between a network server process and client processes. Our end system support architecture, however, integrates Migrating Sockets with a novel protocol/kernel interface, and provides ARC scheduling of protocol threads.

Active demultiplexing functions similarly as *active messages* [6], which has been designed for a Local Area Network environment. Our proposal is for wide-area internetworking, such as in the context of TCP/IP.

6 Conclusion

We presented an end system architecture for networking with QoS guarantees. In addition, the architecture has other advantages, namely efficiency, flexibility and backward compatibilty with existing systems. We plan to use the architecture in developing communication protocols for evolving network infrastructure and application requirements. More detailed descriptions of components in our end system architecture can be found in [7, 8, 9].

References

[1] Chris Maeda and Brian N. Bershad. Protocol service decomposition for high-performance networking. In *Proc. 14th SOSP*, pages 244–255, December 1993.

[2] C.W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proc. IEEE Int'l Conf on Multimedia Computing and Systems*, Boston, MA, May 1994.

[3] D.M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[4] Ralf Steinmetz. Analyzing the multimedia operating systems. *IEEE Multimedia Magazine*, 2(1):68–84, 1995.

[5] C.A. Thekkath, T.D. Nguyen, E. Moy, and E.D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Trans. Networking*, 1(5):554–565, October 1993.

[6] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proc. 19th ISCA*, pages 256–266, May 1992.

[7] David K.Y. Yau and Simon S. Lam. An architecture towards efficient OS support for distributed multimedia. In *Proc. IS&T/SPIE Multimedia Computing and Networking*, pages 424–435, San Jose, CA, January 1996.

[8] David K.Y. Yau and Simon S. Lam. Migrating sockets for networking with quality of service guarantees. Technical Report TR-97-05, The University of Texas at Austin, Austin, Texas, January 1997.

[9] David K.Y. Yau and Simon S. Lam. Adaptive rate-controlled scheduling for multimedia applications. *IEEE/ACM Transactions on Networking*, August 1997 (to appear). An earlier version in *Proc. ACM Multimedia*, Cambridge, MA, November 1996.