15

Supporting Multiple-tier QoS in a Video Bridging Application

Rajeev Koodli and C. M. Krishna
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003, U.S.A.
phone: 413-545-0715, fax: 413-545-1993
e-mail: koodli,krishna@ecs.umass.edu

Abstract

With the emergence of multimedia technology, it has become important to be able to support a diverse set of client applications (simply called *applications* henceforth), each with its own Quality-of-Service (QoS) requirement. Such *soft* real-time applications exhibit remarkable loss tolerance capacity, provided the losses are well regulated. In this paper, we propose the notion of a *noticeable loss*, which directly relates loss pattern to the perceived QoS for an application, and use it to evaluate a novel resource management algorithm that attempts to provide acceptable QoS to individual applications. We apply our model to a real-world application, and provide simulation results to illustrate the performance of this algorithm.

Keywords: deadlines, loss, end-to-end, scheduling

1 Introduction

With the dramatic improvements in processor and network speeds and data compression techniques that have occurred over the past few years, it has now become possible to support highly demanding multimedia applications such as video teleconferencing. However, such applications each have a certain Quality of Service (QoS) requirement, which translates to delay-sensitive demand on the underlying resources. In order to exploit the gains in hardware and support *multiple* multimedia applications with their respective QoS requirements, application *scheduling* becomes critical. This paper addresses the issue of scheduling applications so that their respective QoS requirements are satisfied in a Video Bridging system that facilitates live video conferencing.

Building QoS into Distributed Systems A. Campbell & K. Nahrstedt (Eds.)

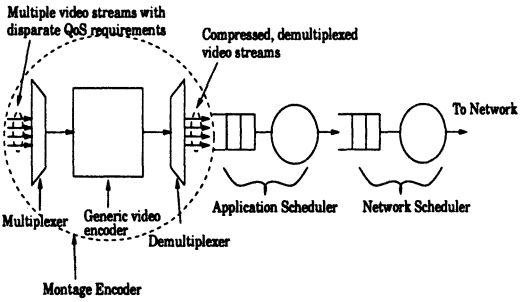


Figure 1: A Video Bridging Server Architecture

It is important to note that an application such as video conferencing requires access to *multiple* resources such as an encoder, the operating system (for access to memory and I/O) and the network. Hence, it becomes important to meet its QoS requirements at each of the individual resources which are shared by other applications, even though the application itself does not specify how its *end-to-end* QoS requirements have to be met at the individual resource nodes. Thus, effectively the scheduling problem becomes one of representing the QoS at each node and supporting it locally for a variety of applications.

The QoS provided to any application is a function of the scheduling at *(a)* the network, and *(b)* the end-system server such as a video bridge. Ideally, there would be one scheduler managing resources at both the network and the server. Such a scheduler would have perfect knowledge of the current state of the entire system (network plus server), and would therefore be able to manage resources efficiently. However, such a global scheduler is infeasible, and we must make do with a distributed scheduling solution. The overall management strategy is to divide the problem into network and server components, and to solve them separately. That is, we decompose the overall scheduling problem into separate network and server scheduling problems. Even at the server, it is frequently infeasible to have a scheduler overseeing all the scheduling there: quite often, the server itself consists of many subunits, for which a distributed scheduler is the best practical solution. While much work has been done in supporting QoS requirements in the networks area [1, 3, 8, 9], relatively little research has been done to provide desired QoS levels in end-system servers. That is the topic of this paper.

In this paper, we use a performance metric called *Noticeable Loss Rate* for evaluating the QoS offered to an application by a server. A task belonging to an application is considered lost if it misses its end-to-end deadline. The loss of a task is said to be *noticeable* if it occurs within an interval bounded by its previous loss instance and an application-specified time bound δ. Using NLR as a performance metric, we introduce the approach of Preemptive Abortion and Cycle Stealing (PACS), which attempts to reduce the NLR of various applications by exploiting their respective time bounds, δ, which we call *loss constraints*. PACS carries out scheduling at individual resource nodes such that all applications are provided an acceptable NLR. The effectiveness of the PACS scheme is evaluated using simulation experiments.

As a motivation for this work, consider the video bridging system shown in Fig-

ure 1. This set-up is borrowed from a prototype at Bell Laboratories[4]. The input to the Multiplexer can arrive from *different* applications with disparate QoS requirements, i.e., the streams can possess different rates of frame generation, deadlines and loss tolerance. Such a sharing of a bridging system among different applications is often necessary for financial reasons. Once we have multiple applications contending for the same resource, scheduling becomes critical. The demultiplexed streams belonging to different applications will have to be scheduled by an Application Scheduler based on a delay-sensitive scheduling discipline such as Earliest Deadline First, or other real-time scheduling disciplines such as Rate Monotonic [10] or Cyclic Executive [2] in order to meet their timing requirements. So, the queue associated with the Application Scheduler in Figure 1 can be a priority-ordered queue based on deadlines (or some other parameter) or a simple FIFO queue that is serviced cyclically. The output of the Application Scheduler is serviced by the Network Scheduler which breaks the application data into smaller sized *cells* and transmits them to the network. Note that the Network Scheduler may augment scheduling with traffic compliance policies, as in ATM systems. The video streams have to traverse many resources such as the Encoder, the Application Scheduler (typically the OS) and the Network Interface Unit. In summary, it is clear that a facility such as a video bridging system will have to support multiple applications with individual QoS requirements and such support will have to be on an *end-to-end* basis.

The rest of the paper is organized as follows. We describe the system model and provide an overview of the solution approach in Section 2. The algorithm is described in Section 3, and numerical results are provided in Section 4. We conclude with a brief discussion in Section 5. Due to space limitations, the literature survey and parts of the original paper have been moved to [7].

2 System Model

We make the following assumptions in this paper.

1. There are multiple applications contending for the available resources. Applications are classified according to their arrival pattern and QoS requirement. There are multiple classes of applications, each with its own distinct arrival pattern and QoS requirement. Each application class has its own priority assigned to it by a higher level entity such as a Call Admission Process, which is orthogonal to the scheduling problem addressed in here.

2. An application with end-to-end processing requirement generates *complex* tasks. Each complex task consists of various *subtasks* which execute at different resources. For example, processing a video frame generated by a camera is a complex task, consisting of subtasks for processing at the Encoder, the Operating System and the Network Interface. Subtasks of complex tasks belonging to different applications compete for resources.

3. An application that generates complex tasks specifies an *end-to-end deadline* and an acceptable *noticeable loss rate*. The end-to-end deadline is the time by which the processing has to be completed at the end-system server such as the video bridge shown in Figure 1. The subtasks of a complex task have no deadlines associated with them; the only specified deadline is the end-to-end deadline associated with the complex task. A complex task which misses its end-to-end deadline is assumed

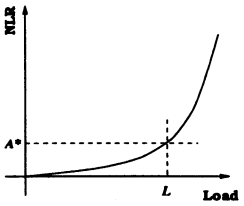


Figure 2: QoS Function

lost.

The loss of a complex task of an application belonging to a class i is said to be *noticeable* if the interval between its loss and the previous loss for that application is no greater than δ_i, where δ_i is a *loss constraint* specified for that application class. QoS is specified in terms of *noticeable loss rate*, NLR.

Example 1. Figure 2 shows the QoS function for some application class. A^* is the maximum acceptable NLR for that application class, i.e., if the noticeable loss rate is greater than A^*, the QoS is no longer acceptable. Ideally, a scheduling algorithm should ensure that all applications are provided acceptable QoS. Note that the loss constraint and the acceptable NLR (ANLR) are related. For example, if δ is 1 second, then, in the worst case, a task may be lost every 1 second and the QoS will still be acceptable to the application. Within a window of 1 second, if 30 tasks are generated, then the ANLR is 1/30, i.e., 3.33%.

The concept of a noticeable loss is important to effectively capture the loss tolerance capacity of soft real-time multimedia applications. It imposes a restriction on how a service provider such as a network or a server may carry out resource management while supporting a large number of requests and during periods of overload. Typically, most resource management strategies simply abort tasks during overloads; some actually do so within the bounds of normally accepted *aggregate* task loss. For example, an application may specify an aggregate task loss of 3%, but, this specification may allow too many *consecutive* losses to occur within a short window of time, which results in poor perceived quality for an application. Consecutive losses result in poor QoS for an application [13]. By placing a time bound on successive task losses, the concept of a noticeable loss forces a service provider to adhere to providing better perceived quality for an application.

4. We do not know the exact execution time of each subtask; however, we do have an estimate.

Given these conditions, we now address the problem of providing acceptable NLR for all applications belonging to various classes.

3 The Algorithm

There are two parts to our approach. In the first part, we concentrate on assigning virtual deadlines to subtasks so that their timely processing improves the chances of meeting the end-to-end deadline of the complex task that they are made of.

In the second part, we present a resource management scheme that attempts to meet the QoS requirements of various applications during periods of overload and unpredictable service requirements. The algorithm works with a given call admission control (CAC) process. Its impact on the CAC, though beneficial, is not the subject of this paper.

3.1 Assigning Nodal QoS parameters

An application only specifies an end-to-end deadline and an end-to-end NLR. These values have to be apportioned among various resources to facilitate nodal (i.e., corresponding to individual resource nodes) support including call admission and scheduling. Assigning nodal deadlines, which we call *virtual* deadlines, follows a similar approach to other work [5], but differs from them in *how* it assigns virtual deadlines. The subtasks belonging to various complex tasks are assigned virtual deadlines based on their *flexibility*, which is defined as the ratio of available slack to estimated service time. The virtual deadlines are derived as follows.

$$d_t = s_x + s_x/(\mu d_f) \tag{1}$$

In the above expression, μ is the mean task service time, and s_x is the estimated service time. d_f is used as a control variable to vary the second term on the RHS, which represents the amount of estimated available slack to a subtask. If we take the ratio of the second term on the RHS to the first term in Equation (1), we get $1/(\mu.d_f)$, which is a constant for all the subtasks. Thus, the above assignment allocates *equal* flexibility to all subtasks at a node. Experimental work has shown equal flexibility to be a superior approach [5] (a detailed description of the aforementioned assignment, including its performance is presented in [6]).

Recall that each application is associated with a loss constraint. This constraint can be apportioned among the various resource nodes of the server, thus converting a single loss constraint to individual loss constraints for each of these nodes. The apportionment process can follow any heuristic; in our own runs, we divide the loss constraint equally among all the nodes [9].

3.2 Handling Overloads and Unpredictable Service Times

The flexibility-based deadline assignment results in a fair share of the resource for all subtasks and hence enhances their chances of meeting the deadline requirement. However, as the load increases (i.e., as more requests are admitted), subtasks may miss their assigned deadlines due to the variance in processing times. It is important to note that a virtual deadline expiration need not result in subtask abortion. Instead, an attempt may be made to extract more processing time for such subtasks. It is clear that any scheme that attempts to extract more time for a tardy subtask would have to respect the QoS requirements of other tasks.

We begin presenting the Preemptive Abortion and Cycle Stealing (PACS) scheme with a simple algorithm that follows a dual-timeline verification. See Figure 3. The estimated processing times of the two subtasks are shown by the rectangular boxes. In step 1, the algorithm verifies if a subtask in the task queue has already missed its assigned deadline. If it has, it is simply skipped and the verification is done for the

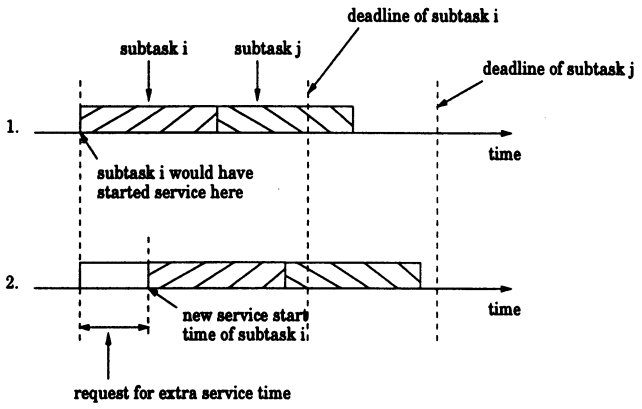


Figure 3: Dual-timeline verification

next subtask in the queue. If a subtask has not already missed its assigned deadline, then, in step 2, the algorithm verifies if it will still meet its assigned deadline if the requesting subtask is allowed to continue in service. If so, the process is repeated for the next subtask in the queue. If no subtask that has not already missed its assigned deadline will miss it as a result of extending service to the requesting subtask, the request is honored. We now extend the concept of dual-timeline verification to allow for preemptive abortion. Note that in step 2 of the dual-timeline algorithm, a bottleneck subtask (one whose assigned deadline would expire if the requesting subtask is allowed to continue) may be eligible for preemptive abortion. A subtask is said to be *eligible* for preemptive abortion if the previous instance of loss associated with its application and the current time differ by more than the application's nodal value of the loss constraint, since, then we can abort such a subtask without affecting the NLR for the application. Such an abortion can yield valuable time for a tardy subtask which might be in danger of experiencing a noticeable loss.

The PACS algorithm is invoked when an assigned deadline expires without the subtask having finished its execution. At this time, the system scheduler – which manages the scheduling of various subtasks at a resource – has to decide whether it should allow the subtask to continue in service for an additional time, which is again estimated. The scheduler consults the PACS algorithm, which in turn decides whether the subtask can be allowed to continue its execution. The system scheduler is assumed to share the task-specific information, usually called Process Control Block (PCB) or Task Control Block (TCB) of various applications with the PACS algorithm.

The pseudo-code of the algorithm is given in Figure 4. An illustration of the algorithm follows the pseudo-code. The parameters used in this algorithm are shown in Table 1. Note that a subtask belongs to an application, which in turn is a member of a class. So, while A_i refers to the previous instance of abortion of a subtask belonging to an application i, δ_i denotes the loss constraint of class i to which the application belongs.

If the algorithm can accommodate the requesting subtask, then the amount of

```
/* Initialize. We start from subtask 1 in the queue */

Ψ = {0}
θ = 0
i = 1 /* a local variable */
T* = T /* T* is a local copy of system time T */

[1] For all the subtasks in the subtask queue,
      if (T* ≥ D_i)
       /* subtask i has already missed its assigned deadline */
            Increment i, goto next subtask
      /* Otherwise verify if subtask i will still meet its deadline if the requesting
      subtask is allowed to continue for X time units */
      if (T* + S_i + X) ≤ D_i
            /* Yes */ T* = T* + S_i, Increment i, goto next subtask
      /* Otherwise, see if subtask i is eligible for preemptive abortion */
      else
            if (T - A_i) ≥ δ_i
                Ψ = Ψ U i /* subtask i is eligible */
            else set θ /* subtask i is not eligible */
      /* Adjust the timeline to repeat the above process for next subtask */
      /* See dual-timeline illustration */
      T* = T* + S_i
      Increment i
    End of loop [1]

  If cardinality of Ψ = 0 and θ is not set
/* no subtask needs to be aborted */
  return (X)

   If cardinality of Ψ ≥ 1,
/* then we have candidates who are eligible for abortion */
      Call AbortBasedOnLossCnst()
/* which picks a candidate from Ψ a subtask with the largest processing
time for preemptive abortion */

   If cardinality of Ψ = 0 and θ is set,
/* we do not have eligible candidates for abortion. So, now we pick
candidates based on user-defined priorities P_i's */
      Call AbortBasedOnPriority()
/* which picks a lowest priority (assigned by Call Admission Control)
subtask for abortion and returns its processing time as the amount of
time salvagable */
```

Figure 4: PACS Algorithm

T	The current system time
X	Extra amount of service needed by the requesting subtask
S_i	The estimated service time of subtask i
D_i	The assigned deadline of subtask i
A_i	Time of last abortion (preemptive or otherwise) of a subtask belonging to application i
δ_i	The nodal loss constraint of an application class i
P_i	The priority of class i
Ψ	Set of possible candidates for abortion
θ	A Flag, when set, implies preemptive abortion should take place
AbortBasedOnLossCnst()	Aborts a subtask based on loss constraint
AbortBasedOnPriority()	Aborts a subtask based on the CAC-assigned priority

Table 1: PACS Algorithm Parameters

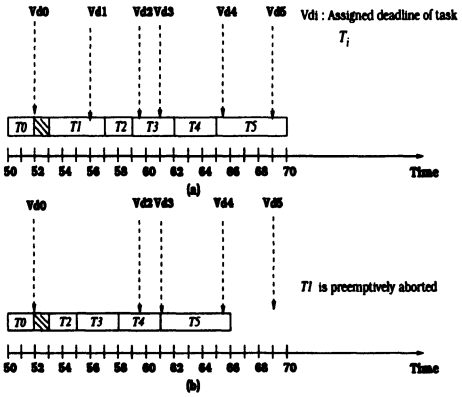


Figure 5: An illustration of the algorithm

salvaged time is returned to the scheduler, which allows the subtask to continue its service. Since the analysis is based on estimated execution times, it is possible that the requesting subtask may again undergo deadline expiration, in which case, the same process will be repeated. It is also possible that some other subtask which was involved in the analysis may itself undergo deadline violation for the same reason.

3.3 Illustration of the PACS algorithm

Example 2. Consider the subtasks shown in Figure 5. The subtask-specific information for each subtask is shown in Table 2. Note that such information is usually maintained as a part of the Process Control Block or Task Control Block data structure.

The subtask T_0 has undergone deadline expiration after executing for 2 units of time at time 52, and is requesting an additional processing time of 1 time unit (shown as a shaded box in Figure 5). At this time, the PACS algorithm is invoked to see if the request for additional time for subtask T_0 can be honored. The PACS

	Estimated Service Time	Assigned Deadline	Loss Constraint	Previous Abortion Instance	CAC-Priority
T_0	2.0	52.0	25.0	30.0	2
T_1	4.0	56.0	30.0	15.0	3
T_2	2.0	59.5	30.0	5.0	3
T_3	3.0	61.0	30.0	0.0	3
T_4	3.0	65.5	20.0	20.0	1
T_5	5.0	69.0	25.0	35.0	2

Table 2: Subtask Parameters

algorithm lays down the subtask profiles as shown in Figure 5(a), where the subtasks in the ready queue are scheduled from a new starting point determined by the amount of additonal request for subtask T_0. The subtasks T_1, T_3 and T_5 miss their assigned deadlines as a result of extending service to T_0. However, while checking the individual subtasks for their eligibility for preemptive abortion based on their nodal loss constraints and previous abortion instances, PACS finds $\Psi = \{T_1, T_3\}$. See Table 2. The algorithm then picks subtask T_1 for abortion, since its processing requirement is greater of the two subtasks in Ψ. Since the amount of time salvaged is greater than the amount requested, (4 time units vs 1 time unit), T_0 can continue its execution. The redrawn profiles are shown in Figure 5(b).
For more detailed illustration of the algorithm, please refer to [7].

4 Performance Evaluation

The system shown in Figure 1 was simulated. For each frame, there is an end-to-end deadline and each corresponding application specifies a loss constraint. The loss constraint refers to the loss constraint for B and P frames, I frames were not considered for preemptive abortion. The audio was assumed to have a separate channel and was not considered for preemptive abortion. So, the following discussion refers to video only. Also, we assume that the frame processing times at each node are estimated using exponential distribution[1].

The frames from individual applications arrive at the multiplexer periodically, e.g., 30 frames every second. Even though the encoder outputs 30 compressed frames in a second, the output stream for each application from the encoder is *not* periodic due to the variance in compression times for individual frames. This results in occasional bursty arrivals at the application and the network schedulers. Assuming that the encoder outputs 30 frames in a second for each application, both the application and the network scheduler have to process 30 frames in a second per application. We achieve this by invoking the respective scheduler once every 33.33 ms, which then runs all the subtasks in its queue. The queue itself is ordered according to the non-preemptive Earliest Deadline First discipline, i.e. the subtask with the most urgent deadline runs first during each invocation.

The performance measure of interest here is the *cutoff* load where the NLR

[1] Ofcourse the algorithm is not limited to any particular service time distribution.

crosses the ANLR for each application class. See Figure 2. We do this with, and without, PACS. For both the cases, however, the nodal allocation of deadlines and loss constraints is performed according to the description in Section 3.1. The metric *Relative Gain* (RG) is defined as follows.

$$RG = \frac{cutoff_{PACS+} - cutoff_{PACS-}}{cutoff_{OPT} - cutoff_{PACS-}} \quad (2)$$

where $cutoff_{PACS+}$ and $cutoff_{PACS-}$ denote the cutoffs achieved with and without the PACS algorithm respectively. $Cutoff_{OPT}$ is the upper bound on cutoff. Since $cutoff_{OPT} \leq 1$, a lower bound of the relative gain is given by

$$RG_L = \frac{cutoff_{PACS+} - cutoff_{PACS-}}{1 - cutoff_{PACS-}} \quad (3)$$

A final note before we present the numerical results: we are not aware of similar work to compare our results with.

4.1 Numerical Results

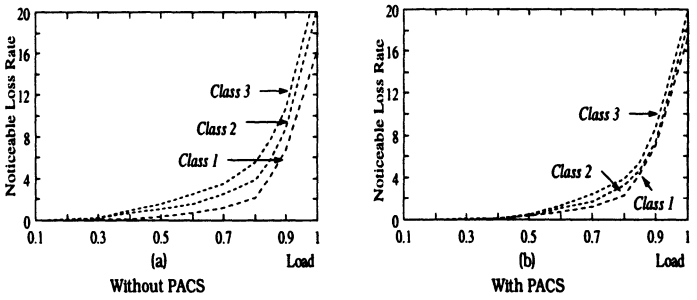
We now focus on our first set of experiments. There are three classes of applications, each with a member application. The parameters values, the performance graphs, as well as the improvements made possible by the PACS algorithm for this set are presented in Figure 6 [2]. In Figure 6(a), the NLR is plotted versus the system load. It can be seen from this graph that Class 3 *notices* more losses than the other two due to its stricter loss constraint (even though at steady state all the three classes undergo approximately the same percentage of deadline misses). It can also be seen that the individual curves are spaced apart.

Figure 6(b) shows the performance of the three classes with the PACS scheme. The following observations can be made from this figure.

First, there is a considerable reduction in NLR for Class 2 and Class 3 applications. The overall improvement comes from two sources. First, where there is no abortion involved, simply additional processing time is borrowed from other subtasks without violating their assigned deadlines. This phenomenon alone appears to be the case until the load becomes moderately heavy (until about 0.6), and it continues along with preemptive abortion at loads beyond 0.6. Secondly, as the load increases (beyond 0.6), PACS aborts eligible candidates to make room for needy subtasks. Class 3 subtasks gain the most by aborting both Class 2 and Class 1 subtasks, since their loss constraints are less stringent. Class 2 subtasks compete with Class 3 subtasks to benefit from aborting Class 1 subtasks; they rarely get a chance to abort Class 3 subtasks. This explains why the improvement is higher for Class 3 subtasks compared to that for Class 2. There is a slight increase in NLR for Class 1 (represented by a negative RG_L of 2.3%) since a small percentage of its subtasks are aborted based on their priority. These priority-based abortions were counted as noticeable losses. However, recall that abortions that occur when the loss constraint allows PACS to do so are *not* noticeable losses.

The reason why preemptive abortion adds to considerable improvement in NLR (along with cycle stealing only) can be attributed as follows. PACS is a greedy algorithm which chooses an eligible subtask with *maximum* processing requirement when it has to perform abortion (based either on priority or loss constraint). And, it does this often during periods of burstiness or overloads during which normally many subtasks are lost resulting in

[2] A recent document by the Multimedia Communications Forum recommends using a *terminal equipment* (which includes video servers) delay of 200 ms and 150 ms for enhanced and premium quality multimedia communication respectively [11].



	Class 1	Class 2	Class 3
Arrival Rate in frames/sec	30	30	30
End-to-end Deadline	200 ms	200 ms	200 ms
Deadline Factor d_f	1	1	1
Overall Loss Constraint	600 ms	1000 ms	1500 ms
Acceptable NLR	5.5%	3.3%	2.2%
$Cutoff_{pacs-}$	0.875	0.755	0.570
$Cutoff_{pacs+}$	0.872	0.800	0.675
Relative Gain RG_L	-2.30%	18.40%	24.41%

Figure 6: Simulation Results: Case 1

higher NLR. By greedily salvaging time during such periods, PACS dampens the effect of burstiness, thus reducing clustered losses which would result in poor QoS.

Our next set of experiments include tighter end-to-end deadline and nodal deadline for all the three classes, as well as the addition of two more classes of applications. The details of these experiments and the performance of the PACS algorithm are available in [7].

5 Conclusion

In this paper, we have presented a resource management strategy for supporting multiple application classes with disparate QoS requirements. We have evaluated the performance of this scheme using a metric that suitably captures the loss tolerance abilities of soft real-time multimedia applications. Finally, we have shown through simulation experiments that the PACS approach indeed provides meaningful performance improvements in a practical application.

Our future work involves applying our loss model and PACS to bundled multimedia such as audio, video and graphics. We are implementing the PACS algorithm in a stored video server (for distance learning) for supporting rate and loss guarantees to individual user connections.

Acknowledgement

This work was supported in part by the National Science Foundation under grant CCR-911 9922

References

[1] R. L. Cruz. "A Calculus for Network Delay, Part II: Network Analysis" *IEEE Transactions on Information Theory*, Vol 37, No. 1, Jan 1991.

[2] C. Douglass Locke. "Software Architecture for Hard Real-Time Applications: Cyclic Executives vs Fixed Priority Executives" *The Journal of Real-Time Systems*, vol4(1), pp 37-53, March 1992.

[3] D. Ferrari and D. Verma. "A Scheme for Real-time Channel Establishment in Wide-Area Networks". *IEEE Journal on Selected Areas in Communication*. Vol. 8, No. 3, pp 368-379, April 1990.

[4] R. D. Gaglianello and G. L. Cash. "Montage: Continuous Presence Teleconferencing Utilizing Compressed Domain Video Bridging". *Multimedia Communications Research lab report*, AT&T Bell Laboratories, Holmdel, NJ 07733. 1994-95.

[5] B. Kao and H. Garcia-Molina, "Deadline Assignment in a Distributed Soft Real-Time System" *Proceedings of the 13th International Conference on Distributed Computing Systems*, pp 428-437, 1993.

[6] R. Koodli and C. M. Krishna. "Supporting End-to-End Deadlines for Soft Real-time Multimedia Applications". In *IEEE International Performance, Computing and Communications Conference*, February 5-7, 1997.

[7] R. Koodli and C. M. Krishna "Supporting Multiple-tier QoS in a Video Bridging Application" TR-CSE-97-2, ECE department, University of Massachusetts, Amherst, MA 01003. ftp://aurelius.ecs.umass.edu/pub/koodli/qos.ps.Z

[8] J. Kurose. "On Computing Per-Session Performance Bounds in High-Speed Multi-Hop Computer Networks", In *ACM SIGMETRICS'92*, pp 128-139, 1992.

[9] R. Nagarajan, J. Kurose and D. Towsley. "Nodal Allocation of End-to-End QoS in High-Speed Networks." *IFIP*, January 1993.

[10] C. L. Liu and J. W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment" *Journal of the Association for Computing Machinery*, vol 20(1), pp 46-61, 1973.

[11] "Multimedia Communications Quality of Service, Part 2: Multimedia Desktop Collaboration Requirements". Final document, Multimedia Communications Forum Inc. Suite 201-931 Brunette Avenue Coquitlam, British Columbia, Canada V3K 6T5.

[12] "REACT Real-time Programmers Guide" *Silicon Graphics Technical Publications*, Document number 007-2499-002.

[13] H. Schulzrinne. "Reducing and Charcterizing Packet Loss for High-Speed Computer Networks with Real-Time Services" *PhD dissertation, University of Massachusetts, Amherst, Computer Science Technical Report 93-54*, 1993.

[14] A. Yan, A. Ganz, and C. M. Krishna, "A Distributed Adaptive Protocol Providing Real-Time Services on WDM Based LANs", *IEEE Journal of Lightwave Technology* Vol 45, pp 753-756, June 1996.

Biography

Rajeev Koodli is a PhD candidate in the ECE department at the University of Massachusetts, Amherst. His research interests include scheduling support for real-time multimedia, QoS in video applications and distributed systems.

C. M. Krishna is a professor in the ECE department at the University of Massachusetts, Amherst. His research interests are real-time systems, fault-tolerance and real-time recovery, distributed systems and high-speed networks.