Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports | Department of Computer Science

1996

On Tailoring Thread Schedules in Protocol Design: Experimental Results

Juan Carlos Gomez

Vernon J. Rego
Purdue University, rego@cs.purdue.edu

V. S. Sunderam

Report Number:

96-018

Gomez, Juan Carlos; Rego, Vernon J.; and Sunderam, V. S., "On Tailoring Thread Schedules in Protocol Design: Experimental Results" (1996). *Department of Computer Science Technical Reports.* Paper 1274.
https://docs.lib.purdue.edu/cstech/1274

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

ON TAILORING THREAD SCHEDULES IN
PROTOCOL DESIGN: EXPERIMENTAL RESULTS

Juan Carlos Gomez
Vernon Rego
V. S. Sunderam

CSD TR-96-018
February 1996

On Tailoring Thread Schedules in Protocol Design: Experimental Results

Juan Carlos Gomez
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
USA

Vernon Rego*
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
USA

V. S. Sunderam†
Department of Mathematics & Computer Science
Emory University
Atlanta, GA 30322
USA

Abstract

A number of factors motivate and favor the implementation of communication protocols in user-space. This motivation is particularly strong in the context of providing multiway connectionless transport for distributed computing, multimedia, and conferencing applications. Here, protocol implementations can hold the key to optimal functionality and performance. As part of a larger study investigating experimental strategies for general multiway, connectionless transport for a variety of applications and multiple media, we report our experiences with the implementation of an unreliable transport mechanism. This step is a precursor to multiway reliable transport. We focus on the use of user-space threads and UDP-based protocol actions, presenting strategies for scheduling threads that either do a client's application-processing or retrieve packets from a client's network receive buffer. Efficient packet retrieval is crucial especially when transport is reliable, because higher packet-loss implies longer network delays. With experiments performed on a Sun SPARC 5 LAN environment, we show how different threads scheduling strategies yield different levels of application-processing efficiency and packet-loss.

* Supported in part by NSF-CCR 9311862, ONR-9310233 and ARO-93G0045.
† Supported in part by NSF-ASC-9527186 and ONR-9310278

1 Introduction

We are witness to a paradigm shift in networking and distributed computing. Within a span of about two decades, isolated computing machines have been transformed into cooperating systems whose processes interact closely with one another via networks and efficient protocols. While reliable protocols have revolutionized computing by making heterogeneous networks transparent, at least in part, to users and applications, they have also revealed to us ways of using networks for supporting virtual collaboratories and enterprises: computing engines once transformed into cooperating engines are now undergoing a further transformation into application-based collaborating engines. But with this transformation comes a number of requirements and problems which require novel solutions.

It is inevitable that new generation netware must support a variety of applications. These include combinations of high-performance numerical computations, audio, video and high-resolution imaging. Ideally, these will operate as integrated services, and may be packaged as multimedia applications, i.e., applications with multiple media involving realtime delivery and interactivity. Videoconferencing, distance-learning, simulation-based collaborative design, reliable distributed interactive simulations, etc. are some examples. Realtime and interactive bandwidth-creep applications, in which mixed media force bandwidth requirements upwards, will be common. As an example of bandwidth-creep, small technical documents typically contain tens of kB of text, but with an image added to the document, size may climb to several MB. Robust protocols we have come to rely on, such as TCP/IP or extant reliable UDP schemes, were simply not designed for clients with such volatile demands, especially when the transfer involves multiple participants (i.e. many-to-many communication).

Different kinds of applications, e.g., realtime, bandwidth-creep, high-performance numerical computations, and collaborative applications, place very different demands on a network. Because of this, algorithms for maximizing throughput and minimizing latency between or within applications are hard to come by. Indeed, a protocol like TCP/IP operates under certain guidelines in a client independent way; often, new protocols are developed to provide specific services (e.g., RMP [1] and AFDP [2]) not provided by existing ones. Even with high speed nets like the 100-Mbps Ethernet, FDDI, or ATM, existing protocols will be hard pressed to deliver integrated services such as scalability, low-latency, isochronous transport, bandwidth reservation and prioritized schemes for smooth delivery. These services are crucial for the high-performance, realtime internetworking applications of the future.

Our interest lies in improving protocol performance in distributed computing environments which may eventually support multicasting and collaborative applications. In this study, we focus on aspects of protocol design and implementation that can reduce latency through reduced packet-loss, while simultaneously enhancing client-related processing. For example, in extant message-based, distributed computing systems, processes alternate between computation and communication through a single thread of control. Processes interact with one another, by sending and receiving messages, at well-defined points in their execution sequences. Examples of such systems include PVM [3], P4 [4], MPI [5, 6] and Conch [7], all of which seek to support high-performance computations

in distributed environments. PVM utilizes communication daemons which talk to one another using UDP. In P4, most MPI implementations, Conch, and PVM (route direct option), compute processes may interact using TCP/IP. Conch uses software routing between processes to support different network topologies, providing for improved scalability.

In Unix environments, for example, alternating between communication and computation within a single thread of control can result in performance loss. When a process expects to receive a message, it must either block and wait for the message, or it can proceed with a computation and periodically check (poll) for the message. A blocking wait prevents the process from doing other useful work, and further aggravates the situation by generating an OS context switch; control is given to the blocked process at some later time, after a message arrives. This occurs, for instance, with PVM tasks and P4 processes. With polling, as done in the Conch system, useful work proceeds but only at the expense of increased overheads and delayed message receipt. The delay depends on the time spent computing between consecutive polls, and the overhead depends on the manner in which polling is implemented. For example, polling in Conch is implemented through signal-based multiplexing. Because of the 500 ms granularity of the timer, Conch examines its receive buffers for messages only every half a second.

Regardless of the particular transport mechanism used, delay in acknowledging message receipt contributes to delay experienced by future messages. This may also be indicative of a potential backup in message delivery between a receiving client and its host's network layer, further delaying client processing and response. Naturally, this may further degrade protocol services. To be specific, if messages occupying receive buffers are not retrieved and rapidly acknowledged by the receiver, the sender is forced to delay sending further messages to this receiver[1]. Even worse, delay in acknowledgement forces a sender to retransmit messages because the sender must assume that the information was lost in transit.

2 Motivation and approach

The advantages of user-space protocols are well-known [8, 9, 10], though implementations are few. Protocol layers, provided mainly for modularity, make protocol actions less sensitive to client needs [11]. Layer boundaries tend to restrict interfaces, and lower layers are unable to query clients so to make the best use of asynchronous actions, timing and buffering. Our intent is to remedy this situation by implementing threads-based protocol actions in user-space for environments supporting high-performance distributed computation and collaboration.

In utilizing user-level threads, the idea is to perform efficient scheduling of compute- and communicate- functions within a single process, in an attempt to minimize the ill effects of OS context switches and message retransmission described earlier. To integrate compute- and communicate- functions within a single process, we resort to user-space multithreading. The resulting protocol actions will be efficient only if thread operations

[1] In PVM's daemon-to-daemon communication there is no flow control, and the sender does not delay sending further messages. In extreme cases, the consequences are disastrous.

are efficient, and threads are scheduled in a manner that minimizes packet delays. Tailored thread schedules will help reduce packet-loss (and thus retransmission delays) at a receiver's network layer, simultaneously increasing client processing. We believe this approach to be important because these factors are critical in real-time message delivery and efficient multicast.

To aid in experimentation with scheduling and to investigate issues in multiway, connectionless transport, we have embarked on the design and implementation of a connectionless, lightweight and multiway communications environment (i.e., CLAM). At this time the CLAM protocol suite supports point-to-point connectionless transport, both unreliably as well as reliably, though the discussion in this paper is restricted to unreliable communication. In the future, we expect to support multiway connectionless transport with facilities for message delivery at various, selectable qualities of service, including bounded loss, lossless, and sequenced transmission; both unordered with respect to the recipients and with atomic multicast semantics. CLAM is an active-message based system implemented with support from the Ariadne [12] threads system. In the interests of portability, CLAM utilizes standard threads primitives available in most threads libraries.

The CLAM system presently consists of two layers. A lower layer provides UDP-based unreliable messaging services. Within this layer, a simple but efficient scheduler dispatches computation and communication functions in a manner that attempts to minimize packet-loss and maximize CPU attention to an application's computation. Supported by the lower layer, the upper layer provides for reliable and realtime message services and an interface to core MPI [5] communication primitives. As an example of immediate applicability, CLAM's unreliable layer (and proposed multicast extension) is ideal for support of the DIS protocol [13], where it is critical that packet-loss be small, and work efficiency high.

The remainder of the paper is devoted to a discussion of the scheduling algorithms used in CLAM's lower layer, for uniprocessor hosts. Further improvements can be shown for shared memory multiprocessors. This requires additional support from the threads system, and is the subject of another paper. In Section 3, we present five different scheduling algorithms, each using threads and time-slicing to multiplex between communication and computation. Experimental results on performance are presented alongside a description of each scheduling algorithm. We conclude briefly in Section 4, and mention future work.

3 On scheduling computation and communication

A process may be made to alternate between compute work and communication work, given appropriate help from a signalling mechanism. Though the use of signals is risky and requires care, the mechanism is effective in separating send and receive task functionality, and is ideal in adapting tasks to load. When used in concert with threads, careful signal-based multiplexing offers potential gains that may significantly outweigh risks. In this study, we envision an application as consisting of multiple threads of control, and focus on two immediate goals. One is the minimization of packet-loss at the network layer, effected by a *receive* thread which is responsible for the rapid retrieval of

packets from a given *receive buffer* (e.g., if UDP is used, this buffer is the UDP buffer). The other is the maximization of work done by *work* threads, which are responsible for compute-bound application processing.

With separate threads for sending and receiving, each of which executes for a given amount of time under signal-based control, a messaging subsystem may readily adapt to traffic. A receive thread may monitor and process incoming packets, to curtail packet-loss at the receive buffer. A send thread may dispatch outgoing packets in a manner that is sensitive to client needs, minimizing time-to-acknowledgement for packets received. Implementing such adaptation, however, requires that the system make low-cost scheduling decisions using transient information. In particular, when should the system schedule a client's *receive*, *send* and *work* threads, and what specific information should be used in scheduling these threads ? A great variety of scheduling options exist, all operating in the millisecond time scale, with an equally great variety of efficiency levels. Identifying the best or even selecting a reasonably good algorithm is a nontrivial task.

We propose, implement and measure the performance of five simple but effective strategies for scheduling *receive* and *work* threads. Each strategy relies on signal-based time slicing between threads, with selective nonpreemption and possible use of I/O interrupts. To enable us to focus our attention on performance at the receiving end of a communicating pair, all algorithms and measurements deal with the *receive* thread and minimization of packet loss, though increase in *work* thread efficiency is an indirect result of increase in *receive* thread efficiency. Since this study is limited to unreliable transmission, and the experiments do not involve particular applications, we do not address the scheduling of *send* threads. Experiments on the scheduling of *send* threads are to be found in a companion study geared towards efficient reliable transmission.

The Standard Model

Consider the standard model illustrated in Figure 1, where machine X hosts two processes (called A and B) and machine Y hosts three processes (called A, B, and C). On opening a socket, each process is given a receive buffer (shown as a network receive buffer). The OS kernel routes a process's incoming packets from its host's IP buffer to the process's receive buffer. Suppose that processes on machines X and Y seek to interact with one another over the network. Labels `W`, `S` and `R` within a process represent functionality for client-related *work*, message *sending* and message *receipt*, respectively. When given control by the Operating System (OS), a process may attend to any one of these functions at any time. But once some functionality is invoked, it must be completed before some other functionality is invoked. If process B on machine X sends a string of packets to process A on machine Y, A would do well to retrieve the packets from its receive buffer as rapidly as possible. Of course, if the CPU is with another process when the packets arrive, process A has little recourse but to wait. But if A is using the CPU in `W` or in `S` when packets arrive, its best chance for not seeing packets drop off the end of the receive buffer is to process incoming packets immediately. Further, to enable the client to take action on the data without undue delay, delivery to the client should be equally rapid. Ideally, delay in detecting a packet in a host's receive buffer and making this data available to a client should be negligible.

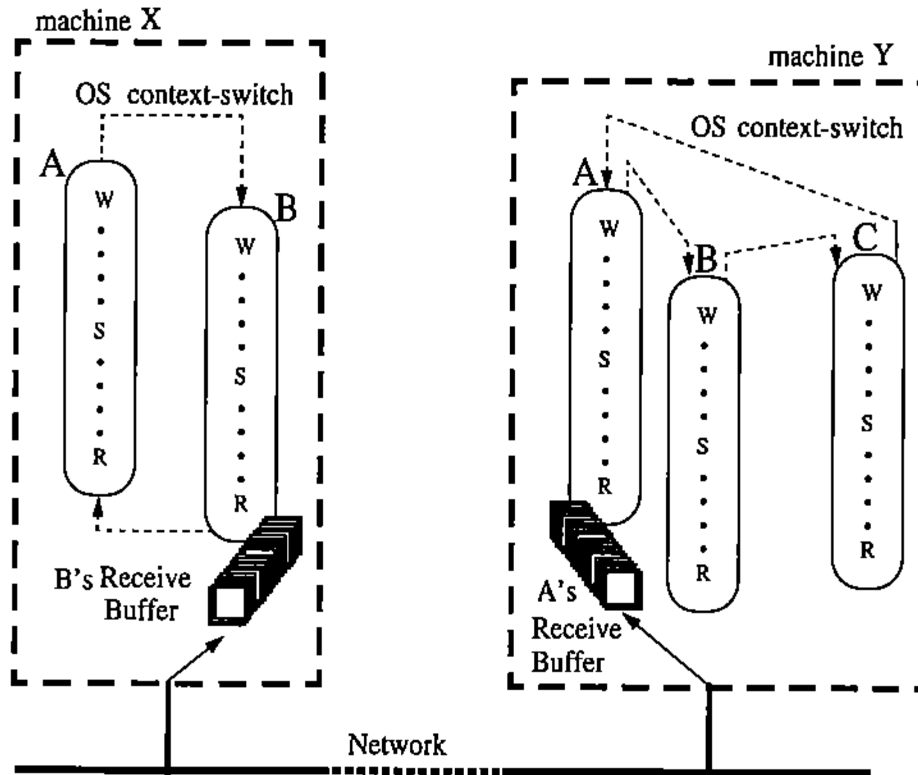

Figure 1: The Standard Model

In extant message-passing systems (e.g., PVM, P4/Parmacs) it is highly unusual for clients to implement *work*, message *sending* and message *receiving* functionality as distinct threads. The client computes until it is necessary to send or to receive. Functions for sending and/or receiving are invoked as and when necessary. If either *sends* or *receives* involve blocking calls to the OS, an OS context switch ensues. Once switched, a process regains control at some later time, but only after the condition for the block has been satisfied. Though it may appear pointless for a CPU to stay with a process once a blocking call is made, the forced context switch actually penalizes the process. It prevents such a process from continuing with other useful work that is unrelated to the block, were the process to have access to such work.

Though PVM exploits multiple processes (i.e., a daemon and a compute process) to overlap communication and computation, one or more OS context switches take place while daemons talk to local compute processes (tasks). This hampers the ability of both a daemon and a compute process to attend to their respective receive buffers. Moreover, communication between compute processes is hindered by the presence of intermediate daemons. Finally, since such systems are generally TCP/IP based, integrating services such as scalability and realtime delivery is difficult. P4 suffers from similar drawbacks and for similar reasons, though daemons are not used in communication.

Experimental Methodology

All experiments were conducted with a SPARCstation 20 sending to a SPARCstation 5 over an Ethernet, both machines running SunOS 5.4. The protocol *receive* actions and application-related *work* actions were implemented as threads, using the Ariadne threads library [12]. Preemption was accomplished with the aid of time-slicing support available in Ariadne. CLAM's unreliable transmission layer is implemented in about 2000 lines of C-code; this excludes threads library code. We used UDP as the underlying protocol. Datagram size was such that Ethernet frames were of maximum size (Ethernet MTU). Both CPUs were only lightly loaded by other applications, and interference from other network traffic was negligible during experimentation.

Packet interdeparture times at the sender were kept constant, ranging from 10 to 250 ms. At the receiving end, however, interarrival times were not strictly constant, due to network jitter. We also studied the effect of correlated arrivals, interspersing periods of heavy traffic among periods of light traffic. Here, arrivals were governed by a two-state Markov chain with states **H** (high rate) and **L** (low rate) controlling packet interarrival times. In the scheduling algorithms, time slice length was used as the primary control parameter, since it is this quantity which determines for just how long a given thread will execute before relinquishing CPU control. Time slice length was permitted to vary between 10 ms and 500 ms. The granularity of the measuring clock was 10 ms. Other control parameters included a simple history of the number of packets received per instantiation of the *receive* thread, expansion and contraction factors for time slice length, and the enabling of interrupts.

A measure of the amount of work done per second is obtained by sampling the number of *CopyBuffer* operations (see Figure 4(a)) executed by the *work* thread `W` during a period of roughly 10 secs of real time. Each such sample is normalized, and an average obtained over 20 such samples. The normalization is a correction done because of timing overruns in measuring copy operations of the *work* thread. First, the number of *CopyBuffer* operations done by the *work* thread in 10 secs of real time is not an integer. Second, this number is random, because it depends on whether one or more OS context switches occur during execution of the *work* thread and the length of time the thread is displaced from the CPU on each switch. The correction enables us to report the actual work done by the *work* thread during its execution. Each point displayed on a graph is a further average of up to 18 batched means, with an approximate 90% confidence interval based on the Student t-distribution.

Packet loss at a receive buffer is measured by tagging each packet with a unique sequence number and identifying gaps in the sequence of packets arriving at the receiver. As with work measurement, this is also done over a period of roughly 10 secs in real time. Note that this scheme is workable in an Ethernet LAN where in-order delivery is guaranteed. In networks where in-order deliver is not guaranteed a more sophisticated accounting scheme is required. As in the measure of work, each point displayed on a graph is the average of up to 18 batched means. Each batch is made up of 20 normalized samples, with each sample measured over a 10 second interval, yielding an approximate 90% confidence interval as before. It took us about 10 minutes to obtain each data point in each of the graphs displayed in this section. The sampling and measurement work for

a single graph required roughly 3 to 4 hours.

3.1 Slicing with Preemptive Receive (SPR)

Consider a multithreaded version of the standard model, as depicted in Figure 2. Because our emphasis is on efficient scheduling of a *receive* thread, to minimize incoming packet loss, the *send* and *work* threads are assumed to be lumped together in `W`. We defer discussion of *send* thread schedules to a companion paper on reliable transmission. As mentioned earlier, we restrict our attention to unreliable transmission, with a single *receive* thread monitoring packet arrivals at a host's receive buffer.

An OS context switch may occur during execution of any thread, placing CPU attention on other processes in the system. When control is returned to the process in question, execution resumes within the thread preempted by the OS. In general, actions taken by the OS are outside our (user) control. What is in our control is deciding just *how* our process's CPU time slice is to be divided between its *receive* thread and other threads, given that the goal is to keep packet-loss low and work efficiency high.

Threads enable distinct functionalities to operate within a single address space, with low thread context switch overheads, compared to the relatively high OS context switch overheads. Given some level of multithreading support within a process, a reasonable scheduling scheme entails sharing CPU attention between *send*, *receive* and *work* threads in some equitable manner. The easiest way to do this is to give each thread a fixed quantum of CPU attention, using a *fixed* time slice. A running thread is preempted by the threads system scheduler when a signal is generated on time slice expiry. Such a fixed-slice, preemptive approach is taken by the Conch message-passing library [7], based on a simple signal-generated multiplexing scheme.

When the threads system scheduler selects a ready thread to run, it has no way of knowing if the thread will have work to do. This may depend on the thread's data. Thus, such a thread is forced to run even though it may have no work to do. If it finds no work when it begins to run, it will typically yield control to the threads system scheduler. If this action is frequent, many unnecessary thread context switches are generated. With too small a time slice, a frequently scheduled *receive* thread will tend to find its receive buffer empty. Even if it finds its buffer occupied every so often, a small time slice will cause the *receive* thread to be preempted even while it has packets to remove from the buffer. Both factors – frequent context switching and small *receive* thread run times – contribute to protocol performance degradation.

Performance of SPR scheduling

How well the simple SPR scheme performs is a function of the size of the time slice and rate of packet arrival. With small interarrival times, small time slices enable the system to clean out incoming-buffers more rapidly than larger time slices. Similarly, when interarrival times are large, larger time slices are more effective because of the corresponding reduction in thread context switches. For a given arrival rate, an appropriately chosen time slice will help minimize context switching overhead and keep the probability of packet-loss at the network layer low.

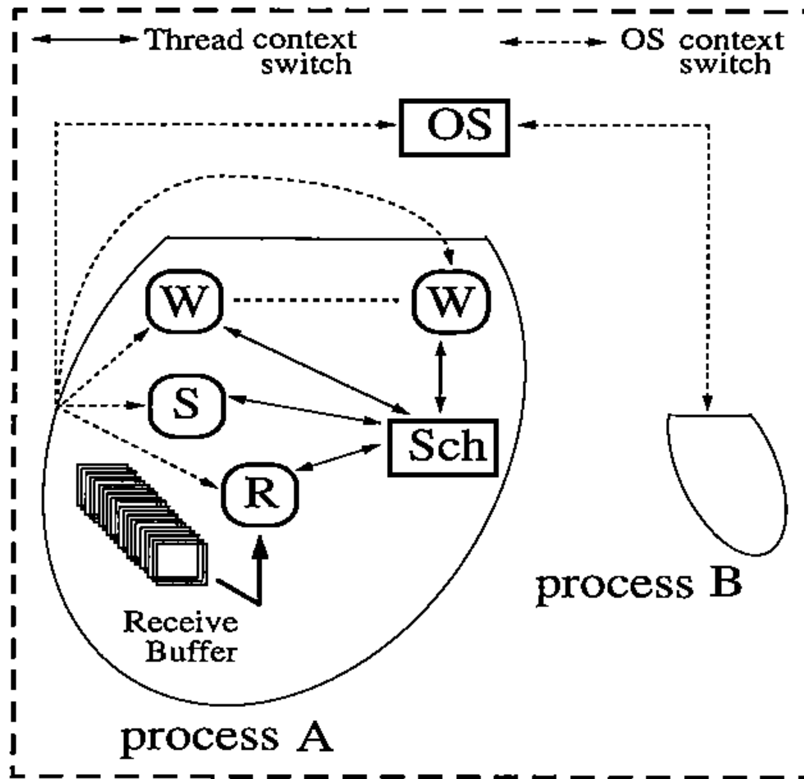

Figure 2: The Multithreaded Model

In Figure 3(a) is shown the average amount of work done per second by the work thread `W` graphed against the packet interarrival time. How work done per unit time and packets lost per unit time are measured is explained in the subsection on experimental methodology. Also, as explained earlier, each data point displayed lies within a 90% confidence interval. The end-points of the confidence intervals are not displayed because standard deviations are negligible, relative to graph dimensions. Pseudo-code for *work, send, and receive* threads is shown in Figures 4(a), 4(b), and 4(c), respectively. In Figure 3(b) is shown the measured average packet loss per second, graphed against packet interarrival time.

The experimental results suggest that increasing the size of the time slice yields a corresponding increase in the amount of work that can be done by the application. Unfortunately, increasing the size of the time slice also causes the minimum interarrival time for which packet loss is significant to increase (see Figure 3(b)). This means that, if time slice size is fixed, regions of maximum work efficiency and minimal packet loss depend on packet arrival rates.

3.2 Slicing with Nonpreemptive Receive (SNR)

With the simple SPR scheme just presented, a *receive* thread is preempted when its time slice expires, even though the receive buffer may not be empty. If packets continue to arrive, packet loss is inevitable. While one solution is to allow a *receive* thread to continue to execute as long as there are packets to be read, this can cause other threads to starve

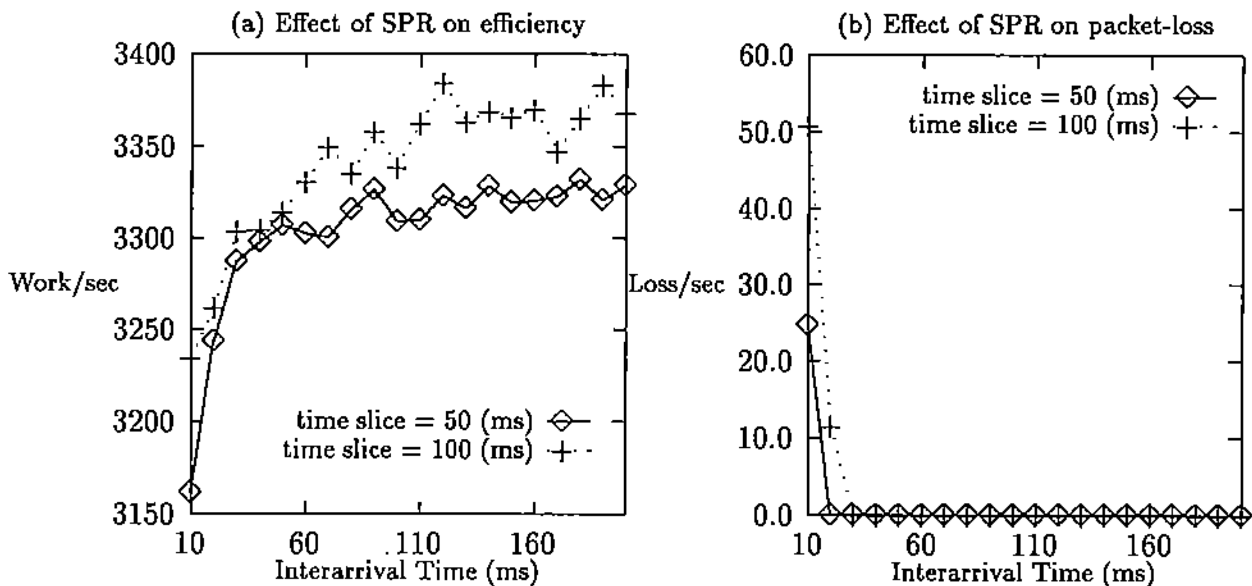

Figure 3: Performance of SPR strategy

for CPU attention. In instances of reliable transmission, if a slow *receive* thread is overworked by a fast source, it is conceivable that its *send* thread counterpart (executing on the same machine) may not be able to return acknowledgements rapidly enough. This will occur in the absence of good flow control, and may lead a source to conclude that the link is down. Also, processing speed mismatches may cause a deadlock in upper layers of the software. With nonpreemptive execution, a *receive* thread can attend to its receive-buffer more aggressively, though it does so only at the expense of other threads.

Performance of SNR scheduling

In Figure 5(a) is shown the average amount of work done per second by the work thread W, graphed against the packet interarrival time, for the nonpreemptive *receive* (SNR) strategy. In Figure 5(b) is shown the corresponding average packet loss per second, also graphed against the packet interarrival time. Though use of a nonpreemptive-receive may appear to offer potential for a considerably smaller packet-loss than use of a preemptive-receive, simply because more attention is paid to emptying a receive buffer, the experiments show no significant difference.

A compromise between preemption and nonpreemption is the use of a limit on the number of packets read by the *receive* thread before it relinquishes CPU control. Besides keeping packet-loss low, this approach has some promise of preventing CPU starvation at *send* and *work* threads. During experimentation, we found this tack to show a slight decrease in efficiency. We attribute this to the overhead of disabling Ariadne's time slice feature so that a *receive* thread may attend to a receive buffer without being preempted.

(a)

```
WorkThread()
{
    while( TRUE ){
        CopyBuffer();
        UpdateStats( Work );
        UpdateStats( PacketLoss );
    }
}
```

(b)

```
SendThread()
{
    while( TRUE ){
        message = MessageDequeue();
        SendMessage( socket, message );
        FreeMessage( message );
    }
}
```

(c)

```
ReceiveThread()
{
    message = AllocateMessage();
    while( ReceiveMessage( message ) = WOULDBLOCK ){
        AdaptTimeSlice();
        yield();
    }
    UpdatePacketLoss( message );
    FreeMessage( message );
}
```

Figure 4: Pseudo-code for Threads

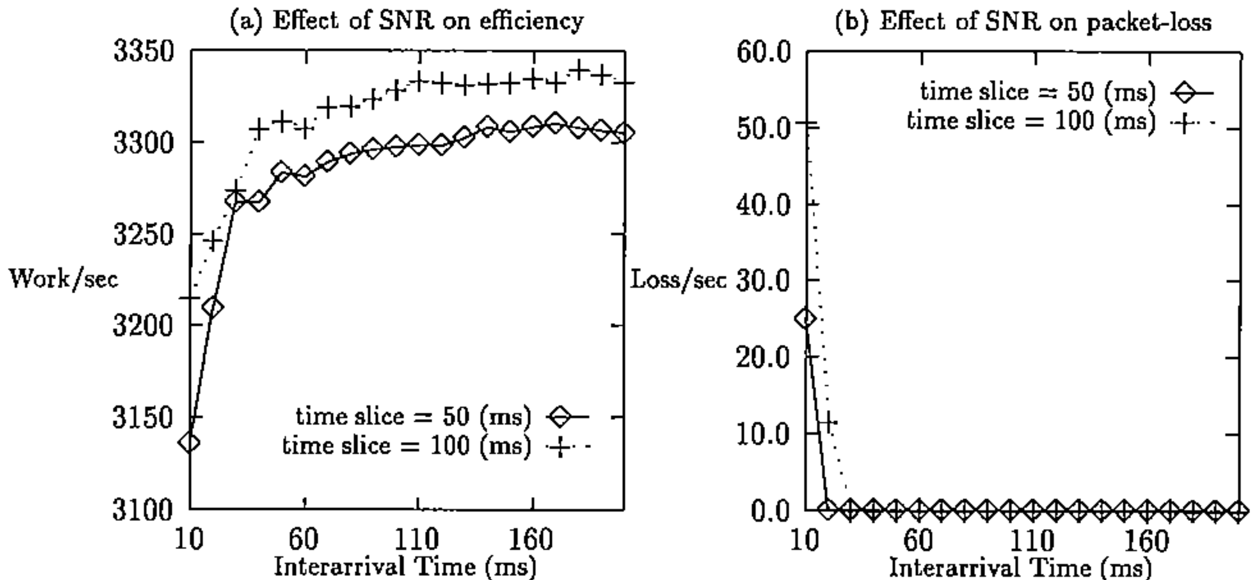

Figure 5: Performance of SNR strategy

3.3 Adaptive time slice mechanisms

For a fixed packet arrival rate λ, where $\lambda = 1/t_a$ and t_a is the average interarrival time, an optimal value of time slice length T is one which minimizes the packet loss l_p and maximizes work thread efficiency. Because λ is application and network dependent, it will typically vary with time. Given a varying λ, the idea is to adjust T dynamically with λ so as to keep l_p small. Since the average cost of a thread context switch in Ariadne is small (e.g., 10–15 μ-secs for a Sun SPARC 5) a near-optimal procedure for repeatedly adjusting the time slice would be possible only if the CPU is dedicated to the process. Unfortunately, as depicted in Figure 2, CPU attention is typically shared between multiple processes. An OS context switch may occur, unpredictably, during the execution of any thread (a *receive*, *send*, or any *work* thread) within a CLAM process. Because of this, the time that elapses between consecutive activations of a CLAM process's *receive* thread is random. This time depends on, among other things, the number of processes in the system, the demands placed by each on the OS, and the length of a *receive* thread's time slice. Thus, keeping thread time slices small is one way of ensuring smaller expected elapsed times.

Adaptation based on receive-buffer statistics

The effectiveness of a given value of the time slice (T) can be gauged by the average number of packets that a *receive* thread tends to find queued at the network receive buffer when it begins to run. Each time it gets control, a nonpreemptive *receive* thread (over) estimates the number of waiting packets as the count of the number of packets it reads before the buffer is found empty. If activated too frequently, the *receive* thread will

often find the receive buffer empty, with no packets to read. Though this may help keep packet-loss low, excessive context switching to the *receive* thread and back will generate unnecessary overheads and may cause other threads to starve for CPU attention. On the other hand, if activation is not frequent enough, the *receive* thread may often find the receive buffer full. This is indicative of potential packet-loss.

If the *maximum* queue size Q_{max} physically allowable in a network receive buffer can be determined, a reasonable approach is to activate a *receive* often enough to keep the queue size within an acceptable range. Naturally, the the high end should be kept below Q_{max} as often as possible, to keep the packet loss (l_p) low. Indeed, this very same approach is used in congestion avoidance algorithms [14]. But in this case, the scheme is used to prevent buffer overflow at a receiver, instead of at gateways or switches located en route between a sender and a receiver. In the CLAM overflow control scheme, the control signal (i.e., the queue size) is sensed at the receiving end; the control action (i.e., *adapting* the time slice) also takes place at the receiving end. Thus, the control loop lies fully within the receiver. It is not distributed across the network, as in the most congestion avoidance schemes. Because of this, we can expect a fast and more reliable response.

The quantity Q_{max} is machine and environment dependent. A simple scheme to estimate Q_{max} at a given machine is as follows. A *consumer* process is initiated on the machine: it opens a UDP port, awaits keyboard input, and reads as many packets as possible from this port upon keyboard input. A *producer* process is initiated on a remote machine, to avoid the effects of local machine optimizations. While the *consumer* awaits keyboard input, the *producer* opens a UDP port and floods the *consumer*'s port with packets. The maximum number of packets read by the *consumer* is the required estimate L_{max}. That is, L_{max} is an estimate of the maximum number of packets Q_{max} that the UDP protocol may queue, at any given time, at a port on the receiving machine. The CLAM system uses an automatic procedure to obtain L_{max} for each machine during system initialization.

The idea behind the adaptive algorithm is to allow a *receive* thread to alter the value of T, thus altering its frequency of activation with the kind of packet load it encounters at the network receive buffer. Unfortunately, even though a CLAM process can allocate portions of its time slice to resident threads in any manner, how often its gets to run and for how long it runs before being displaced from the CPU is decided by the OS. Context switching between processes occurs at unpredictable times and processes may hold a CPU for equally unpredictable durations. This is largely a function of the OS scheduling policy and system load. Despite this problem, however, we have found simple time slice adjustment schemes to be highly effective in keeping *response times* (i.e., the interval of time between consecutive activations) of the *receive* thread small.

Let X_i denote the number of packets read from the network receive buffer by a *receive* thread on its i-th activation, for some large i. Because an OS context switch may have occurred between a *receive* thread's $(i-1)$-th and i-th activations, it is impossible for this thread to quickly determine if a large value of X_i is solely a function of network input load and the current length T_i of the time slice, or if a large value of X_i is the result of an OS context switch. Utilizing system calls for timing is expensive, and it is precisely such extra overheads that we wish to avoid.

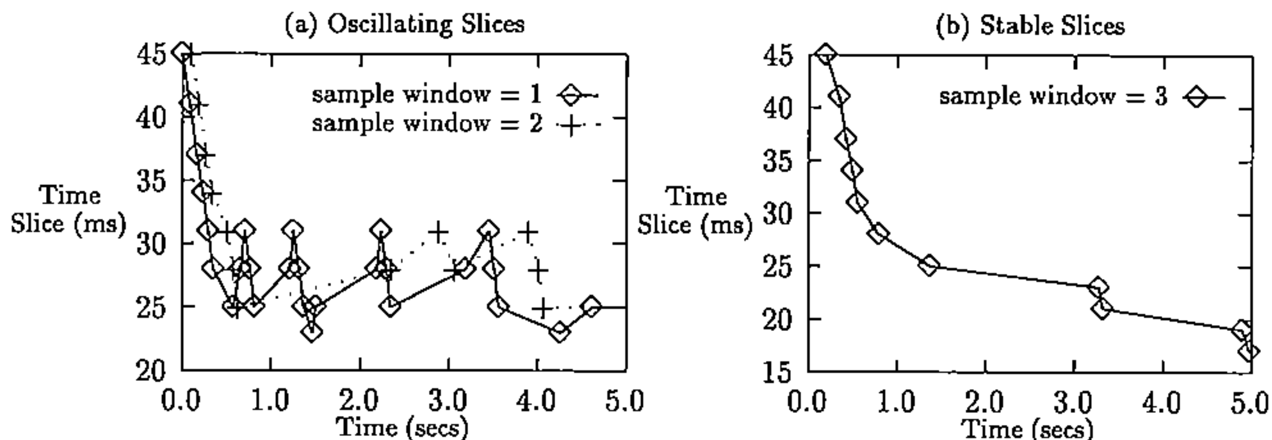

Figure 6: Effect of sample window size on response-time ($t_a = 10$ ms)

If a *receive* thread computes a new time slice length T_{i+1} purely as a function of T_i and X_i, its response-time will tend to oscillate with X_i. To effect a more stable response, it is advisable to compute T_{i+1} as a function of T_i and a more stable value, such as that based on the set $X_{i-(n-1)}, \ldots, X_{i-1}, X_i$ of n consecutive queue size estimates. The quantity n defines the size of the sample window. Any oscillation and related overhead can be further reduced by using hysteresis, as suggested in [15]. Details of how this feature is used by the CLAM overflow control algorithm is described below.

Experimental results measuring the ability of a *receive* thread to respond to input load, as a function of the sample window size n, can be seen in Figures 6(a) and 6(b). As predicted, for $n = 1$ and $n = 2$, the response time fluctuates considerably, even for relatively large values of T. With $n = 3$, however, the response time appears to stabilize, though smaller windows clearly yield a more rapid response for certain values of T. We have found too frequent an adjustment of T to be an additional source of overhead. A window size of $n = 3$ was found adequate for our purposes and yields an acceptable average response time of about 1 second. Compare this, for example, with the granularity of the TCP/IP timer, which is 500 ms in BSD Unix.

3.4 Slicing with Adaptive, Nonpreemptive Receive (SANR)

The adaptive scheduling algorithm works as follows. Each time it is activated, a *receive* thread estimates the size of its receive queue, obtaining estimate X_i on activation i. If X_i and all $n - 1$ previous estimates of the receive queue size are found to be less than some minimum acceptable size L_{min}, or if these estimates are all greater than some maximum acceptable size βL_{max}, where $\beta \leq 1$, the *receive* thread's time slice length must be adjusted. This adjustment is done to help push the value of X_j into the interval $[L_{min}, \beta L_{max}]$ for $j > i$. The new time slice length T_{i+1} of the *receive* thread is computed as

$$T_{i+1} = \begin{cases} T_i\alpha & \text{if } Y_L(i,n) = 1 \\ T_i/\alpha & \text{if } Y_H(i,n) = 1 \\ T_i & \text{otherwise} \end{cases}$$

where $Y_L(i,n) = \prod_{j=i-(n-1)}^{i} I_{\{X_j<L_{min}\}}$, $Y_H(i,n) = \prod_{j=i-(n-1)}^{i} I_{\{X_j>L_{max}\}}$ and α is a suitably chosen constant greater than 1. Here, $I_{\{a<b\}}$ is the indicator function which is 1 if $a < b$, and 0 otherwise. At most one of the indicator products $Y_L(i,n)$ and $Y_H(i,n)$ can take on the value 1 for any value of i. The constant α should be chosen in a way that ensures an acceptable granularity in time slice lengths.

The results shown in Figure 7 correspond to experiments in which we used values of $n = 3$, $L_{min} = 1$, $\alpha = 1.125$, and an experimentally determined value of $\beta L_{max} = 5$. A value of $\alpha = 1.125$ was found to yield an acceptable granularity in time slice lengths. A well-chosen value of α (e.g., $1 + 1/2^3$) is one which permits binary shifts, addition, and subtraction in code to circumvent the use of more expensive multiplications and divisions. Thus, efficient calculation of new time slice values is enabled by these rapid operations, as

```
T(i+1) = T(i) + ( T(i) >> 3 ); /* right-shift, addition */
T(i+1) = T(i) - ( T(i) >> 3 ); /* right-shift, subtraction */
```

In Figure 7(a) we observe a trend in which the amount of work done per second increases with interarrival time. In Figure 7(b), there is a peak in packet loss when interarrival time is very small. This is due to the granularity of the time slice length (10 msecs). For interarrival times approaching 10 msecs, the adaptive algorithm is unable to effect further reductions in time slice length. As as a result, packet loss is unavoidable. In Figure 7(c), we see that the adaptive algorithm defines time slice lengths in a such a way that they are near perfectly (positively) correlated with interarrival times. Though a *receive* thread is given more respite when interarrival times are large, packet loss is still kept low. It is instructive to note that this SANR scheduling algorithm outperforms the nonadaptive SPR and SNR algorithms for almost all arrival rates.

3.5 Interrupt-driven, Nonpreemptive Scheduling (SINR)

Consider a scheme in which all threads, including the *work* threads, are scheduled to run for fixed (i.e., nonadaptive) time slices. Recall that in our experiments, the only threads which may run are a *work* thread and a *receive* thread. As in the previous case, assume that a *receive* thread runs nonpreemptively once it is activated. One possible way of activating a *receive* thread is on asynchronous input: it is scheduled to run whenever a SIGIO interrupt occurs. The SIGIO interrupt indicates the arrival of one or more packets at the receive buffer monitored by a *receive* thread. The motivation for this approach should be obvious. The *receive* thread should run only when input packets await processing.

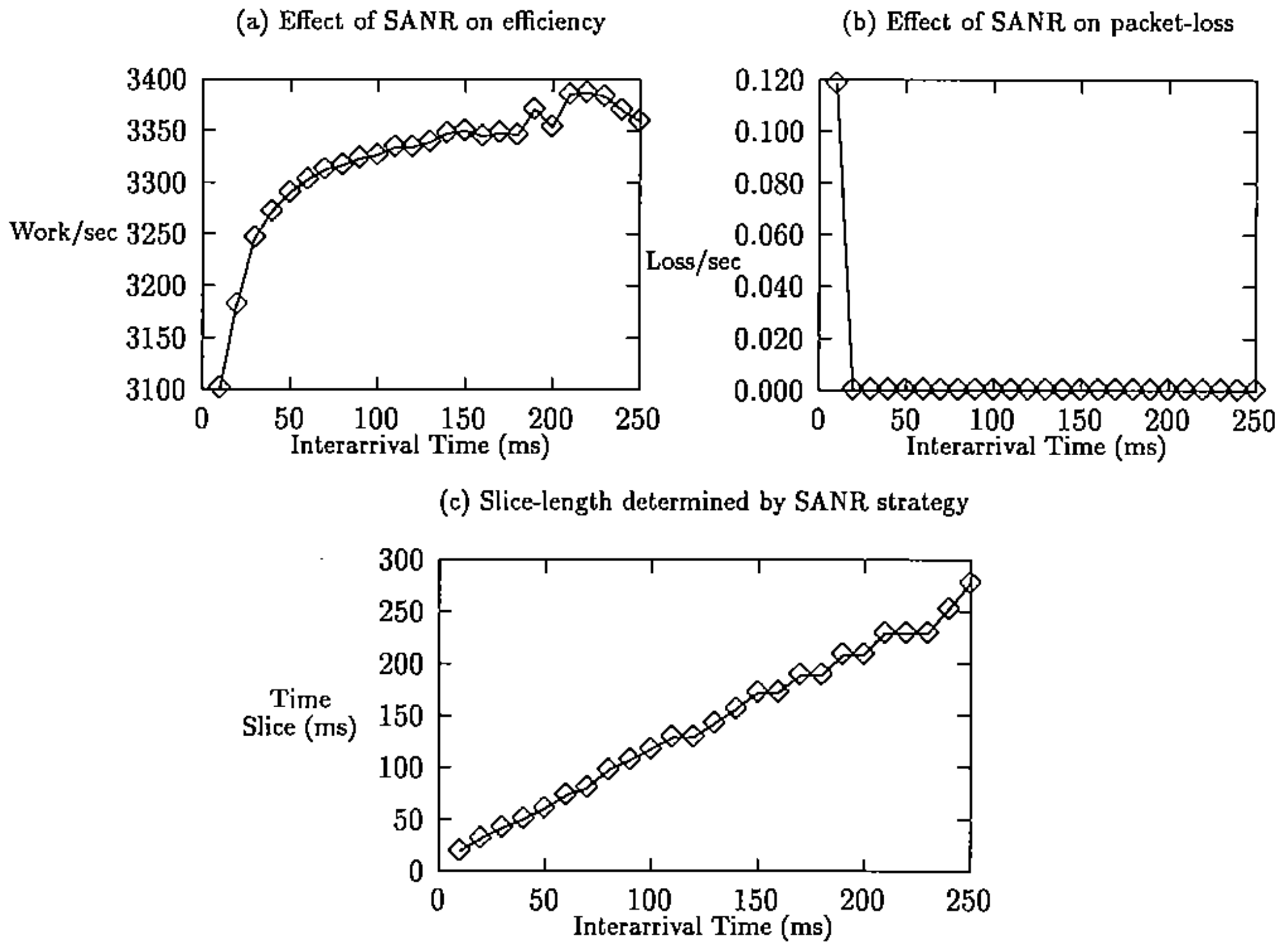

Figure 7: Performance of SANR strategy

In most threads systems, as in Ariadne [12], threads may run at different priority levels. When a thread relinquishes CPU control voluntarily, or is preempted, the threads system scheduler ordinarily places this thread at the tail end of its priority queue and gives CPU control to the thread at the head of the highest-priority queue. In general, when a SIGIO interrupt is generated by an incoming packet, the threads system may not be able to immediately run or even schedule an appropriate *receive* thread. This happens, in particular, when the threads system is executing within a critical section. Any attempt to bypass this restriction has the potential of complicating portability requirements of the software. Because of this, there may be some delay between the arrival of one or more packets and the activation of a *receive* thread.

Instead of context switching to a *receive* thread on each SIGIO interrupt, only the first such interrupt triggers a context switch. When a SIGIO signal is delivered, a *receive* thread – which ordinarily sleeps, blocked on a semaphore, while other threads run – is awakened by the threads system and moved to the head of the highest-priority queue. It runs immediately after the current thread relinquishes control of the CPU. If the signal arrives while the threads system is in a critical section, this action is taken only after the critical section is exited. Naturally, if the time-to-activation of the *receive* thread is large, it will not be able to respond quickly to packets which arrive in quick succession. On the other hand, imagine a scheduling scheme in which the CPU is usually allocated to a number of work threads, in round-robin fashion, before being allocated to a *receive* thread. The use of I/O interrupts enables the threads system to break the round-robin cycle and rapidly attend to a receive buffer.

In Figures 8(a) and 8(b) are shown measurements of the amount of work done per second by the *work* thread, and the average packet loss, respectively. Each measured quantity is graphed against packet interarrival times. For the given interarrival times, there appears to be an upper bound to the amount of work that can be done by the *work* thread. As seen in Figure 8(b), the use of a fixed, nonadaptive time slice can cause serious packet loss when packet arrival rate is high. For larger time slice lengths, the response-time of the *receive* thread is bound to be larger, ensuring a higher likelihood of packet loss for a given arrival rate. Since we use only a single *work* thread, there is no significant difference between this interrupt-driven approach and the SPR and SNR approaches. With many *work* threads, and thus a round-robin scheduling of work, interrupt-driven scheduling (SINR) will outperform the SPR and SNR scheduling algorithms because it ensures a smaller time-to-activation of a *receive* thread.

3.6 Interrupt-driven, Adaptive Scheduling (SAINR)

Problems encountered in the use of fixed, nonadaptive time-slicing with SIGIO-based scheduling of a *receive* thread can be overcome by resorting to time slice adaptation. The idea is to reduce a *receive* thread's response-time when packet arrival rate is high, and to reduce context switching overhead between other threads (i.e., *work* threads) when packet arrival rate is low. With SAINR, the approach used to adjust the length of a time slice is identical to the scheme presented in the SANR scheduling strategy.

Time slice length adaptation is not possible in the absence of packet arrivals, because after all, it is the responsibility of a *receive* thread to perform the adjustment. If no

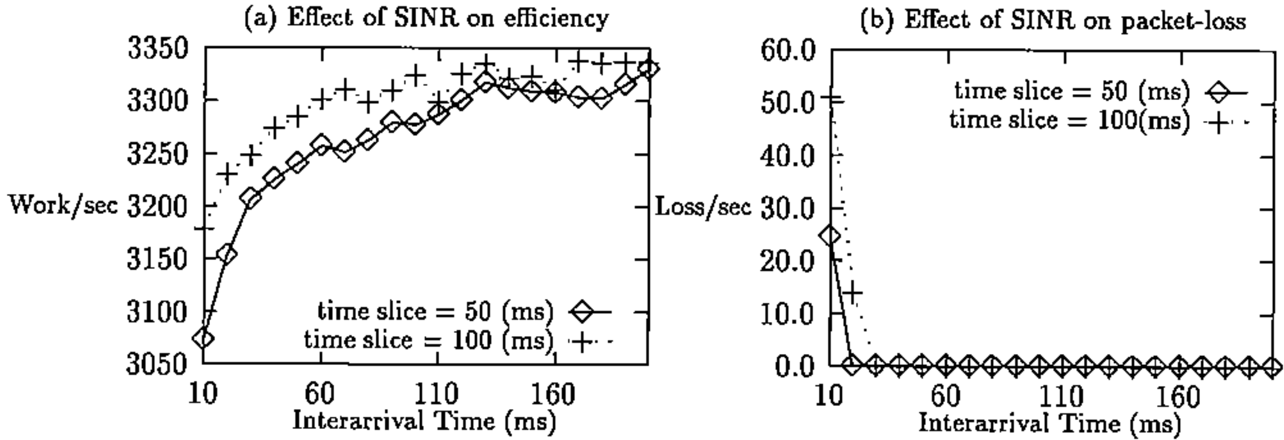

Figure 8: Performance of SINR strategy

packets arrive for a while, a *receive* thread is not given control and is thus not able to make a time slice adjustment. If a sudden burst of packets was to arrive after a period of silence, even though a *receive* thread may respond quickly (with the help of the interrupt feature), it will resize the time slice parameter to a value that may unnecessarily increase context switching overheads for *work* threads.

To solve this problem in CLAM, we use a special *timer* thread. This thread activates a *receive* thread every second. This is done repeatedly, even in the absence of arriving packets, to enable a *receive* to execute the adaptive algorithm. In this way, were a burst of packets to arrive, followed by a period of inactivity, the time slice length will automatically and slowly be readjusted upwards. In Figures 9(a), 9(b), and 9(c) are shown measurements made for the SAINR scheduling algorithm. The performance shown is similar to that obtained with the SANR scheduling algorithm. Observe, however, that Figure 9(a) shows a reduced efficiency compared to that seen in Figure 7(a), probably due to SIGIO interrupt overheads.

3.7 Experiments with bursty and correlated packet arrivals

In all the experiments described thus far, the time between successive packet generations at a source was kept almost constant. A simple procedure was used for generating packets. A sender injected packets into the local net with an almost constant delay between consecutive injections. Any jitter experienced in sends was due to OS context switches at the sending machine, and any jitter experienced by packet arrivals at a receiver was due to network load. To model more realistic traffic, with interleaved burst and silent periods, we used the following simple Markovian strategy. At any given time, a sender is made to reside in one of two possible states: **H** (high) and **L** (low). If a sender is in state **H**, the delay between consecutively sent packets is set to 10 ms. If a sender is in state **L**, the delay between consecutively sent packets is set to 100 ms. The sender makes a state transition after sending a packet, using the following transition matrix:

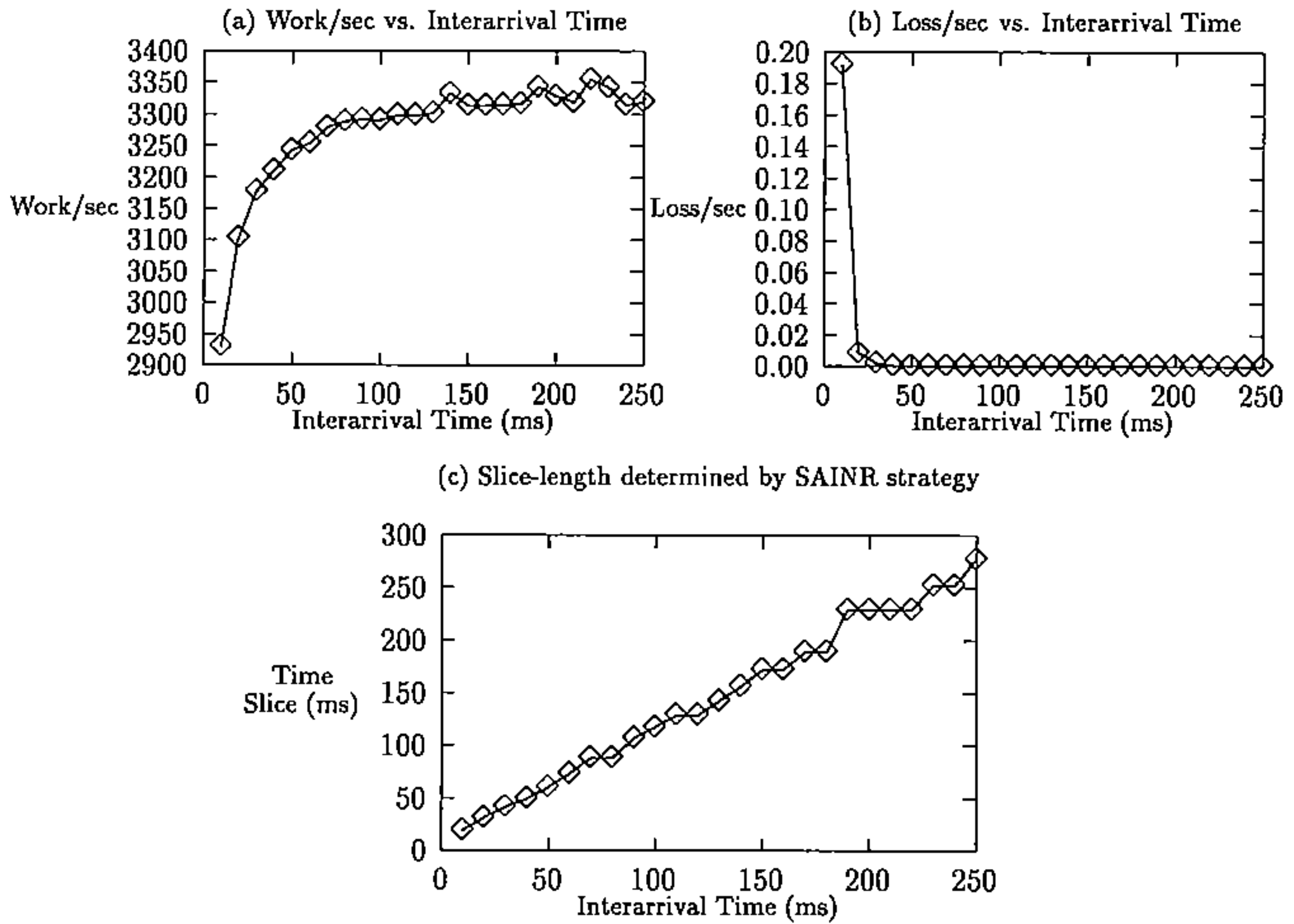

Figure 9: Performance of SAINR strategy

$$\begin{array}{c c} & \begin{array}{cc} \mathbf{H} & \mathbf{L} \end{array} \\ \begin{array}{c} \mathbf{H} \\ \mathbf{L} \end{array} & \begin{pmatrix} \gamma & 1-\gamma \\ 1-\delta & \delta \end{pmatrix} \end{array}$$

The sequence of states visited by such a sender is governed by a Markov chain with the given transition probability matrix. Further, consecutive states are correlated, with correlation coefficient given by $\rho \equiv \gamma + \delta - 1$.

To measure the effects of correlated arrivals on the scheduling algorithms we need to examine how work efficiency and packet loss vary with ρ. Because multiple combinations of γ and δ yield the same value of ρ, it is necessary to choose an appropriate parameterization. For the experiments described below, we used a value of $\gamma = (1+\rho)/2$, with this choice yielding a symmetric transition probability matrix. As γ approaches 1, ρ also approaches 1, and packets either arrive in rapid succession (bursts) or slowly. As γ approaches 0, ρ approaches -1, and packet arrival is low, as the sender transitions quickly between sending off a packet fast (after 10 ms), and sending off a packet slowly (after 100 ms). Larger values of ρ are indicative of more stable interarrival times.

In Figures 10 and 11 are shown plots of efficiency and packet loss for all scheduling algorithms. For the nonadaptive schemes, with results graphed in Figure 11, the time slice value used was 50 ms. We see that the adaptive time slicing algorithms outperform their nonadaptive counterparts in terms of work efficiency. However, with respect to packet loss, the nonadaptive schemes appear to show better performance. This is probably due to the fact that the minimum arrival rate of 10 ms is not too small compared to the length of fixed time slice used, namely 50 ms. If the fixed time slice length is increased from 50ms to 100ms, to get better efficiency, packet loss can be expected to increase over that offered by the adaptive schemes. In other words, the time slice of 50 ms used in the experiments with the nonadaptive algorithms is close to the optimal value for the two arrival rates used: 10 and 100 ms. The SANR algorithm performs better, for most values of the parameter ρ, than the SAINR algorithm.

As is to be expected, all algorithms perform equally well with respect to packet loss as long as ρ is not high (see Figure 11(b)). There is a significant difference, however, in how the algorithms perform with respect to work efficiency, as can be seen in Figure 11(a). In Figure 10(b), packet loss appears to be uniformly high, relatively speaking, for correlated arrivals. This occurs because the sampling-window of $n = 3$ is sufficiently large that adaptation to arrivals while the sender is in state **H** is not fast enough. For example, when ρ is near 1, a sender that transitions from **L** to **H** may send a large number of packets before the *receive* thread (operating with a time slice length of roughly 150 ms, as given by Figure 7) finally realizes that it is time to reduce the length of the time slice. Packet-loss can be significant during this period. We are currently investigating variations of the adaptive strategy that may adapt window size to changing traffic patterns.

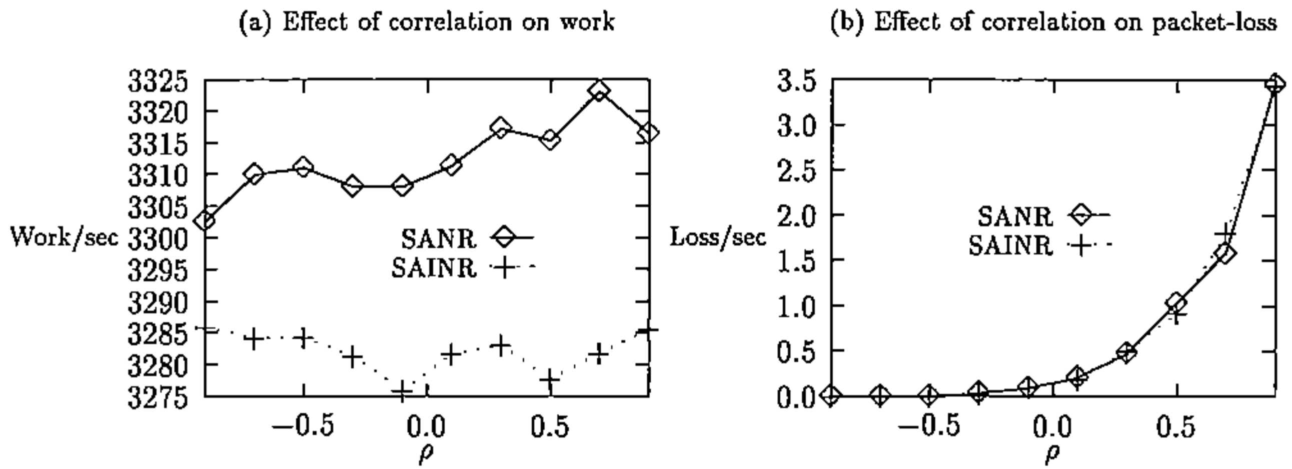

Figure 10: Performance of Adaptive strategies with Markovian arrivals

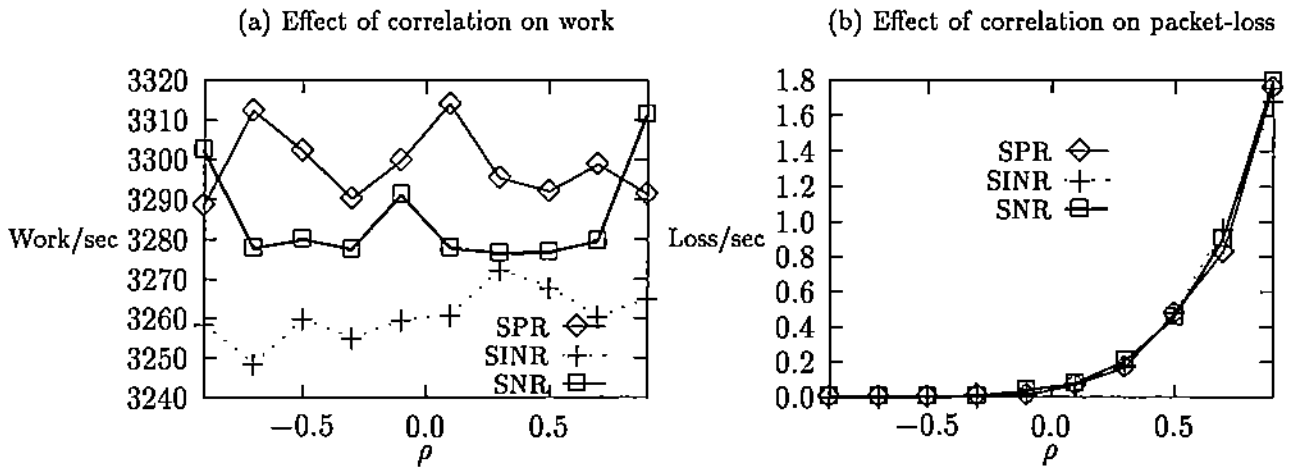

Figure 11: Performance of Nonadaptive strategies with Markovian arrivals

4 Conclusions and Future Work

Though this work is but a first step towards novel protocols and their implementation, we have found threads-based adaptive scheduling schemes to be crucial in achieving efficiency, through overlap of communication and computation. Adaptivity enables reduced communication overheads by lowering retransmission rates, and allows efficient reliable transmission with the CLAM system. Our results indicate that the adaptive time slicing algorithms have the potential to yield significantly higher amounts of compute-bound work than algorithms that do not adapt to network load.

Interrupt driven schemes appear to yield reduced work efficiency, possibly because of attendant I/O interrupt overheads. Despite this, interrupt-driven schemes may still be beneficial because of their capacity to respond quickly, and keep packet loss low when packet arrivals occur in bursts with periods of silence in between. Some disadvantages of the interrupt-driven schemes include potential problems with portability and maintenance. Because of this, the CLAM system resorts to adaptive time slice scheduling (SANR), using I/O interrupts only as an option.

It is instructive to note that CLAM's model of overflow control is useful when the bottleneck is located at a receiving node, as is the case for most LANs and high bandwidth networks. However, upper layers must still resort to regular congestion control and/or congestion avoidance algorithms to prevent congestion at intermediate gateways or switches. This covers the case where a bottleneck lies at some intermediate point between a sender and a receiver, and is not located at the receiver. Indeed, this situation is typical in the WANs of today.

The techniques presented here comprise the initial stages of a project aimed at the provision of efficient point-to-point and multiway message transport with different, selectable qualities of service: reliable, unreliable, and prioritized delivery. A portion of this work has already been completed. Preliminary results appear promising, compared to the performance of related systems which guarantee reliable service based on UDP. In addition to reliable and in-order delivery, we have successfully used CLAM to implement transaction oriented delivery [16] and active messages [17, 18], with reports on this work in preparation. We currently plan on adding other features, including support for partial-order service, as proposed in [19]. Such services are fundamental in communication environments that support distributed multimedia applications.

References

[1] T. Montgomery. *Design, Implementation, and Verification of the Reliable Multicast Protocol.* West Virginia University, Morgantown, West Virginia, 1994.

[2] J. Cooperstock and S. Kotsopoulos. Why use a fishing line when you have a net? an adaptive multicast data distribution protocol. In *USENIX Technical Conference Proceedings*, 1996.

[3] V.Sunderam, G.Geist, J.Dongarra, and R.Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–545, April 1994.

[4] R. Butler and E. Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20(4):547–564, April 1994.

[5] *MPI: A Message-Passing Interface Standard*, June 1995.

[6] P. Bridges, N. Doss, W. Gropp, E. Karrels, E. Lusk, and A. Skjellum. *Users' guide to mpich, a Portable Implementation of MPI*, October 1995.

[7] B. Topol. The CONCH system. Technical report, Emory University, Atlanta, GA. 30322, 1993.

[8] A. Edwards and S. Muir. Experiences implementing a high performance TCP in user-space. In *SIGCOMM'95*, pages 196–205. ACM, 1995.

[9] C. Maeda and B. Bershad. Protocol service decomposition for high-performance networking. In *14th ACM Symposium on Operating Systems Principles*, 1993.

[10] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. In *SIGCOMM'93*, pages 64–73. ACM, 1993.

[11] D. Tennenhouse. Layered multiplexing considered harmful. In *Proceedings of the 1st International Workshop on High-Speed Networks*, pages 143–148, 1989.

[12] E. Mascarenhas and V. Rego. Ariadne: Architecture of a portable threads system supporting thread migration. *Software-Practice and Experience*(to appear), 1996.

[13] *IEEE Standard for Distributed Interactive Simulation – Application Protocols*, IEEE Standards Department 1995.

[14] R. Jain. A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *ACM Computer Communication Review*, 19(5):56–71, October 1989.

[15] K. Ramakrishnan and R. Jain. A binary feedback scheme for congestion avoidance in computer networks with connectionless network layer. In *SIGCOMM'88*, pages 303–313. ACM, 1988.

[16] R. Braden. *T/TCP : TCP Extensions for Transactions Functional Specification.* Network Working Group, ISI, July 1994.

[17] T. von Eicken. *Active Messages: an Efficient Communication Architecture for Multiprocessors.* PhD thesis, University of California at Berkeley, 1993.

[18] J. Seizovic. The reactive kernel. Technical Report CS-TR-88-10, California Institute of Technology, 1988.

[19] T. Connolly, P. Amer, and P. Conrad. *An Extension to TCP: Partial Order Service.* Network Working Group, University of Delaware, November 1994.