1996

The CLAM Approach to Multithreaded Communication on Shared-Memory Multiprocessors: Design and Experiments

Joan Carlos Gomez

Edward Mascarenhas

Vernon J. Rego

Purdue University, rego@cs.purdue.edu

Report Number:

96-036

Gomez, Joan Carlos; Mascarenhas, Edward; and Rego, Vernon J., "The CLAM Approach to Multithreaded Communication on Shared-Memory Multiprocessors: Design and Experiments" (1996). *Department of Computer Science Technical Reports*. Paper 1291.
<https://docs.lib.purdue.edu/cstech/1291>

**THE CLAM APPROACH TO MULTITHREADED
COMMUNICATION ON SHARED-MEMORY
MULTIPROCESSORS: DESIGN AND EXPERIMENTS**

**Juan Carlos Gomez
Edward Mascarenhas
Vernon Rego**

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

**CSD TR-96-036
June 1996**

The CLAM Approach to Multithreaded Communication on Shared-memory Multiprocessors: Design and Experiments*

Juan Carlos Gomez, Edward Mascarenhas and Vernon Rego

Department of Computer Sciences

Purdue University

West Lafayette, IN 47907

USA

Abstract

The Connectionless, Lightweight, and Multiway (CLAM) communications environment provides efficient and scalable multi-protocol support for distributed applications using multimodal data. This paper presents motivation and design decisions behind the CLAM system, along with two simple but effective scheduling algorithms for the simultaneous support of multiple, threads-based user-space protocols. One algorithm is readily portable to shared-memory multiprocessors, and enables two or more protocols to coexist within a process's space. We present experimental results on the performance of both algorithms. To demonstrate the utility of the approach in a typical application, we present the results of an experiment in which one algorithm is used to implement a Distributed Interactive Simulation (DIS) application. The performance of this implementation is compared to the standard UDP-based implementation of DIS for the given application.

* Supported in part by ONR-9310233, ARO-93G0045 and BMDO-34798-MA.

1 Introduction

In extant software systems for high-performance distributed computing, interprocess communication is effected through explicit message transfers. Single-threaded processes communicate with one another at well defined points in their execution sequences. A send operation at a sending process is paired with a receive operation at a receiving process. In general, both operations are encapsulated within communication services layered upon TCP/IP or UDP protocols (such as in PVM [1], P4 [2], MPI [3], for example, which execute on workstation clusters) provided by the OS kernel, or upon specialized communication facilities tied to a given hardware environment (e.g., Intel NX library [4], IBM EUI [5, 6]). Though this model has proven adequate for many distributed applications with known or well-structured process interactions, there is a recognized need for improved communicability in situations where process interactions are either unstructured or unpredictable or both. To satisfy this need, proposals for threads-based services in messaging systems have begun to appear (e.g., TPVM [7], Nexus [8], LPVM [9], ParaSol [10]). For the most part, these proposals involve high-level enhancements to interprocess communication based on threads and related services.

In contrast to threads-based communications services provided at higher levels, we are motivated by the need for an improved low-level communications framework. Such a framework should be independent of how local or remote computations are enabled, provide a versatile and extensible environment with a number of useful communications services, and also support scalability. With appropriate support for multithreaded computation and communication, both for uniprocessors and shared-memory multiprocessors, such a substrate can support higher level libraries in high-performance computation and collaboration applications. In support of our proposal, we have designed and implemented both unreliable and reliable communication services in the form of a library which can serve as a scalable and extensible communications substrate.

We advocate the implementation of low-level communication protocols in user-space [11]. Advantages of this approach include a more responsive binding between client and protocol actions, more efficient scheduling of compute- and communicate-functionality, and improved handling of asynchronous actions [12, 13]. Consider, for example, the difficult problem of scheduling the *receive* functionality in a distributed system. If the underlying protocol is unreliable and buffer space is limited, latency and packet loss can be kept low only if incoming messages are processed rapidly. Frequent scheduling of the receive functionality may enable rapid processing, but only at the expense of undesirable overhead. For example, most UDP [14] implementations provide a bounded receiving queue at each port. This bound is either due to a hard limit on the number of buffers, or on the total amount of memory allocated [15]. If arriving messages are not handled promptly by a receiving process, packet loss is inevitable. For an application, this will mean retransmission delays and unused or wasted CPU cycles.

Current message-passing systems use one of two basic approaches in scheduling the receive functionality: polling, and asynchronous I/O interrupts. The strategy of polling for message arrival is simple and portable, but leads to high overheads if used in a naive fashion. Further, it can also lead to high packet loss and even to deadlock. In some communication libraries, polling is done only when an application invokes specific library primitives, or at regular time intervals. Depending upon how packets arrive, this may lead to high polling overheads (if polling is too frequent), high packet loss (if polling is too infrequent), and decreased throughput.

I/O interrupts are useful in that they can help reduce latency and eliminate polling overheads. On the other hand, software supporting I/O interrupts is not easily portable across different platforms, and implementation and debugging of code with such interrupts is hard. Also, depending on the type of hardware used, interrupt driven input may be more expensive than periodic polling.

In this effort, we show how threads may be used to implement an efficient communications substrate for distributed applications. We base our work on the Ariadne [16, 17] portable threads system which was designed for experimentation with mobile-thread-based simulation and general distributed computing. In previous work [11], we studied threads-based algorithms for adaptive polling and asynchronous I/O interrupts on uniprocessor systems, showing how efficient scheduling of communication threads can simultaneously increase an application's CPU utilization and decrease its incoming packet loss. In this work, we study the effects of these algorithms in the multiprocessor setting, and present yet another scheduling algorithm which can operate on both uni- and multiprocessor systems without modification.

In this work, we study the effect of two threads scheduling algorithms on unreliable messaging. The work described in [18] extends this work to reliable messaging and other support (e.g., real-time, voice and video) on uni- and multiprocessor LANS and WANs. To enable both experimentation and production mode operation, we built a software prototype known as the CLAM (connection-less, lightweight and multi-way) communications environment. In Section 2 we present a brief overview of CLAM, and in Section 3 we present different features of communication thread scheduling algorithms [11]. Two select algorithms, for scheduling threads on multiprocessors, are described in Section 4, and experimental results on their performance are given in Section 5. To demonstrate the utility of the CLAM approach, we present a simple case-study, in Section 6, comparing the performance of a standard DIS (Distributed Interactive Simulation) implementation which is supported by the usual UDP multi-cast, with a CLAM-based DIS implementation. A brief conclusion is presented in Section 7.

2 Multimedia Communication Architectures

In terms of current technology, it is critical that protocols for distributed multimedia applications be able to support unstructured and unpredictable transfer modes and data access patterns over a single communication channel. There are two important questions that must be answered in the design of such protocols. First, how should the requisite functionality be structured within a given OS ? Second, where should the multiplexing and de-multiplexing functionality be located, for maximum effectiveness ? Though answers to these questions can be many and varied, some structures will perform significantly better than others. Further, these will impact strongly not only upon application performance, but also upon the overall performance of the environment in which the application executes.

To enable us to investigate strategies for implementing and structuring lightweight protocols for multi-way, collaborative applications supporting multi-modal data transfer, we designed the CLAM communications system. Interfaced with a user-space threads system, CLAM enables various modes of data transfer within a single, integrated framework. By eliminating the need for distinct protocols operating on connections and on different software systems, we eliminate attendant OS overheads and enable efficient multiway interoperability in heterogeneous

Figure 1: CLAM Implementation Structure

distributed environments.

As shown in Figure 1, CLAM is currently layered upon UDP [14] but is readily portable to any best-effort delivery system. Integrated communications support is a result of encapsulating all necessary functionality within a single process. A modular implementation of protocols in user-space makes the system suitable for a wide range of applications, from standard messaging applications to those requiring reliable and real-time transfer. Examples of its applicability include support for high-performance distributed computing (e.g., PVM, MPI), distributed interactive simulation [19], and computer supported cooperative work (CSCW).

In most distributed multimedia applications, functionalities with distinct communication requirements are mapped into distinct processes [20]. For example, if an application requires both reliable stream transfer as well as real-time voice transfer, these services are provided by two different processes. One process provides support for reliable stream transfer, and the other provides support for real-time voice transfer. This almost automatically implies that the multiplexing/de-multiplexing functionality, for reasons of efficiency, must lie within the kernel (see Figure 2).

One advantage of this approach is that application programs will have relatively simple structures. Disadvantages include expensive OS context-switching overheads, and the high cost of software integration: the assembling of different parts of an application into a whole. If communication requirements are diverse and support is provided by many processes, high context-switching overheads can hinder efficient message delivery. This is exacerbated in situations where interaction between different parts of an application makes communication between either local or remote processes necessary.

The architectural design of the CLAM system was influenced by our view of what future applications will require: support for seamless multimodal communication and computation in virtual collaboratories. Such applications may be expected to exhibit a high level of coupling and interaction among their constituent parts. For example, consider an application in which voice either triggers or controls a reliable transfer, or one in which high-performance computational transformations are applied to distributed data for subsequent visualization and interaction at multiple sites. The structure shown in Figure 2 is not appropriate for such applications.

Figure 2: Multiprocess Multimedia Application

The task of implementing a process which provides integrated support for all the functionality required by a distributed multimedia application may seem onerous. But with the support of an appropriate user-level threads system the task is not difficult. The threads system simplifies integration, and enables cheap local interaction – through local variables – between different parts of the application. Most importantly, such interactions between threads require at most the cost of a thread's context-switch, as opposed to the relatively high cost of a context-switch between processes at the kernel or OS level (see Figure 2).

In our approach, there is some flexibility with regard to where the multiplexing and de-multiplexing functionality is located. One alternative is to drive some of this functionality up to the application level, as shown in Figure 3(a).

Performing multiplexing and de-multiplexing at the user-level, in a way that is transparent to the user, has some limitations. If the underlying communications protocol already does multiplexing/de-multiplexing within the kernel, we would be adding some overhead by replicating this functionality. Further, because all packets will queue at a single queuing facility before serial processing, the latency of one type of traffic can be significantly effected by other types of application-generated traffic.

An alternate approach is to leave all multiplexing and de-multiplexing operations to the kernel, as shown in Figure 3(b). This is appropriate when a best-effort delivery protocol has such a provision in kernel-space. A key advantage of this approach is that it helps separate different types of traffic, at least in user-space, thus reducing the effect that one type of traffic has on the latency or throughput of another. This scheme is modular, and more scalable than that shown in Figure 3(a). A disadvantage of this approach, however, is that it may entail higher overheads. Unlike the scheme shown in Figure 3(a), where all traffic is managed at a single point, the scheme shown in Figure 3(b) will require traffic management at multiple I/O points, where each point corresponds to a distinct type of traffic. Because multiway connections and scalability is an important objective, the CLAM design is based on the latter scheme.

Figure 3: Uniprocess Multimedia Application

Multimedia Protocol Structures

In our view of distributed multimedia support, the chief requirement is that processes have the ability to transmit and receive distinct types of data, in distinct modes of operation, using an integrated communications framework. This is in contrast to support from different processes for different media. The design and implementation of communication frameworks that can satisfy this requirement, even in a limited form, is complicated. Designing for generality often leads to poor efficiency in applications with specific requirements, precisely because of generality-related overheads [21]. Further, if the design leads to heavyweight protocols, their implementation and maintenance will be nontrivial.

Despite the problem of how integrated multimodal data transfer is to be supported, there is a strong motivation for providing such support. An increasing number of applications seek to exploit communication for the transfer of a wide range of media (e.g., voice, video, graphics and text) with an equally wide variety of access patterns (e.g., request/response, bulk transfer, real-time transfer) and process configuration options [22]. Also, applications may configure processes in such a way that communication is point-to-point, one-to-many, or many-to-many. Some applications (e.g., high-performance distributed computation, collaborative systems) may have very specific communication requirements like guaranteed order of delivery, and isochronous transfer; others may have more relaxed ordering requirements or require no order at all (e.g., DIS [19]). Protocols are expected to satisfy these requirements over a wide spectrum of network hardware options, with transmission rates ranging from K-bits/sec to G-bits/sec.

Kernel-Space Protocol Structures

A variety of different protocol structures have been proposed in recent years [12, 13, 23]. In general, these differ from one another based on how the protocol functionality is structured within an OS, and precisely where the network device management functionality is placed. We present a brief overview of some of these proposals.

Figure 4: In-kernel Protocol Structure

In general, there has been a trend to prefer in-kernel protocol implementations because of good performance and secure interoperability. Traditional in-kernel implementations, however, were not designed for low-latency operations with multiple media, and are thus not well-suited to the integrated support of multiple protocols within single processes. Further, they are cumbersome to prototype and maintain [13, 24]; performing customization operations, to improve the performance of particular applications, is difficult [23]. This difficulty stems mainly from the high cost of transferring information from user-space to kernel-space and vice-versa, an operation that is a key component of application-specific optimizations. Further, kernels are preferably kept small; the provision of multiple coexisting protocols within the kernel would only increase its size, effectively reducing space and other resources available for user-level applications. The general structure of an in-kernel implementation with provision for multiple protocols is shown in Figure 4.

User-Space Protocol Structures

There are several alternatives to the in-kernel protocol support described above. One possibility is the layout shown in Figure 5. This protocol structure, known as the single-server model [23], embeds all the system-supported protocols within a single, trusted user-level process (i.e., the server). There are variations on this scheme, with each differing on how network devices communicate with the server process. Typical ways of enabling this communication include direct memory mapping (Figure 5(a)), shared-memory (Figure 5(b)), and message interfaces (Figure 5(c)). Though this protocol structure bears a strong resemblance to the in-kernel structure, its performance is generally poorer because of the high-cost of frequent OS level context switches generated during the processing of packets.

Another possible layout for user-space protocol support is the dedicated-server model [23] shown in Figure 6. Here, an independent but trusted user-level process is dedicated to the support of each protocol. As in the case of the single-server structure, this scheme suffers from performance limitations related to the large number of context switches occurring during packet processing and delivery [24]. Protocol structures like the server models are based on

Figure 5: Single-server Protocol Structure

Figure 6: Dedicated-Servers Protocol Structure

Figure 7: User-level Protocol Structures

the provision of trusted user-level privileges. We do not consider these suitable for the needs of flexible and portable distributed applications, in part because they are difficult to implement in environments with large numbers of geographically dispersed nodes that are maintained by different system administrators.

There is a definite advantage to minimizing kernel-level interactions or depending on trusted processes in the implementation of user-space protocols. Development and maintenance of software will not involve kernel-level intrusion, and will also not require intervention on the part of system administrators. Further, these protocols may be made available as user-level libraries against which libraries may freely be linked. Two possible structures for user-level protocols are shown in Figure 7.

One approach is to implement user-space protocol libraries which interact directly with local network devices. To ensure secure operation, however, this scheme often involves the use of an additional, trusted user-level process during the connection setup phase of the protocol. Another approach, as shown in Figure 7(b), is to implement part of the protocol within a user-level library, leaving part of the protocol embedded within the kernel. This approach has an advantage in that operational security is managed by in-kernel protocol actions, and so the additional trusted user-level process is not required. A disadvantage of this approach, however, is that data transfer to and from the network requires crossing a memory boundary between user-space and kernel-space.

Advantages of User-Space Protocols

There are many advantages to implementing protocols in user-space [23, 25]. New protocol design principles like Application Level Framing and Integrated Layer Processing [26] cannot be implemented efficiently in systems where protocols are not highly sensitive to application needs. For example, regular in-kernel implementations of TCP [27] do not understand boundaries imposed by the application on any data that is transmitted, and a TCP layer cannot deliver data received out of order. In-kernel protocols are generally unable to present an integrated set of data manipulation functions such as those for rapid data presentation and the movement of

data to and from an application's address space. Such features can be critical to the performance of applications that are highly sensitive to rapid data transfer.

Protocols implemented in user-space have been shown to perform at least as efficiently as in-kernel implementations; given appropriate low-level hardware control, user-space implementations have even outperformed kernel-space implementations [25, 28, 29]. Performance improvement is generally the result of a reduction in the number of memory-copy operations required in moving messages from an application's address-space to the network and vice-versa, and/or locating the multiplexing and de-multiplexing functions at the hardware level [29]. Enabling low-level network control through actions in user-space has proven useful in improving communication performance because of the potential for exploiting application-specific messaging knowledge. For example, in applications involving the transmission of video or voice data over networks with high error-rate, performance can be improved by enabling receipt of corrupt packets. In real-time applications, when a buffer overflow occurs, performance can be improved by configuring a hardware interface to discard older packets instead of newer ones. Such control will also enable implementation of so-called single copy protocols [29] in which data moves directly from a network interface to an application's buffers, without having to visit a kernel's memory buffer in between.

User-level protocols are generally more scalable than in-kernel protocol implementations with regard to the number of simultaneous conversations supported per process. Indeed, while kernel-space protocols impose hard limits on communication resources (number of protocol buffers, number of sockets, etc.) that a process may use, user-space implementations need not suffer from such limitations. Interaction techniques like up-calls[12, 30] are convenient when implemented in user-space. If up-call actions are made in kernel-space, a resulting failure may directly impact upon all services provided by the kernel. In contrast, a failure resulting from an up-call made in user-space will only directly impact upon the application making the call. Operating systems that support up-calls, such as the x-Kernel [31], are ideal for protocol implementations. Unfortunately, kernel up-calls to user code are not widely supported in commercial operating systems. Typically, these support kernels which use interrupts to communicate asynchronously with user processes. As mentioned earlier, interrupts can be expensive, and are a source of race conditions. In addition to these advantages, prototyping, debugging and maintaining protocols in user-space is considerably simpler and easier than for protocols in kernel-space.

Protocol structuring in CLAM

CLAM's protocols are implemented in a modular fashion, and provided as user-level libraries. Depending on their specific requirements, applications may be linked to appropriate libraries. To enable low latencies and high throughput, all protocols are kept simple and lightweight. The overall protocol structure used in CLAM is more general than the structure shown in Figure 3(b), and is more appropriately depicted in Figure 8.

The design provides for a configurable architecture that can support multiple protocols simultaneously, within a single process. Because the design is based on threads support, the architecture enables applications with multiple types of transfers and unpredictable access patterns. Every process cooperating in a distributed multimedia application will be provided with the requisite multiplexing and de-multiplexing functionality to handle several source and

Figure 8: CLAM Architecture

destination points. The key point here is that an application is viewed as a set of one or more computing threads sharing system resources with communication-enabling threads, all within a single host process.

The CLAM user-space protocol structure is based on the approach shown in Figure 7(b). Here, the protocols are partially implemented in user-space, and made available as user-space libraries against which applications may be linked. Though this approach does not provide direct access to hardware devices, it does not require the use of trusted processes, and can provide sound functionality for a fundamental design goal: multiple, coexisting protocols. We use the UDP in-kernel protocol as a base protocol, so that multiplexing and error detection is performed within the kernel. All other services, such as the transport, presentation and application-specific services are performed in user-space. This tack enables the incorporation of various application-specific optimizations within the distinct user-space protocols.

Though we designed the CLAM architecture and protocol suite to be compatible with current Internet technology, we have payed special attention to features enhancing portability to future high-speed network technology. The system is based on the Ariadne user-level threads library [16, 17] which runs stateful threads with the aid of a priority-queue based but customizable scheduler. Ariadne is enhanced with a configurable time-slicing feature through which a process may allocate portions of its CPU time-slice to different threads. An asynchronous sleep function enable threads to sleep for a specified amount of time. Ariadne also supports a shared-memory programming model in which threads are shared by distinct processes, permitting the parallel execution of shared threads.

Based on Ariadne's shared-memory support, CLAM protocol modules are able to exploit shared-memory multiprocessor architectures: various aspects of protocol functionality are implemented by different threads, all operating concurrently. Communication protocols naturally exhibit parallelism through simultaneous conversations, the send and receive functionalities of a single conversation, and also during message-processing [32, 33]. There is also the potential for the pipelined processing of different protocol layers. These are crucial design considerations, given the current shift of the performance bottleneck from network bandwidth to protocol processing overheads.

The CLAM architecture provides two basic protocol modules for environment initialization: the reliable transport module, and the unreliable transport module. Both are transaction-oriented protocols, based on the idea of active messages proposed in [34]. In addition, both protocols support remote thread activations, thus eliminating restrictions on the execution time of tasks initiated by active messages. CLAM also provides for a reliable multicast module, for efficient multiway communication and failure resilience. The implementation of this module is currently in progress.

The CLAM environment is enabled as a system of interacting processes, supported by software that is layered upon the basic reliable and unreliable protocol modules. This layer provides an environment in which CLAM processes can dynamically sign on and off an ongoing computation or collaboration. Much of the global information required in maintaining this dynamic environment is distributed among constituent CLAM processes. When required, migrant active messages dynamically locate and access information necessary for a process. Initialization of CLAM processes is achieved through a generic start-up system based on a regular Web browser and the HTTP protocol [35]. The interface enables start-up, shut-down, and some status querying of a CLAM process group. Within each CLAM protocol module, the time for application-related processing is optimized without compromising the processing requirements of network I/O or protocol processing [11].

3 Features of Scheduling Algorithms

In further developing the model shown in Figure 7(b), we arrive at the model shown in Figure 9. Within the kernel, a best-effort communications subsystem delivers packets to a *receive buffer* related to a single CLAM process or processes. With UDP, for example, the kernel routes incoming packets from its host's IP buffer to a given UDP-port receive buffer. Within user space, a *receive* thread routinely retrieves packets from a process's receive buffer and delivers these packets to an application. Since CLAM operates on top of the Ariadne threads library, all of CLAM's communication threads run as user-level threads, sharing resources with the application. Client-related processing is accomplished by *work* threads *W*, while CLAM's *send* and *receive* functionality is managed by a *send* thread *S* and a *receive* thread *R*, respectively. All threads run within application-level processes which are managed by the host's OS. To simplify the process of demonstrating how packet-loss is minimized at the receive buffer, through efficient scheduling of *R*, we assume in Section 5 that the *S* and *W* threads are lumped together in *W*. Experiments pertaining to *send* thread schedules, in the context of threads-based reliable transmission, can be found in [18]. In this study, we restrict our attention to unreliable transmission, and the scheduling of *receive* threads on multiprocessors.

On shared-memory multiprocessors, Ariadne's threads are made to reside in shared-space so that they are accessible to all processors. When a process is assigned a CPU by the OS, the process allocates portions of its time-slice to various runnable threads which await attention on the shared ready queue. We are interested in simple, effective algorithms that can maximize work done by the *W* and *S* threads, while minimizing overflow-generated packet loss at a process's receive buffer. The problem boils down to one of how a process's CPU time-slice is to be allocated between an *R* thread and other threads. As shown in a prior study of thread scheduling algorithms for uniprocessors [11], scheduling can be done in various ways. Below,

Figure 9: The Multithreaded Model

we briefly outline some of the key features of scheduling algorithms for communication threads and *work* threads.

Fixed Time-slicing

The easiest way to schedule various threads in an equitable manner is to give each thread a fixed quantum of CPU attention, sharing a process's time-slice among runnable threads in a round-robin fashion. All threads are made to run at the same priority level. When a thread's time-slice expires, its host process's scheduler returns the thread to the tail of the ready queue, and transfers control to a runnable thread retrieved from the head of the ready queue. There are two readily apparent problems with this approach: on obtaining control from the scheduler, a *receive* thread may find no work to do (i.e., its receive buffer is empty), or it finds its time-slice expiring even though it still has work left to do (i.e., its receive buffer is not empty). Both situations lead to performance degradation. The first case leads to at least two unnecessary context-switches, and the second case may lead to a wasted context-switch and packet-loss.

The frequency of a *receive* thread's activation depends on the number of threads in the system, the time-slice length, and the frequency of OS context switches. Key factors in the performance of this scheme are the size of the quantum and the packet arrival rate. If the time-slice is small, relative to packet arrival rate, unnecessary scheduling of the *receive* thread leads to high overheads. On the other hand, if the time-slice is large, relative to packet arrival rate, packet-loss can be high. With reliable transmission, this will lead to higher latencies and higher retransmission overheads. To accommodate variable packet arrival rates, a good scheduling strategy will have to repeatedly adjust time-slice length to meet an application's messaging demands.

Preemption vs. Non-preemption

To ensure some level of fairness and to prevent threads from starving for CPU attention, threads are returned to the ready queue when their time-slices expire. In particular, a *receive* thread may find itself being preempted and returned to the ready queue even though it still has work to do in emptying out the receive buffer. To keep latency low, it is preferable to run a *receive* thread non-preemptively so that it relinquishes CPU control only after it has emptied out the receive buffer.

One problem that may arise with non-preemptive *receive* thread execution is that a broken sender may swamp a receive buffer with packets, causing other threads to starve for CPU attention. A simple strategy to solve this problem is to limit the number of packets that can be read by a *receive* thread each time it runs. Though the strategy is effective in preventing starvation, it will not perform well without a mechanism for adjusting time-slice lengths to accommodate packet arrival rates.

Interrupt-driven Processing

Because a *receive* thread only needs to run when packets are present in the receive buffer, an asynchronous SIGIO interrupt can be used to schedule *receive* threads. *Send* and *work* threads can still be scheduled in round-robin fashion, using fixed time-slices. When an interrupt occurs, a *receive* thread may run non-preemptively, at a priority higher than that of other threads. If packets arrive sporadically and at critical arrival rates, a *receive* thread may encounter only one or only a few packets in the receive buffer on each interrupt. This can be a source of high interrupt-processing and context switching overheads.

When *receive* threads are freely shared between processes, paring SIGIO interrupts with particular *receive* threads is cumbersome. A packet arriving at a given receive buffer generates an interrupt meant for the *receive* thread servicing that buffer. It is up to the application to determine exactly which *receive* thread is to be run, an expensive proposition when *receive* threads are many. Further, if the threads system is in a critical section when an interrupt is delivered, the appropriate *receive* thread can run only after the current thread exits the critical section and completes its time-slice. Because of this, performance depends on the time-slice length. Since fixed length time-slices do not allow the system to respond quickly to incoming traffic, there is a need for tailoring time-slice lengths to suit changing packet-arrival patterns [11].

Adaptive Time-slicing

A *receive* thread can be made to adapt to the kind of load it encounters at the receive buffer by altering its frequency of activation. It achieves this, in turn, by repeatedly adjusting the value of its time-slice. The idea is for a *receive* thread to be attentive enough to minimize packet loss, while keeping its use of the CPU low. If the packet arrival rate increases, a *receive* thread reduces its time-slice length. This enables it to check the receive buffer more often, to process any waiting packets. Conversely, when the packet arrival rate decreases, a *receive* thread correspondingly increases its time-slice length. Because this decreases its frequency of activation, context-switching overheads are reduced.

Adaptive time-slicing may be used in conjunction with interrupt-driven scheduling of the *receive* thread, assuming that the *receive* thread runs at a higher priority than other threads. Scheduling multiple *receive* threads on shared-memory multiprocessors remains a difficult problem, because matching threads to interrupts requires additional work.

4 Scheduling Receive Threads on Multiprocessors

In this section, we describe two possible schemes for scheduling *receive* threads on shared-memory multiprocessors. One algorithm is based on adaptive-time slicing and was originally designed for uniprocessor-based execution [11]. The other algorithm is based on *receive* threads which sleep for adjustable intervals of time before attending to their receive buffers. Unlike the study in [11], we have not implemented interrupt-driven receive schedules in the multiprocessor case, mainly because of higher overheads arising in matching threads to interrupts.

Slicing with Adaptive, Non-preemptive Receive (SANR)

On each of its activations, a *receive* thread attempts to estimate the size of its receive queue. On activation i , $i > 0$, it obtains an integer estimate X_i . For a given $n > 0$, if X_i and all $n - 1$ preceding 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 L_{max} , the *receive* thread must adjust its time slice length. The adjustment is done to help push the value of X_j into the interval $[L_{min}, L_{max}]$ 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.

Though this scheme optimizes the work done, and minimizes packet loss at each CLAM node, it has some definite disadvantages. If the system runs many *receive* threads, as is the case when several user-level protocols coexist in the same CLAM environment, managing time-slice lengths is difficult. For example, the packet arrival rate at one receive buffer for one protocol may be significantly higher than that at another buffer for another protocol. Decreasing the length of the global time-slice to accommodate the high input-rate buffer would unnecessarily increase processing overhead at the low input-rate buffer. This problem can be solved by using *receive*-thread based time-slices, at the expense of a small modification to the threads system. Timers will need to be set by the thread scheduler by examining a thread's state just before the thread is run.

Even if the system runs only a single *receive* thread, there is a difficulty. Because distinct Ariadne processes use different time-slice lengths for their threads, and because processes run ready threads from a common ready queue, coherence problems arise with time-slices. It is possible to overcome this limitation by allowing *receive* threads to define their time-slice lengths locally, immediately before they are given CPU control. Alternately, the time-slice length can be made global. Strictly speaking, neither scheme is necessary because – as we shall see shortly – multiprocessor scheduling of *receive* threads is simply and effectively accomplished through sleep-based scheduling.

Adaptive Timed-Sleep, Non-preemptive Receive (ATSNR)

In this scheme, *work* threads and *send* threads running at priorities p_w and p_s , respectively, are given fixed and equal amounts of CPU attention. The *receive* thread is made to run at a priority level $p_r > p_s = p_w$. When the *receive* thread is scheduled, it runs for as long as is necessary for it to empty out the contents of the receive buffer. When the buffer is empty, the *receive* thread determines an interval of time S for which it can go to sleep, before its next activation. The sleep functionality is enabled through provision of a special threads primitive based on the OS timer.

The *receive* thread adjusts its frequency of activation, depending upon the load it encounters at the receive buffer, using precisely the same strategy used in the SANR algorithm. There are several advantages to this adaptively-timed, sleep-based approach to scheduling receive functionality. Besides support for many *receive* threads, it readily supports both uniprocessor as well as multiprocessor operation, without the need for different algorithms for different architectures. Each *receive* thread maintains its own local value of S , meant for adapting its receptivity to the traffic it handles. On shared-memory multiprocessors, because this information is local to shared threads, there is no need for maintaining global information.

The provision of multiple, coexisting user-space protocols is fundamental to the CLAM communications architecture. By enabling the support of distinct *receive* threads for processing incoming traffic in distinct protocols, this scheduling algorithm provides functionality that restricts multiplexing functionality to lower layers, and is critical to our design goals. Because the threads system cannot allow interrupts when it is in a critical section, an awakening *receive* thread has to wait until a current thread exits a critical section and completes its time-slice, before it can run. The additional delay, requiring completion of the time-slice, is inherent to Ariadne's design. We have determined that the probability of this event's occurrence is low, and its effect on performance is almost negligible.

5 Experiments

In this section, we present experimental methodology and experimental results on the performance of the ATSNR scheduling algorithm. For the purposes of comparison, we also obtain performance results for the SANR scheduling algorithm. In each experiment, we focus on the ability of the receiver to process incoming packets while simultaneously doing application-related processing. To minimize packet loss and latency, a destination process's *receive* threads

must rapidly retrieve packets from the network and deliver it to its client. Further, these *receive* threads should consume CPU time sparingly, so as not to interfere with client processing.

5.1 Experimental Methodology

Each experiment involved a sending process on a SPARCstation 5 and a receiving process on a 4-processor SPARCstation 20, both communicating over an Ethernet. At the time of experimentation, both machines were observed to be only lightly loaded by other applications, and network interference due to traffic from other sources was negligible. Using the UDP protocol, the sending process placed packets on the network at regular intervals of time, ranging from 3 ms to 200 ms. Though packet injection rate at the sender was constant, the rate of packet arrival at the receiver was not constant due to network jitter. Datagram size was such that Ethernet frames were of maximum size (Ethernet MTU).

At the receiving process, client-related processing was accomplished by *work* threads, and CLAM's receive functionality for unreliable messaging was accomplished by a set of special threads. These included a *receive*, a *measure* and a *sync* thread. The *work*, *receive*, *measure* and *sync* threads were made to run at priority levels p_w , p_r , p_m , and p_s , respectively. Without loss of generality, the number of *work* threads (n_{work}) was set to one, for most experiments. Scheduling of *work* was achieved with the aid of Ariadne's time-slicing support. In the case of $n_{work} > 1$, all *work* threads ran at the same priority p_w .

The *measure* thread was made to sleep for a fixed length of time (10 secs) before waking up to do throughput and packet-loss measurements. Because of its short activation periods, this thread ran at a high priority. The *sync* thread was only used for synchronizing termination of all threads in the system. This enabled us to avoid the situation where one *work* thread terminated before other threads completed; in this way, we avoided biasing efficiency measurements by early terminations. Pseudo-code for each of these threads can be seen in Figures 10, 11 and 12.

The SANR algorithm

For the SANR approach, relative thread priorities were defined by

$$p_s > p_m > p_r = p_w$$

Both the *receive* threads and *work* threads were scheduled by time slicing. The time slice length varied between 10 ms and 500 ms. The granularity of the timer used for time slicing was 10 ms. Other control parameters included a simple history of the number of datagrams received during the last n activations of the *receive* thread, n itself, and the expansion and contraction factors for the time slice length. For the results shown in this paper, $n = 3$. That is, only the number of packets received during the three last activations of the *receive* thread were taken into account in adaptive scheduling. The contraction and expansion factor were set to $1/(1 + 2^3)$ and $1/(1 - 2^3)$, respectively.

The ATSNR algorithm

For the ATSNR approach, relative thread priorities were defined by

(a) SANR

```

ReceiveThread(){
    while( TRUE ){
        message = AllocateMessage();
        while( ReceiveMessage( message ) ==\
                                WLDBLOCK ){
            AdaptTimeSlice( s );
            yield();          /* SANR */
        }
        UpdateLoss();
        UpdateThrpt();
        Enqueue( message, RcvQ );
        if( finish )
            return;
    }
}

```

(b) ATSNR

```

ReceiveThread(){
    while( TRUE ){
        message = AllocateMessage();
        while( ReceiveMessage( message ) ==\
                                WLDBLOCK ){
            AdaptSleepTime( s );
            a_sleep( s );      /* ASTNR */
        }
        UpdateLoss();
        UpdateThrpt();
        Enqueue( message, RcvQ );
        if( finish )
            return;
    }
}

```

Figure 10: Pseudocode for the *receive* thread

(a)

```

WorkThread(){
    done = FALSE;
    while( TRUE ){
        CopyBuffer();
        if( !done && ( 10 sec elapsed\
                        since last measure ) ){
            UpdateStats( Work );
            if( TerminateCheck( Work ) ){
                signal( SyncSem );
                done = TRUE;
            }
        }
        if( finish )
            return;
    }
}

```

(b)

```

SendThread(){
    while( TRUE ){
        message = Dequeue( SndQ );
        SendMessage( socket, message );
        FreeMessage( message );
        if( finish )
            return;
    }
}

```

Figure 11: Pseudocode for *work* and *send* threads

<p>(a)</p> <pre> MeasureThread(){ done = FALSE; while(TRUE){ a_sleep(10 sec); if(!done){ UpdateStats(Throughput); UpdateStats(Loss); } if(TerminateCheck(Loss)){ done = TRUE; signal(SyncSem); } if(finish == TRUE) return; } } </pre>	<p>(b)</p> <pre> SyncThread(){ for(n = 0; n < nworks + 1; n++) wait(SyncSem); finish = TRUE; } </pre>
--	--

Figure 12: Pseudocode for *measure* and *sync* threads

$$p_s > p_m = p_r > p_w$$

so that *work* threads ran at the lowest priority. The *receive* thread was scheduled to run on expiry of the real-time OS timer (with 10 ms granularity), distinct from the execution timer used by Ariadne for time slicing. We used the sleep time of the *receive* thread as a control parameter. This time was permitted to vary between 10 ms and 500 ms. As in the case of the SANR algorithm, other control parameters included a simple history of the number of datagrams received during the last n activations of the *receive* thread, n itself, and the expansion and contraction factors for adjusting the sleep time. All parameter values were set as in the case of the SANR experiment. These values are proven, in that they yield a reasonable response time with a good granularity in the sleep time [11].

The measure of the amount of work done per second is obtained by sampling the number of *CopyBuffer* operations (see Figure 11) executed by a *work* thread during a period of roughly 10 seconds of real-time. Each of these samples is normalized and an average obtained over a number of samples ranging from 30 to 1800. Sampling terminates when a 90% confidence level for the estimate is obtained, with a relative error of less than 10%. When several *work* threads are used, the value reported is the average amount of work performed by all the *work* threads. If the number of *work* threads is $nwork$, and the average amount of work done by each thread is w , the total amount of work done is $nwork * w$.

Packet-loss is measured by tagging each datagram at the sender with a unique sequence number, and identifying gaps in the sequence of datagrams arriving at the receiver. The *receive* thread keeps track of the number of datagrams received and the number lost. The *measure* thread samples and resets such measurements every 10 seconds, computing averages over

Figure 13: Packet Loss vs. Packet Inter-arrival Time (uniprocess)

samples as described in the case of work measurements. The number n_{proc} of processes used in the multiprocessor experiments was always less than or equal to the number of physical processors available on the receiving host (in this case, four). With this, we can expect linear speedup in efficiency when the number of *work* threads n_{work} , and the number of processes n_{proc} , are simultaneously increased.

5.2 Experimental Results

In Figures 13 and 14 are shown the results of an experiment in which packet loss is estimated at a given destination for different rates of packet arrival. As expected, the uniprocess implementation of the ATSNR scheduling algorithm performs much the same as the SANR algorithm. Packet loss measurements for the multiprocessor version of the SANR algorithm are shown in Figure 14(a). For two or more processes, the graphs are highly irregular because the time between activations of the *receive* thread become highly random. The length of this interval fluctuates between very small values (i.e., the time for a thread's context switch) to some multiple of the time slice length, depending on the number of threads and processes in the system¹.

The SANR algorithm was actually developed for uniprocess implementations [11] and operates under the assumption that the *receive* thread's response interval is a relatively stable function of time slice length and thread density. Because this assumption is not true in the multiprocessor case, performance of the SANR algorithm is poor. Corresponding results for the multiprocess version of the ATSNR algorithm are shown in Figure 14(b). The increase in packet loss seen for increasing number of processes is likely due to locking/unlocking overheads arising in scheduling threads from a common shared queue. The same effect can also be observed in Figure 14(a). This source of overhead may also have some influence

¹In both the uniprocess and multiprocess implementations, OS context switches are an additional source of randomness in times between *receive* thread activations.

Figure 14: Packet Loss vs. Packet Inter-arrival Time (multiprocess)

on *receive* thread scheduling, causing an increase in packet loss as the number of processes increases.

In examining the results for the cases where $nproc = 2, nwork = 2$ and $nproc = 2, nwork = 1$, as shown in Figure 14(b), it can be deduced that having a dedicated process to run the *receive* thread does not help in reducing packet loss significantly. This is because the *receive* thread, which possesses a priority higher than that of the *work* threads, generally runs as soon as it awakens. It is unlikely that the threads system is in a critical section when the *receive* thread awakens – a situation that is sufficient to delay the *receive* thread's activation – because *work* threads typically do not use thread primitives.

The measured throughput for the uniprocess implementations of both the SANR and the ATSNR algorithms are shown in Figure 15. These performance numbers are consistent with the ones shown in Figure 13, indicating that both algorithms exhibit practically no difference in performance. Corresponding throughput numbers for the multiprocess versions of the SANR and ATSNR algorithms are shown in Figures 16(a) and 16(b), respectively. Once again, the behavior of both algorithms is almost the same regardless of whether the *receive* thread runs within a dedicated process or not. Performance remains the same even as the number of processes is increased. These results are also consistent with the results shown in Figures 14(a) and (b). Note that in the latter figures, even though both algorithms exhibit some difference in packet loss in the multiprocess case, there is no significant difference in the throughput graphs. This is so because the packet loss rate is only a small percent of the throughput achieved for the arrival rates considered.

In Figures 17 and 18 are shown the results of experiments in which we measured the amount of work done for different rates of packet arrival, for both scheduling algorithms. For both uniprocess implementations, the graphs in Figure 17 reveal that the SANR scheme performs marginally better than the ATSNR scheme. This is largely due to the performance of the threads system itself, and not because of the scheduling algorithms. In particular, the SANR algorithm requires less interrupt-related and scheduling overhead than the ATSNR, because of the manner

Figure 15: Throughput vs. Packet Inter-arrival Time (uniprocess)

Figure 16: Throughput vs. Packet Inter-arrival Time (multiprocess)

Figure 17: Efficiency vs. Packet Inter-arrival Time (uniprocess)

in which the Ariadne threads system is implemented. With the SANR algorithm, every time a running thread's time slice expires, a single interrupt is generated. With the ATSNR algorithm, scheduling of the *receive* thread is driven by an asynchronous interrupt, which is generated by the real time OS timer. When this interrupt occurs, the sleeping *receive* thread must be extracted from a global sleep queue and placed on a ready queue. Only after this can the *receive* thread run. Further, independently of how the *receive* thread is scheduled, a time slicing mechanism is used to schedule *work* threads. As a result, interrupt and scheduling overheads are slightly higher with the ATSNR algorithm.

The *average* amount of work² done per *work* thread, for the multiprocess implementations of both algorithms, can be seen in Figures 18(a) and (b). The SANR algorithm exhibits marginally better performance than the ATSNR algorithm, for the same reasons outlined in the uniprocess situation. The amount of work done decreases slightly with the number of processes for both schemes. This is due to locking and unlocking overheads.

The graphs in Figure 19 show how the *receive* thread adjusts its sleep time, as the packet arrival rate changes. These measurements are single point measures and not averages, observed at the end of our experiments. Nevertheless, they demonstrate the ability of the ATSNR algorithm to adapt to packet input. Note that in the multiprocess case, the ATSNR algorithm does not exhibit a significant performance fluctuation with the number of processes.

For completeness, we also show how the SANR algorithm adjusts its time slice length to packet arrival rate (see Figures 20(a) and (b)). The linear behavior seen in Figure 20(a) is similar to that shown in Figure 19(b). In viewing the SANR algorithm's time-slice adaptability in the multiprocess case, as shown in Figure 20(b), we observe that adaptability is hampered by the increased randomness in the times between *receive* thread activations. This results in an increase in packet loss, as shown in Figure 14(a).

²The *total* amount of work done by all threads is a product of the *average* work and the number of *work* threads.

Figure 18: Efficiency vs. Packet Inter-arrival Time (multiprocess)

Figure 19: ATSNR Sleep Time Response

Figure 20: SANR Time Slice Response

6 Distributed Interactive Simulation with CLAM

In a typical Distributed Interactive Simulation (DIS) application [19], the movement of a set of dynamic entities is simulated over a two-dimensional grid. Though entity movement occurs under well-defined constraints, changes in speed and direction are unpredictable. In general, a DIS application is distributed across processes on several hosts, with each process being directly responsible for simulating the movement of a distinct subset of entities, called *local* entities. Entities that are not local to a process are called *remote* entities. Each process periodically updates the positions of all entities on the grid. The management of local entity movement is simple, since the current state of each such entity is locally available. The management of remote entity movement is more complicated, requiring the use of a so-called *dead-reckoning* algorithm [36].

The dead-reckoning algorithm operates as follows. During simulation, each process simulates the movement of its remote entities using its best estimate of current state information for each of these entities. At regular time intervals, determined by a *heartbeat rate*, each process issues packets containing current state information for each of its local entities. These packets, called entity-state PDUs, are transmitted using a standard UDP multi-cast and meant for all hosts possibly interested in the state of this entity. Upon receipt of an entity state PDU, a receiving host compares its own estimate of the corresponding remote entity's state (i.e., location) with the data in the PDU. If the difference is greater than some acceptable threshold, the recently received PDU state information is used to update the estimate of entity state, since this information is more reliable than the process's ability to estimate future state.

In using CLAM's unreliable protocol to implement DIS, our intention is to show how packet-loss can be reduced while simultaneously increasing throughput in real applications. Since our focus is on DIS scalability and not on DIS results, we simplify implementation by ignoring collisions and other interactions. Further, without implementing a graphical interface, we make an attempt to include graphics-related DIS overheads through software loops. To

enable a comparison, we implement two versions of the same DIS application. One version is single-threaded and is based on standard UDP multi-cast, as is the case with standard DIS [19]. The other version is multi-threaded, and is based on CLAM's unreliable multi-cast.

The Standard DIS Implementation

In the standard implementation, each process runs in single-threaded mode, first updating the state of all entities, before issuing local-entity state PDUs. Next, the UDP socket is checked for incoming datagrams, which are retrieved and stored in an input queue. The PDUs just received are then processed. Finally, statistics are obtained on packet loss and throughput. These steps are repeated until a specified statistical accuracy is obtained. The number of incoming datagrams read during each iteration is limited to 20. Without such a limit, nodes may be deluged with incoming packets, thus preventing entities from being processed at the required frequency. A node which stalls in datagram receipt tends to exhibit sudden and rough entity movement over the grid.

The CLAM-based DIS Implementation

In using CLAM, the entity processing functionality and the receive functionality of the application are separated. This separation is enabled by threads, as follows. One thread repeatedly loops over all local and remote entities, updating their states and generating entity state PDUs for local entities that have changed their pattern of movement, or for those entities for which no PDU has been sent for a given period of time. Another thread, the *receive* thread, periodically checks the input socket and posts incoming datagrams in an input queue. This thread schedules itself using the ATSNR algorithm. To enable a fair comparison with the standard DIS implementation, the number of incoming datagrams that can be read by the *receive* thread on each activation is limited to 20. A third thread, scheduled to run every 20 seconds, measures packet loss and throughput, and a fourth thread processes input PDUs, to simulate the computation time necessary for grid display updates.

Experimental Methodology and Parameters

We ran the same DIS application using both the standard and the CLAM-based implementations. The heartbeat rate, i.e., the maximum desirable time between entity state PDU transmissions, was set to 500 ms. Five different hosts were used in the experiments, including four SPARCstation 5 machines and one SPARCstation 20, all on an isolated Ethernet. Entity movement pattern was kept constant, over each unit interval of time, with a probability of 0.01 of change in pattern. Whenever an entity changed its movement pattern, an entity state PDU was issued. Each state PDU datagram was 128 bytes long, and transmitted in a UDP multi-cast datagram. Each PDU was transmitted in a distinct datagram at the moment it was generated. During our experiments, the network and the hosts were observed to be only lightly loaded by other applications.

The number of entities simulated per host was made to vary from 20 to 1000. At a given receiving process, throughput and packet loss measurements were aggregated over all remote sources. All estimates were obtained using a 90 % confidence interval, with a relative error

Figure 21: Packet Loss vs. Number of Entities

of less than 10 % of the estimate. We also measured the interval of time between consecutive entity updates, called the *Entity Update Interval*. In using a multiprocessor host, the CLAM-based implementation distributes all simulated entities equally among all processes on the multiprocessor.

Experimental Results

The results of our experiments are presented in Figures 21, 22 and 23. These show the performance of the two implementations, as measured by each of the participating hosts. Host A was an 85MHz SPARCstation 5 with 32 MB of RAM, and hosts B, C, and D were all 70MHz SPARCstation 5 machines, each with 32 MB of RAM. Host E was a SPARCstation 20 with 4 50MHz processors and 128 MB of memory.

In Figure 21 it can be seen that both implementations suffer from increasing packet-loss, as the number of entities simulated – or computational work – is made to increase. In the CLAM-based approach, however, this packet loss is noticeably less. This is largely due to Host E, the multiprocessor. Both the single process and the multiprocess versions of CLAM exhibit low packet loss on the multiprocessor. Even when each host runs only a single *receive* thread, the high CPU availability at the multiprocessor allows for more computational work to be done. As can be seen in Figure 23, the Entity Update Interval in the CLAM-based implementation is only slightly higher than the corresponding interval in the standard DIS implementation. This difference, more noticeable as the number of entities increases, is due to the higher amount of entity-related traffic. In attempting to minimize packet-loss, CLAM's thread scheduler activates the *receive* thread more often, stealing CPU cycles from the entity-processing *work* threads. As a result of lower packet-loss, a CLAM-based DIS simulation can be expected to be more accurate and reliable than a standard UDP implementation. Observe that the best Entity Update Interval is given by the CLAM implementation running on Host E, i.e., the multiprocessor. Because all entities are distributed across E's multiple processes, each process – effectively

Figure 22: Throughput vs. Number of Entities

Figure 23: Entity Update Interval vs. Number of Entities

running on its own processor – has fewer entities to work with.

Datagram throughput, as shown in Figure 22, is also noticeably higher in the CLAM-based implementation when the number of entities is large and entity-related network traffic is high. When the number of entities is small, CLAM's thread scheduling overheads may account for the improved datagram throughput exhibited by the standard implementation. Note, however, that in the realistic situations where the number of entities simulated is very high (i.e., in the order of 800) the throughput given by the CLAM-based implementation is an order of magnitude higher than the standard implementation's throughput.

From the application viewpoint, a high throughput and low packet loss is an indication that a larger number of entities can be simulated, with fewer rough jumps in entity-state caused by loss of PDUs. The smaller entity Update Intervals obtained in the multiprocess CLAM implementation result in smoother movement of simulated entities. Of course, this is true only if the decrease in Entity Update Interval length is accompanied by reduction in packet loss and increased throughput.

7 Conclusions and Future Work

The work presented here is an attempt to provide simple and efficient threads based user-space protocols for transaction-oriented/message-oriented distributed computation. We envision a system, supported by CLAM, that will comprise a suite of distinct protocols which enable an application – running on a combination of uni- and multiprocessors – to use different protocols or multiple instances of the same protocol simultaneously. This is accomplished by a novel strategy combining a user-space threads system with protocol actions.

The empirical results presented here suggest that the ATSNR algorithm we propose is simpler and more convenient to implement than the SANR algorithm that was originally proposed for uniprocessor-based distributed applications, at least for unreliable traffic. The ATSNR algorithm also exhibits better scalability, for multiprocessor and multi-protocol applications. The algorithm exhibits an important limitation, however, due to the granularity of the timer used for the *receive* thread's sleep action. Use of the current 10 ms granularity is dictated by the OS, and is not a limitation of the threads system. This is larger than the latency of most LANs, and may cause problems when packet arrival rate is large enough to result in protocol-buffer overflow before a *receive* thread's reactivation. Though the *receive* thread may set its sleep time to under 10 ms, the time it takes to regain CPU control lies anywhere between the time requested and 10 ms. This time may be larger if an OS context switch occurs in between.

In implementing reliable transmission, we have found the *receive* thread to be more responsive when operating under the SANR algorithm than the ATSNR algorithm. We are currently examining ways of extending the SANR scheduling algorithm to the multiprocess and multi-protocol case. We have completed implementation and testing of a reliable, user-level transaction-oriented protocol, based on a slightly modified version of the SANR algorithm. The modifications involve the use of priorities, control transfer at key points, and an extension to support multiple protocols within a single process. A report on this study is in preparation. This work forms a basis for modular multi-protocol support in distributed computing applications. Our future research plans include the experimental development of reliable point-to-point and multicast modules for message-passing support, multi-protocol support within a single process,

and protocol extensions for shared-memory multiprocessors that enable both application and protocol processing to take advantage of parallelism.

References

- [1] V.Sunderam, G.Geist, J.Dongarra, and R.Manckek. The PVM Concurrent Computing System: Evolution, Experiences, and Trends. *Parallel Computing*, 20(4):531–545, April 1994.
- [2] R. Butler and E. Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20(4):547–564, April 1994.
- [3] *MPI: A Message-Passing Interface Standard*, June 1995.
- [4] Paul Pierce. The NX Message Passing Interface. *Parallel Computing*, 20(4):463–480, April 1994.
- [5] V. Bala et al. The IBM External User Interface for Scalable Parallel Systems. *Parallel Computing*, 20(4):463–480, April 1994.
- [6] Agerwala et al. SP2 System Architecture. *IBM Sytem Journal*, 34(2):152–184, 1995.
- [7] A. Ferrari and V. Sunderam. TPVM: Distributed Concurrent Computing with Lightweight Processes. In *4th IEEE International Symposium on High Performance Distributed Computing*, August 1995.
- [8] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing* (to appear).
- [9] H. Zhou and A. Geist. LPVM: A Step Towards Multithread PVM. Technical report, Oak Ridge National Laboratory, 1995.
- [10] E. Mascarenhas, F. Knop, and V. Rego. ParaSol: A Multi-threaded System for Parallel Simulation Based on Mobile Threads. In *Proceedings of the Winter Simulation Conference*, pages 690–697, 1995.
- [11] J. Gomez, V. Rego, and V. Sunderam. On Tailoring Thread Schedules in Protocol Design: Experimental Results. Submitted for publication.
- [12] D. Clark. An Alternative Protocol Implementation. Computer Laboratory, Cambridge, May 1982.
- [13] D. Clark. Modularity and Efficiency in Protocol Implementation. RFC-817, July 1982.
- [14] J.Postel. User Datagram Protocol. RFC-768, August 1980.
- [15] G.Wright and R.Stevens. *TCP/IP Illustrated*, volume 2, chapter 15 and 16. Addison Wesley, 1995.
- [16] E. Mascarenhas and V. Rego. Ariadne: Architecture of a Portable Threads System Supporting Thread Migration. *Software-Practice and Experience*, 26(3):327–357, March 1996.

- [17] E. Mascarenhas and V. Rego. Migrant Threads on Process Farms: Parallel Programming with Ariadne. Technical Report TR 95-081, Purdue University, 1995.
- [18] J. Gomez, V. Rego, and V. Sunderam. RTP3: a Multithreaded User-level Protocol for Message Passing Systems. Report in preparation.
- [19] IEEE. *Standard for Distributed Interactive Simulation – Application Protocols*, IEEE Standards Department 1995.
- [20] H. Eriksson. MBONE: The Multicast Backbone. *Communications of the ACM*, 37(8), August 1994.
- [21] J. Saltzer, D. Reed, and D. Clark. End-to-End Arguments in Systems Design. *ACM Transactions on Computer Systems*, 2(4), November 1984.
- [22] D. McAuley. *Protocol Design for High Speed Networks*. PhD thesis, University of Cambridge, 1989.
- [23] C. Thekkath, T. Nguyen, E. Moy, and E. Lazoswka. Implementing Network Protocols at User Level. *IEEE/ACM Transactions on Networking*, 1(5), October 1993.
- [24] J. Mogul, R. Rashid, and M. Accetta. The Packet Filter: An Efficient Mechanism for User Level Network Code. In *11th ACM Symposium in Operating System Principles*, pages 39–51, November 1987.
- [25] C. Maeda and B. Bershad. Protocol Service Decomposition for High-Performance Networking. In *SIGOPS*, pages 244–255. ACM, 1993.
- [26] D. Clark and D. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *SIGCOMM*, 1990.
- [27] J. Postel. Transmission Control Protocol. RFC-793, September 1981.
- [28] A. Edwards and S. Muir. Experiences Implementing a High Performance TCP in User Space. In *SIGCOMM*, 1995.
- [29] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Dalton, and C. Calamvokis. User-space Protocols Deliver High Performance to Applications on a Low Cost Gb/s LAN. In *SIGCOMM*, 1994.
- [30] D. Clark. The Structuring of Systems Using Upcalls. In *10th ACM Symposium in Operating System Principles*, pages 171–180, December 1985.
- [31] N. Hutchison and L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1), January 1991.
- [32] M. Bjorkman and P. Gunningberg. Locking Effects in Multiprocessor Implementation of Protocols. In *SIGCOMM*, 1993.

- [33] M. Kaiserwerth. The Parallel Protocol Engine. *IEEE/ACM Transactions on Networking*, 1(6), December 1993.
- [34] T. von Eicken. *Active Messages: an Efficient Communication Architecture for Multiprocessors*. PhD thesis, University of California Berkeley, 1993.
- [35] R. Pasquini and V. Rego. Distributed Services with SHOW. Report in preparation.
- [36] Kuo-Chi Lin. Dead Reckoning and Distributed Interactive Simulation. *DIS Systems in the Aerospace Environment/Critical Reviews*, CR58:16–36, 1995.