CLAM: Connection-less, Lightweight, and Multiway Communication Support for Distributed Computing*

Juan Carlos Gomez[1], Vernon Rego[1], and Vaidy Sunderam[2]

[1] Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, USA

[2] 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. There is a particularly strong motivation for the provision of scalable, multiway and connectionless transport for distributed computing, multimedia, and conferencing applications. This is also true of high speed networking, where it is beneficial to keep the OS kernel out of the critical path in communication. User-space protocol implementations may hold the key to optimal functionality and performance. We describe the Connectionless, Lightweight and Multiway (CLAM) communications system which provides efficient and scalable user-space support for distributed applications requiring multiple protocols. The system supports heterogeneous networked applications with irregular or asynchronous communication patterns and multimodal data. We focus on motivating and describing the CLAM architecture and present some experimental results that evaluate an specific protocol module inside this architecture.

1 Introduction

Distributed computing has undergone rapid and profound changes in recent years, both with respect to the paradigms used and also with respect to implementations and hardware. Active messages [31], Optimistic Active Messages [33], and Active Networks [30] have appeared as flexible and versatile alternatives to more traditional approaches like message passing, shared memory, and message driven computing. Communication hardware and software architectures have reduced workstation-cluster latencies from milliseconds to microseconds [32]. New applications requiring efficient many-to-many transfers, not supported by traditional protocols, have stimulated design and implementation of novel application-specific protocols [24]. With advances in software technology, distributed computing systems have begun to accommodate threads-based computing models [1, 9, 10, 21].

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

Traditional distributed computing software architectures that are based on in-kernel protocols cannot fully deliver to an application the low latencies and high bandwidths offered by current network technology. To overcome this limitation, new software and hardware architectures have been proposed [2, 15, 22, 32] that depart from the traditional approach by moving the kernel out of the critical path in communication. The idea is to implement protocols in user space, with user-level processes having direct access to network devices. As a result, such architectures have reduced message latencies from milliseconds to tens of microseconds, and in some cases, round-trip times in the order of microseconds have been reported [22]. There are several reasons for implementing communication protocols in user space: to keep the kernel simple [29] (i.e. Mach, Amoeba), to obtain functionality not available with traditional in-kernel protocols (e.g., many-to-many communication [24]), to increase scalability, and to make protocols more sensitive to application needs [6].

Partly motivated by concerns of performance, and partly pressured by available software technology, designers of distributed systems have begun to exploit lightweight threads as application-level interfaces [9, 10, 14, 20, 21]. This is also the result of a general acceptance that threads can enhance application development, support unpredictable data access patterns, mask communication and I/O latencies, enable dynamic computation schedules, facilitate fine-grained concurrency in shared memory multiprocessors, and potentially provide for load balancing. It must be recognized, however, that these are all constructs that exploit concurrency at the application level, and coexisting threads must still rely on the old communication framework (e.g., TCP/IP or reliable UDP) for message delivery. Most importantly, threads enable a smooth and efficient integration of communication protocols and application code, in keeping with the needs of current day applications and technology.

There is an alternative to the simple top-down use of threads in application-level constructs. This is a bottom-up view in which the communication framework relies entirely on threads for transaction-level communication, eliminating much of the protocol layering separating computing threads from communicating threads. In such a view, the communication subsystem is free to interact with the computation subsystem in any way that is beneficial to a distributed application. Further, locating all threads in user-space has the added advantage of placing the controllable part of the communication framework in user-space and offering the often undervalued benefits of application portability and system flexibility [6, 8]. Finally, the use of threads at both the application and the protocol level enables the efficient integration and scheduling of communication and user computation inside a single OS process. With such support, an application need not use distinct OS processes for communication and computation. With user-space protocols, there is no need for communication daemons.

The Connectionless, Lightweight, and Multiway environment (CLAM) [11, 12, 13] combines the advantages of threads, user-level protocols, and multi-protocol support to provide an efficient and scalable framework for integrating collaboration and computing on heterogeneous networks. CLAM improves efficiency by reducing expensive OS-level context switches, and by optimizing the message polling used to retrieve data from in-kernel protocols or network interfaces. CLAM augments the message passing environment with its user-level protocol modularity, providing support for reliable point-to-point, real time, and other types of traffic. Because of its scalability, multi-protocol and threads support, CLAM covers a potentially wider spectrum of distributed applications than most other runtime platforms.

Though a few systems provide support for multithreaded distributed computing at the application level [9, 14, 21], CLAM is the only platform that supports threads-based computing and communication in a fully integrated manner, including threads in protocol and message passing software. CLAM's protocol modules support a wider functionality than that provided by the traditional protocols used to support distributed computing (i.e. TCP/IP and UDP), including out-of-order delivery, remote thread activation, active messages, etc. Finally, CLAM was designed to be portable and easy to deploy, requiring no kernel coding or modification.

In the following section (Section 2), we describe the CLAM environment and, in Section 3, we present some experimental results comparing the performance of the reliable protocol module of CLAM to well-known message passing libraries. We conclude in Section 4 with a mention of future work.

2 The CLAM Environment

We expect future applications to exhibit a host of communication requirements, including low-latency, high-bandwidth, scalability, flexibility, multiple-protocol support, multimodal data, multiway transfers, and realtime and isochronous transport, among other things. CLAM is a connectionless, lightweight and multiway communications architecture designed to address the requirements listed above and, in particular, to support scalable, high-performance and collaborative applications that manage multimodal data (see Figure 1). To avoid heavy and cumbersome functionality, CLAM offers a plug-and-play methodology with its protocol suite [11]. It is layered on top of the Ariadne threads library [21] and the UDP protocol. It consists of a software layer that offers runtime support for global process management, and three native protocol modules: an unreliable module that provides efficient unreliable unicast and multicast, a reliable multicast module, and a reliable point-to-point module. Each protocol module is implemented with a specific set of communicating threads, depending on the functionality required. For example, the reliable point-to-point module requires three threads (i.e., a *receive*, a *send*, and a *timer* thread). The *send*, and *timer* threads handle asynchronous events, such as transmissions delayed by flow and congestion control algorithms, retransmissions, etc. The *receive* thread handles polling of the underlying communication layer. The unreliable module requires only a *receive* thread.

The CLAM system can be structured in one of two ways: a uniprocess model and a multiprocess model (see Figure 2). In the uniprocess model, all of the

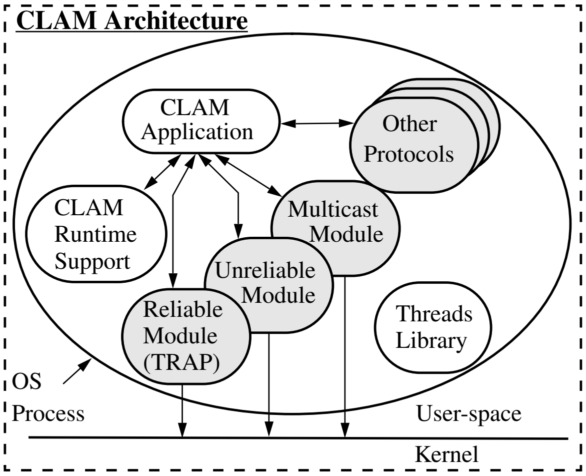

Fig. 1. Process View of CLAM

CLAM functionality is integrated with an application and runs within a single process at the OS level. The Ariadne scheduler is in charge of doling out CPU quanta between the different runnable threads. The multiprocess model may be used on shared memory multiprocessors, so that available parallelism can be exploited. Here, all threads reside in shared memory and may be scheduled to run in distinct OS-level processes (and thus, processors) to enable concurrency. A third model is currently under development, and it involves the use of kernel threads to exploit parallelism in shared memory multiprocessors. Benefits of this approach include reduced Operating System related costs, and the fact that no special software version has to be maintained for the multiprocessor architectures.

The reliable protocol module, that is, the Transaction-oriented Realiable and Point-to-point protocol (TRAP), is a key component of the CLAM protocol suite, designed to either augment or even replace TCP/IP in those situations where it cannot deliver the requisite performance or functionality. TRAP provides unreliable, reliable in-order, and reliable out-of-order at-most-once transmission. Further, each message may be delivered as a regular passive-message, an active-message, or a thread-activation. UDP datagrams may be lost, delivered out of order, or may get replicated. CLAM's reliable layer (TRAP) tackles these problems and provides a reliable, in-order, message oriented service on top of UDP or any other best-effort network protocol. TRAP is a sliding window protocol based on packets. This makes it highly suited to messaging applications. Positive acknowledgments, timed retransmission, and a modified version of the Fast Retransmit algorithm [17] are used to deal with packet loss. TRAP uses selective acknowledgments. Sequence numbers are used to detect packet replication and guarantee in-order delivery. TRAP computes retransmission timeouts using an adaptive algorithm proposed in [7]. Karn's algorithm, with binary-exponential

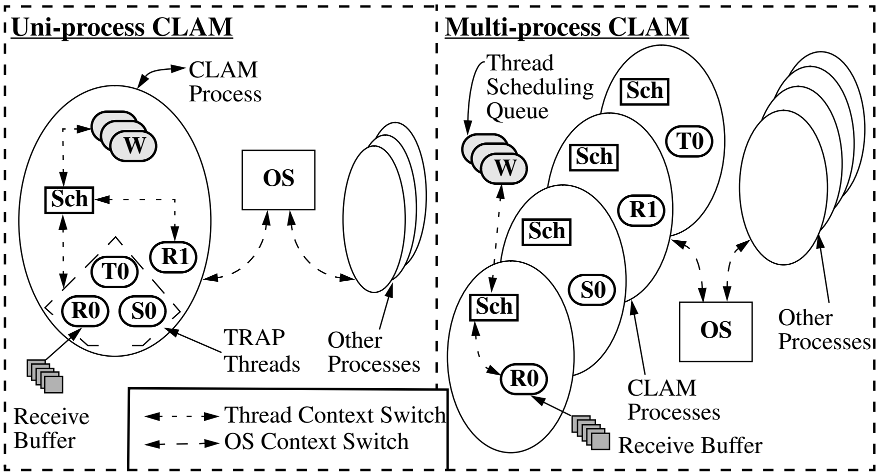

Fig. 2. Integrating CLAM to the OS

back-off, is used to modify the retransmission timeout on retransmissions [19]. TRAP also provides flow and congestion control. To improve the performance of our TRAP implementation, we used well-known optimization techniques such as the "Last-Received Cache" scheme proposed by Mogul [23], schemes suggested by Clark [4, 5], and the header prediction scheme [16].

The multicast module, currently under development, is intended to support reliable, totally-ordered multicast on a LAN or a set of locally interconnected LANs. By restricting the protocol to these environments, we expect to be able to achieve good performance through use of a simple multicast algorithm.

In a typical distributed computing application, a program explicitly specifies where sends and receives must occur in its execution sequence. This enables *send* and *receive* threads to run. Automatic polling strategies, based on compiler support, are likely to be inefficient. With threads support, however, a communications library may run *send* and *receive* threads with or without user-knowledge, enabling transparent message transmission and latency hiding. When they run, communication threads run at a higher priority than application threads. Each protocol module has a *receive* thread that polls the underlying protocol session for incoming messages. The scheduling of the *receive* thread is adaptively tuned to minimize polling overhead and datagram loss [11, 12, 13]. Adaptation is accomplished by adjusting the frequency of activation of the *receive* thread, based on a recent history of the datagram arrival rate.

As stated earlier, each protocol in CLAM supports active messages and remote thread activations. An active message carries an index that points to a function previously registered at the destination. Upon receipt of an active message, the protocol module invokes the function pointed-to, and passes it the message as a parameter. If the received message is a remote thread activation, the protocol module creates a new thread with the function pointed-to by the index, and passes the message to the new thread at creation time. The sender decides, through a flag set in the message header, whether the message is to be executed as a thread or as a function upon arrival at its destination.

Applications supported by CLAM consist of one or more threads that can access any of the protocol modules or any other in-kernel protocol provided by the host OS. Each of these protocol modules can have simultaneous sessions to many remote sites. Although the CLAM environment provides a broad range of communication support, applications on top of CLAM may still make use of in-kernel protocols like TCP/IP. This may be useful in applications where reliable stream oriented transfer is required.

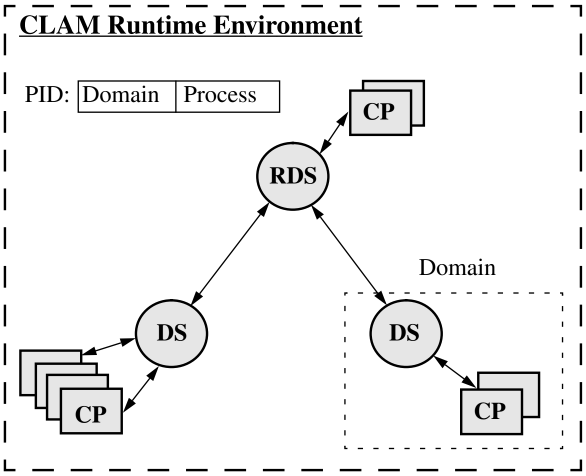

Fig. 3. Global View of CLAM

Figure 3 shows a global perspective of the CLAM environment. On top of the basic protocol modules, CLAM provides the additional runtime support necessary for the implementation of distributed applications. This support includes process management and message routing. CLAM's processes are structured in a three-level hierarchy, with three types of processes: Root Domain Server process (RDS), Regular Domain Servers processes (DS), and Computing Processes[3] (CP). This hierarchy divides CLAM's processes into domains. A CLAM domain is defined as a set of processes consisting of a Domain Server, with Computing Processes that depend on this Domain Server. Domain Servers are processes that enable user computation through a set of services (e.g., initial routing, global naming). The Root Domain Server is a special Domain Server process that is authorized to assign domain identifiers. Computing Processes enable user or application code to execute.

[3] What is referred to here as a process may, in fact, be a set of OS processes if the multi-process model is used.

All processes, including the Domain Servers, are addressed through a process identifier number (pid). The upper half of this pid defines the domain, and the lower half identifies a specific process in the given domain. Each Domain Server is named by a pid in which the process specific part is zero. It is responsible for a number of Computing Processes, all of which are located (preferably) on the same local area network. Pids are dynamically mapped to IP address and UDP port number tuples with the help of the Domain Servers.

3 Implementation Performance

We report the results of a set of experiments that we conducted to measure TRAP's performance. The intent is to determine how TRAP-based communication compares to popular TCP/IP- or UDP-based software systems for message-oriented computations on heterogeneous networks. Because these popular environments offer only single-threaded process executions, we made every effort to keep the comparison fair. In particular, we did not exploit the advantages of TRAP's support for multithreaded processing. We compare TRAP's communication performance to communication in version 1.4 of P4 [3], version 3.3.9 of PVM [28], and version 6.0 of LAM-MPI [25]. Our proposed message-passing interface (CMPI) is based on the TRAP multithreaded and user-level protocol library.

3.1 Experimental Methodology

All experiments presented here were conducted on homogeneous networked platforms; the message-passing libraries were *not* configured to do data conversions (PvmDataRaw option and LAM -O switch). Latency is measured from the viewpoint of the application. That is, we consider a message to be delivered *only when* it arrives at an application's buffer and is detected by the application. The $-nger$ switch was used during the LAM-MPI tests to turn off the Guaranteed Envelope Resources feature [26]. The experiments were performed on SPARCstation-5 machines, running SunOS Solaris 5.5, with cpu clock rates of 70 and 85 MHz.

We used two different types of environments in our experiments. The LAN experiments were conducted on a 10-Mbit/sec Ethernet and an ATM LAN composed of two workstations connected through a Fore Switch $200wg$ running version 4.0.0 of the switch software. The ATM interfaces used were the SAB-200E/UTP5 running at 155 Mbits/sec (OC3) over unshielded twisted pair. While conducting the experiments, it was ensured that "other" network and CPU load was low. The hosts used in these experiments ran NFS and shared the same file system.

In all cases, the experiments involving different libraries were run consecutively and not simultaneously. The interval of time between consecutive runs was kept low so as to complete all tests under similar conditions. Each message passing library was compiled using the *Makefile* contained in its distribution package.

All test programs, except for the ones using the LAM library, were compiled using *gcc* version 2.7.2 with the $-O$ optimization switch. Applications using the LAM-MPI library were compiled with the library-provided *hcc* command and the $-O$ optimization switch. The Ariadne threads library was compiled with the same *gcc* compiler, but with no optimizations. The TRAP library was compiled with the same version of *gcc* using the $-O$ switch.

To ensure that factors like initial ARP [27] cache miss and connection establishment did not bias measurements, at least one message was initially transmitted before timing-statistics were initiated. The function *gettimeofday* was used for all timing measurements. Since the overhead of invoking this function is about two orders of magnitude smaller than the minimum measurement obtained (i.e., μ-secs versus msecs), we ignored this effect in our measurements. The size of the maximum window used by TRAP in all the experiments was set to 32 packets, except where stated otherwise. For the other libraries, default send and receive protocol buffer sizes were used.

3.2 Round-trip Time

Low communication latency is a feature that can be crucial to the performance of distributed and collaborative software. To measure such latency we measure the amount of time it takes for one host to send a message of a fixed size to, and receive a reply from, another host. This test is commonly referred to as the ping-pong test.

The results of experiments conducted on Ethernet and ATM LANs are shown in Figure 4. We experimented with two variants of TRAP. With the first, labeled simply as TRAP, the user invokes message-passing primitives using application-buffers that are completely independent of the communication system. With the second variant, labeled TRAP-NC, the user puts application-level data directly into buffers initially allocated through communication-library primitives. As can be seen in Figure 4, the extra copy between application-space and communication-library space consumes a small amount of time that increases with message size. Observe that on an Ethernet, TRAP's message-latency is virtually the same as that offered by PVM-TCP (see Figure 4(a)). On an ATM LAN however, TRAP's latency is smaller than most of the other systems even in the presence of the additional copy operation (see Figure 4(b)). Further, both TRAP and TRAP-NC yield a performance that is significantly superior to that exhibited by the other UDP-based systems. The low latency achieved by TRAP is related to the use of a larger segment size at the transport protocol level. The average round-trip time estimate is based on 10,000 samples. The error, based on a 90% confidence interval, was found to be negligible relative to graph dimensions. The step seen in the PVM-UDP graphs (see Figures 4(a) and (b)) is a consequence of buffer handling overheads due to fragmentation. The spike seen in the LAM-TCP graph (see Figure 4(b)) recurred over several independent runs of the test program. We are unable to explain its cause.

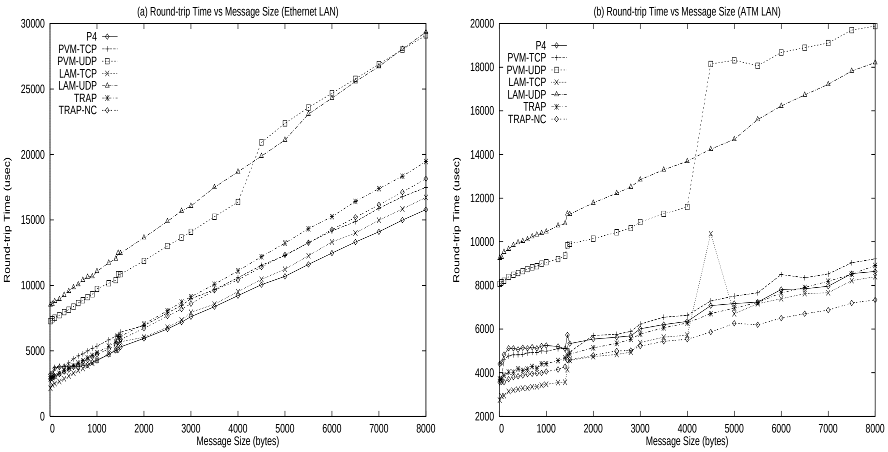

Fig. 4. Round-trip time (LAN)

3.3 Transaction-oriented Traffic

Distributed applications with irregular communication patterns require an asynchronous send and receive functionality. To measure TRAP's performance at handling asynchronous messages from different sources we designed the following test. A server process handles transactions that are sent by up to n clients. Each client process repeatedly sends the server a 100-byte message and awaits a reply to this message before sending the server a next message. The server's reply consists of a 1-byte message. The clients synchronize with one another during startup, with the help of the server. Client messages are sent to the server only after the synchronization, when all clients are known to be ready. Each client obtains an estimate of the average round-trip time (for its send-reply activity with the server) based on 10,000 messages. All nodes involved in this experiment were located on the same 10 Mbits/sec Ethernet LAN.

The server collects a round-trip time estimate from each client and computes a grand average, at the end of the test. We used up to 53 clients in this experiment: up to 43 clients on hosts with clock rates of 85 MHz and the rest on hosts with cpu clock rates of 70 MHz. The average number of transactions executed by the server per second, and the overall average round-trip time obtained by the server are shown graphed against the number of clients in Figures 5(a) and (b), respectively. In Figure 5(a), error-bars are shown in those situations (i.e., for TRAP, P4, PVM-TCP and LAM-TCP) where the size of the 90% confidence interval around the estimate is significant relative to graph dimensions.

In terms of the server's transaction processing rate, it is clear (see Figure 5(a)) that TRAP outperforms the other communication libraries we studied. P4 was not able to operate with more than 27 clients. This was the case even when the maximum number of file descriptors per process was 64. PVM-TCP

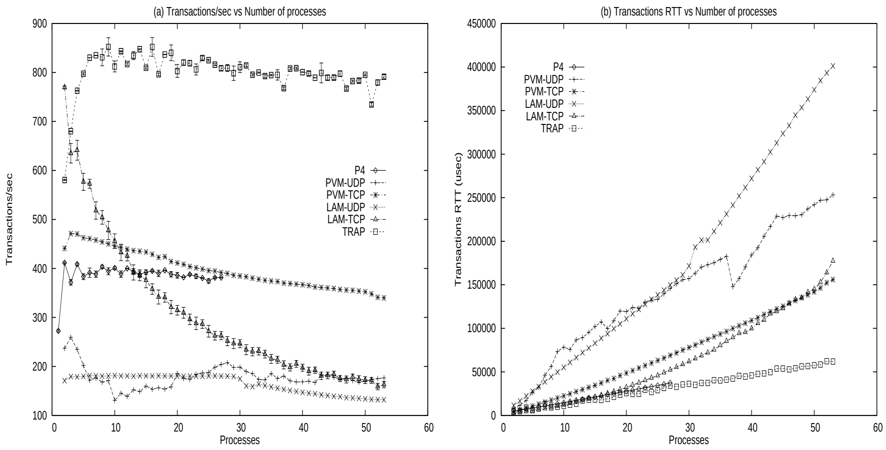

Fig. 5. Transaction Throughput and Delay (LAN)

and LAM-TCP were fully operational for all the values of n tested. This may be because PVM and LAM use a smaller number of file descriptors per session with a remote host. As shown in Figure 5(b), the experiment indicates that a TRAP-based server exhibits better scalability characteristics, with respect to the number of clients it can handle, than a server implemented with the other systems. Further, TRAP yields the smallest round-trip time, which implies the best support for rapid process interactions.

At first glance it may appear that TRAP's significant performance difference, as seen in Figure 5, is due to communication and computation overlap. A closer look at the situation, however, will show that TRAP's scalability is due to reasons other than user-threads based communication and computation overlap. In all the test programs, TRAP's transactions were found to be atomic; that is, the server sends a reply to each message received before the next arriving message is processed. This was due to the 10 ms time-slice length in the TRAP test program, and the small 100-byte messages. In further investigating causes for the performance difference we found that the TCP/IP-based libraries look for incoming messages using the *select* call. This call requires about 40 μ-secs to poll a single socket on a host with cpu clock rate of 70 MHz. Further, this time increases at the rate of about 10 μ-secs for *each* additional socket polled . In TRAP, sockets are polled with the help of non-blocking receives; the cost of each unsuccessful receive is roughly 60 μ-secs. Thus, with more than three clients, the TCP/IP libraries incur a polling cost that is larger than TRAP's polling cost. Further, the TCP/IP-based libraries incur an additional cost in that the *select* call they use must be followed by a *receive* or *read* at every socket that has input data available to read. In contrast, TRAP's non-blocking receives may directly retrieve such data.

3.4 Iterative Solution of a Linear System

To test TRAP's performance on a typical application, we implemented a distributed version of the Jacobi [18] iterative method for solving linear systems. The algorithm is explained as follows. A set of n slave processes are made to synchronize with one another through a master process. After this, each slave iteratively computes a piece of the solution vector. After the i-th iteration, each slave computes its contribution to the difference $|X_i - X_{i-1}|$ and sends the result of this computation to the master process. Each slave also multicasts (using point-to-point communication) its portion of the solution vector to the other slave processes, so that distributed iteration can continue, if necessary. The master process computes the total value of $|X_i - X_{i-1}|$ and uses this value to decide whether iteration must continue. If so, the slaves proceed with the next iteration and the process is repeated. Otherwise, the distributed computation is terminated.

The application was implemented in a uniform manner using four different systems: CLAM-based message passing, P4, PVM, and LAM. We used the versions of P4 and LAM indicated earlier, but used version 3.3.11 in the case of PVM. In each case, compilation was done with version 2.7.2 of *gcc* and optimization option $-O$, except for the case of LAM, where the library-provided *hcc* command was used. Each process ran on a distinct host. All hosts were interconnected through a 10 Mbit/sec Ethernet. We sampled execution times for 30 runs, and estimated an average with a 90% confidence interval. This was repeated for different numbers of slave processes and different vector sizes. The results are presented in Figure 6. Observe that, along with LAM, TRAP exhibits the best performance for this application.

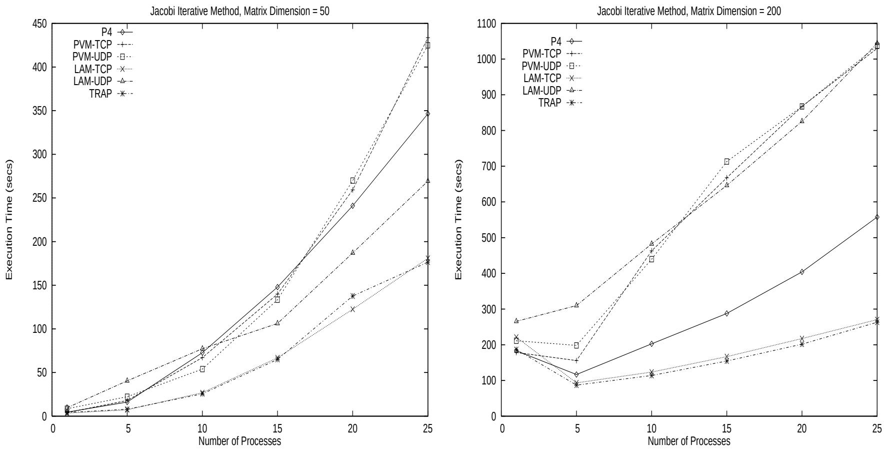

Fig. 6. Execution time vs Number of processes (Distributed Jacobi)

4 Conclusions and Future Directions

Based on our experiences with the design, implementation and testing of the TRAP protocol, we conclude that UDP-based user-level protocols are highly competitive in their ability to provide communication support for distributed computing. Indeed, in our experience, this support is almost as efficient as in-kernel protocol support, and even more valuable in terms of portability, scalability and flexibility. Further, integrating user-space protocol actions with user-space threads enables us to avoid problems inherent to TCP/IP-based and single-threaded message passing systems, including problems of deadlock with unrestricted two-way traffic and low throughput on ATM LANs. Threads also enable the hiding of programming complexities which may result from the use of single-threaded asynchronous calls in distributed computing.

Our experiments show that TRAP's polling mechanism exhibits scalability characteristics that are significantly better than the *select*-based polling mechanism used in most TCP/IP-based distributed computing systems. There are other important features of the TRAP implementation that were discussed, but were not experimented with in this paper. Such features may lead to improved performance, and include its fast connection-establishment scheme, its support for active-messages, and its support for threads-based concurrency on shared memory multiprocessors. Experiments involving these features are a subject of our ongoing work. The overhead of using threads also proved to be insignificant during our experiments. Using a profile tool we found this overhead to be of the order of 1% of the total execution time for a CLAM process that is actively sending or receiving messages.

Our plan is to implement a scalable and reliable multicast protocol, with low-level support for active-messages and remote thread activations. This protocol will enhance CLAM's TRAP module and its unreliable module, and will provide a versatile and efficient protocol-level communication platform for distributed computing and collaboration on heterogeneous networks. In addition, we intend to layer TRAP directly upon AAL5 to reduce latency and increase throughput in ATM LANs. Based on this support, our goal is to provide an interface for efficient collaborative and distributed computing with multimodal data.

References

1. R. Bhoedjang, T. Ruhl, and R. Hofman. Panda: A Portable Platform to Support Parallel Programming Languages. In *Symposium on Experiences with Distributed and Multiprocessor Systems IV*, pages 213–226, September 1993.
2. M. Blumrich, C. Dubnicki, E. Felten, K. Li, and M. Mesarina. Virtual-Memory-Mapped Network Interfaces. *IEEE Micro*, pages 21–28, February 1995.
3. R. Butler and E. Lusk. Monitors, Messages, and Clusters: The p4 Parallel Programming System. *Parallel Computing*, 20(4):547–564, April 1994.
4. D. Clark. Modularity and Efficiency in Protocol Implementation. RFC-817, July 1982.

5. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 27(6), June 1989.
6. D. Clark and D. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *ACM SIGCOMM*, 1990.
7. S. Edge. An Adaptive Timeout Algorithm for Retransmission across a Packet Switching Network. In *ACM SIGCOMM*, March 1983.
8. A. Edwards and S. Muir. Experiences Implementing a High Performance TCP in User Space. In *ACM SIGCOMM*, 1995.
9. A. Ferrari and V. Sunderam. Multiparadigm Distributed Computing with TPVM. Technical Report CSTR-951201, Department of Math and Computer Science, Emory University, 1995.
10. I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. Journal of Parallel and Distributed Computing. (To appear).
11. J. Gomez, V. Rego, and E. Mascarenas. The CLAM Approach To Multithreaded Communication on Shared-memory Multiprocessors: Design and Experiments. Technical Report TR-96-036, Purdue University, West Lafayette, IN 47907, 1996.
12. J. Gomez, V. Rego, and V. Sunderam. Efficient and Reliable Multithreaded Transport in User-space: The Design and Implementation of the TRAP Protocol. Technical Report TR-96-X, Purdue University, West Lafayette, IN 47907, 1996.
13. J. Gomez, V. Rego, and V. Sunderam. On Tailoring Thread Schedules in Protocol Design: Experimental Results. Technical Report TR-96-018, Purdue University, West Lafayette, IN 47907, 1996.
14. M. Haines, P. Mehrotra, and D. Cronk. Chant: Lightweight Threads in a Distributed Memory Environment. Technical report, ICASE, June 1995.
15. R. Horst. TNet: A Reliable System Area Network. *IEEE Micro*, pages 37–45, February 1995.
16. V. Jacobson. 4BSD Header Prediction. *ACM Computer Communications Review*, 20(2):13–15, 1990.
17. V. Jacobson. Modified TCP Congestion Avoidance Algorithm, April 1990.
18. J.Rice. *Matrix Computations And Mathematical Software*. McGraw-Hill, 1981.
19. P. Karn and C. Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. In *ACM SIGCOMM*, August 1987.
20. 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.
21. 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.
22. R. Minnich, D. Burns, and F. Hady. The Memory Integrated Network Interface. *IEEE Micro*, pages 11–20, February 1995.
23. J. Mogul. Network Locality at the Scale of Processes. In *ACM SIGCOMM*, September 1991.
24. T. Montgomery. *Design, Implementation, and Verification of the Reliable Multicast Protocol.* West Virginia University, Morgantown, West Virginia, 1994.
25. N. Nevin. The Performance of LAM 6.0 and MPICH 1.0.12 on a Workstation Cluster. Technical Report OSC-TR-1996-4, Ohio Supercomputer Center, March 1996.
26. Ohio Supercomputer Center. *MPI Primer/Developing with LAM*, December 1995.

27. D. Plummer. An Ethernet Address Resolution Protocol. RFC-826, November 1982.
28. 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.
29. A. Tanenbaum. A Comparison of Three Microkernels. *Journal of Supercomputing*, 9:7–22, 1995.
30. D. Tennenhouse and D. Wetherall. Active Networks. In *15th Symposium on Operating Systems Principles*, December 1995.
31. T. von Eicken. *Active Messages: an Efficient Communication Architecture for Multiprocessors*. PhD thesis, University of California Berkeley, 1993.
32. T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *15th Symposium on Operating Systems Principles*. ACM, 1995.
33. D. Wallach, W. Hsieh, K. Johnson, M. Kaashoek, and W. Weihl. Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation. In *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.