XTP as a Transport Protocol for Distributed Parallel Processing*

W. Timothy Strayer   Michael J. Lewis   Raymond E. Cline, Jr.

Distributed Computing Department
Sandia National Laboratories, California
{strayer,mlewis,rec}@ca.sandia.gov

Abstract

The Xpress Transfer Protocol (XTP) is a flexible transport layer protocol designed to provide efficient service without dictating the communication paradigm or the delivery characteristics that qualify the paradigm. XTP provides the tools to build communication services appropriate to the application. Current data delivery solutions for many popular cluster computing environments use TCP and UDP. We examine TCP, UDP, and XTP with respect to the communication characteristics typical of parallel applications. We perform measurements of end-to-end latency for several paradigms important to cluster computing. An implementation of XTP is shown to be comparable to TCP in end-to-end latency on preestablished connections, and does better for paradigms where connections must be constructed on the fly.

1 Introduction

The purpose of distributed parallel processing systems such as PVM, Mentat, Express, Linda, p4, and others, is to make a cluster of workstations appear to the user as a single multicomputer. These systems are typically software wrappers that facilitate the distribution of large problems among otherwise autonomous workstations, and that coordinate the communication necessary to achieve concurrent processing. This places a great deal of emphasis on the quality of the design and implementation of the data delivery system. In most cases, this delivery system is a network running the Internet protocol suite.

The cluster computing approach to parallel processing provides several advantages for both the research and production communities. Cluster computing offers a low cost alternative to expensive massively parallel processing (MPP) platforms, often allowing the utilization of existing resources and the gradual cumulation of systems. These systems can be expanded and improved through the purchase of additional resources and through the replacement of hardware components that limit performance. Further, public domain cluster computing software packages allow researchers to explore parallel processing options without substantial investments in new computing infrastructures.

However, a local area network is, in general, slower than the internal interconnection network of an MPP. More work must be done per communication event to amortize the communication cost over the computation. Consequently, the algorithms appropriate for implementation on a cluster of workstations are those with medium- and course-grain decompositions. Reducing latency and increasing throughput widens the category of algorithms appropriate for workstation clusters. One way to do this is to increase the capacity of the shared physical medium by replacing the ubiquitous 10 Mbit Ethernet with 100 Mbit LANs like FDDI, or by using switched media with aggregate

*This work was supported by Sandia Corporation under its Contract No. DE-AC04-94AL85000 with the United States Department of Energy, and CRADA No. 1136 between Sandia National Laboratories and AT&T Bell Labs.

bandwidth capacities on the order of a gigabit per second. This will help, but high latency and low throughput are also due to delays between the end user and the physical medium. The interface to the data delivery services, along with the protocols that comprise these delivery services, need attention.

The protocols used by distributed parallel processing systems should match the requirements of the applications they intend to support. Choosing a network protocol that provides excessive functionality can lead to unnecessary overhead that limits performance. Choosing one with inadequate functionality forces the missing services to be provided by the parallel processing system or application program. These makeshift extensions to existing protocols typically cannot be implemented as efficiently as protocols specifically designed to provide the appropriate services. Nonetheless, most cluster computing environments employ protocols from the Internet suite. PVM and Mentat, for example, both utilize their own reliable datagram protocols built above UDP, and PVM's alternate routing option uses TCP.

In this paper, we examine the Xpress Transfer Protocol (XTP) as the transport protocol for distributed parallel processing. We look at the communication characteristics of typical parallel applications, and use them to identify the performance and functionality requirements. We then discuss the features of XTP, TCP, and UDP that are relavent to providing efficient parallel processing. We run several different tests to measure end-to-end latency for all three protocols. Although protocol performance comparisons are largely implementation dependent, our results show that XTP's design and flexibility make it better suited than single paradigm solutions for meeting distributed parallel processing requirements.

2 Communication characteristics

Communication efficiency depends on several layers in the communication stack: the network protocols, the interface to the protocols, and the underlying network hardware. Shared media MAC layer protocols, like Ethernet, FDDI, and Token Ring, require a sender to gain exclusive access to the wire. Contention for the network increases with the number of processors onto which the problem is distributed, resulting in higher latency. The problem is exacerbated by parallel algorithms that progress in lock-step fashion and, in so doing, cause communication to take place simultaneously. Consequently, algorithms with medium- to course-grain decompositions are better able to amoritize the latency, and are therefore more appropriate for clusters over shared media LANs.

Switched media LANs (switched Ethernet, switched FDDI, and ATM), however, reduce or remove much of the contention, resulting in latency that is closer to the propogation time. Aggregate bandwidth, bounded in shared media LANs by the medium's data rate, becomes proportional to the number of channels in the switch. With these new technologies, cluster interconnects begin to resemble MPP interconnects in construction and performance. Even so, it has been shown that popular parallel processing systems, such as PVM, are still unable to fully exploit the performance of these high speed networks [6]. Emphasis must now be placed on optimizing the parallel processing systems and their transport protocols.

2.1 Parallel Algorithms

Several problem types have emerged as particularly appropriate for parallelization. Among these are iterative calculations over fields of values, such as stencil algorithms and multi-body dynamics simulations that use space-cell decompositions. A field of values is typically represented as a two- or three-dimensional matrix that is decomposed into cells and distributed among worker processes. The workers perform calculations, exchange results, and perform more calculations based on new information, continuing until some end condition is met. Typically, a cell communicates with the neighbors in it's "sphere of influence" to exchange the information necessary for the next round of calculations. A cell's immediate sphere includes eight neighbors in the 2 dimensional case, and 26 neighbors in the 3 dimensional case. A sphere of two cell-widths contains 24 and 124 neighbors for the 2D and 3D cases, respectively.

The amount of calculation by each processor in each iteration is proportional to the size of the cell, which is determined by the problem size and the number of processors onto which it is distributed. More processors implies more potential speedup, since the amount of computation per iteration per processor decreases with the size of the cell, leaving the communication phase as the limiting factor. In cluster computing, the communication costs can be orders of magnitude greater than for MPP machines. This argues for a large cell size to amortize communication costs over the longer calculation phases. But this limits speedup. The tradeoff can be shifted only by reducing the cost of the communication phase.

2.2 Communication topologies

Many algorithms have static communication topologies. For instance, in stencil algorithms, a vast majority of the communication takes place between processors that contain neighboring cells. Preestablishing connections between these communicants is certainly more efficient than connect-on-the-fly connection establishment, where connections are set up and torn down for each communication. However, some algorithms progress through several different phases during which the communication topologies are very different. In these algorithms, the penalty of preestablishing connections must be incurred not just once, but each time the program enters a phase that calls for a new topology.

Predetermining static communication topologies is impossible for some classes of applications. In the common bag-of-tasks paradigm, for instance, communicant pairs are not determined until runtime, when a task is assigned to a processor or set of processors. Using preestablished connections would require a fully general topology with connections between all possible pairs of processes. This scheme does not scale well, especially since connection oriented protocols often must incur the overhead of maintaining a separate communication record for each connection. Even worse, most protocol implementations typically treat an open connection as a limited resource, and impose a hard limit on the number that can be simultaneously maintained. Fully general topologies, and stencil algorithms that operate on cells that contain a large "sphere of influence," can push this hard limit. When the limit is reached, a connection must be closed each time a new one is required, resulting in performance no better than connect-on-the-fly connection establishment.

In connection oriented protocols, such as TCP, the cost of establishing a connection on the fly is often significant, especially when the amount of data being sent per communication is small, and when communication happens frequently. Connectionless protocols, such as UDP, avoid the overhead of explicit connection. However, UDP does not export enough functionality (e.g. reliability) to be suitable for distributed computing.

3 Protocol issues

Protocol implementors and protocol designers must do their parts to reduce latency. The implementor must limit the amount of processing required per packet. The designer must limit the number of packets per service operation and provide functionality that is appropriate to the higher layer protocol or application. Distributed parallel computing applications need several different types of communication paradigms. A protocol successful in distributed parallel processing must be flexible enough to support these paradigms efficiently. In this section, we discuss the features of TCP, UDP, and XTP, that are relevant to providing appropriate service.

3.1 TCP and UDP – the Internet suite

TCP and UDP over IP have served well as the foundations of the Internet, especially for applications that fit into one of the two paradigms offered. One-shot unreliable data delivery is provided by UDP. This is appropriate when the application or higher level protocol produces a limited amount of data sequentially unrelated to any other UDP datagram, and when recovery of lost data is unnecessary. Otherwise, some protocol above UDP must provide complete in-order delivery. TCP employs the internal mechanisms to track data delivery and to recover lost data. In this way, TCP provides the notion of a reliable stream of data in which order is relative and important. TCP does not provide message boundary markers.

TCP uses a three-way handshake to establish a connection [4], where initial sequence numbers for both directions of flow are exchanged. The three-way handshake guards against connection hazards caused by duplicate packets. Once open, the two communicants exchange data reliably. Graceful close of a connection requires each side to know that the other has received all data sent.

The Berkeley and System V Unix-based implementations of TCP [5, 7], through the socket and TLI interfaces, separate the connection establishment, data transfer, and connection closing into

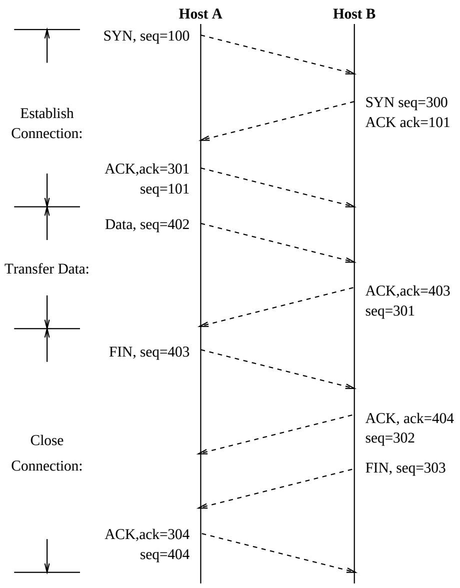

Figure 1: TCP data transfer

three stages. Since the TCP specification does not prevent the connection close handshake from piggybacking on the connection setup handshake, in theory only three segments are needed to reliably deliver TCP data [3, 2]. In practice, however, at least nine packets are used, as shown in Figure 1.

Because of its popularity and longevity, TCP is a target for implementation optimizations. Van Jacobson has observed that the predictability of future packets, based on previously received packets, is high enough to allow significant performance tuning of TCP (some of the observations made by Jacobson and others to tune TCP and UDP are chronicled in [8]). Further, fast-path studies of TCP [1] suggest that TCP packets can be parsed in about 200 instructions. Less can be done with issues that involve the design of the protocol. Once a protocol is widely available, the packet use and functional offerings are already decided and difficult to alter.

3.2 The Xpress Transfer Protocol

The Xpress Transfer Protocol (XTP) [11, 9] provides a transport layer service without dictating the communication paradigm or the delivery characteristics that qualify the paradigm. Communication paradigms are patterns of packet exchanges. Qualifiers on a paradigm, such as reliable or unreliable delivery, indicate how the endpoints (called *contexts* in XTP) act under various conditions, such as lost data. Central to the design of XTP is the notion that protocol service flexibility is essential for providing support for a wide range of applications. XTP allows considerable control over the pattern of packet exchanges and, through option bits carried in the packet header, provides control over the way these packets and their contents are handled.

There are three main packet types in XTP: FIRST packets, DATA packets, and CNTL packets. A FIRST packet is the first packet of an association. It is only used once if not lost, and carries addressing information and optional data. A DATA packet carries only data. A CNTL packet carries context state information, including flow and error control notifications.

Figure 2 shows several of the option bits of the header. The SREQ (status request bit) causes the destination context to respond with a CNTL packet that carries state information, including the current status of the data transfer. This bit allows the transmitter to request acknowledgements at times appropriate to the paradigm.

Several mode bits qualify the association between endpoints. The FASTNAK (fast negative acknowledgement) bit instructs the receiver to generate a CNTL packet in response to a DATA packet that is received out of sequence number order. The NOFLOW (no flow control) bit indicates that transmitted data cannot be throttled and that allocation information sent back to the transmitter

Bit	Description
NOCHECK	Turns off checksum over all but header
NOERR	Turns off error control
MULTI	Indicates multicast association
RES	Indicates conservative allocation policy
SORT	Indicates the presence of a priority value
NOFLOW	Turns off flow control
FASTNAK	Indicates aggressive error reporting
SREQ	Status request immediately
DREQ	Status request after data has been delivered
RCLOSE	Signals close of reader process
WCLOSE	Signals close of writer process
EOM	Marks end of message
END	Indicates end of association
BTAG	Indicates the presence of out-of-band data

Figure 2: XTP option bits

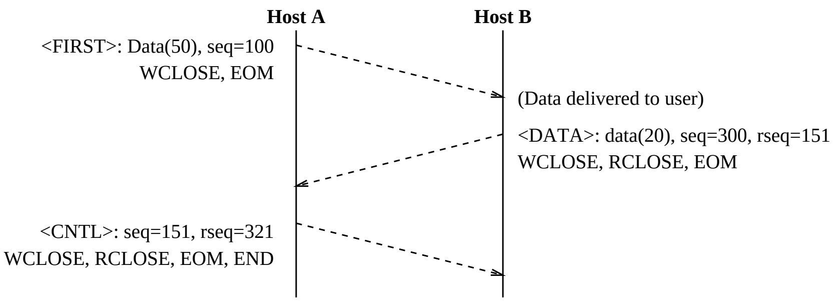

Figure 3: XTP reliable transaction

will be ignored. The NOERR (no error control) bit instructs the receiver to ignore gaps in the data stream.

Other option bits mark the data stream. The EOM (end of message) bit marks the end of a logical unit of data. The transmitter sets the WCLOSE (writer closed) bit to indicate that no new data will be sent. This marks the end of the data stream—only retransmitted data may be sent thereafter. The RCLOSE (reader closed) bit indicates that the context setting this bit will accept no more data packets on its incoming data stream. It is used to acknowledge the receipt of the WCLOSE bit, and implies that all data in the stream has been received. The END (end of association) bit indicates that the sender has closed the association and that the receiver should do so as well.

The FIRST packet's addressing information is used by the destination host to match the packet with the appropriate listening context. Since this FIRST packet may also carry data, overhead for unreliable data delivery is minimal. (Even in assured delivery paradigms, XTP allows the application to send data aggressively before it confirms association establishment.) A reliable datagram is constructed by sending data and an SREQ in the FIRST packet, then waiting until the CNTL packet is returned. A CNTL packet from the initiator to the receiver closes the association.

A transaction is built similarly, as illustrated in Figure 3. A FIRST packet with data (and as many DATA packets as required) serves as a request. The end of the request is marked by setting the EOM and WCLOSE bits in the last packet of the message. The response, in as many DATA packets as are necessary, implicitly acknowledges the request. If the transaction is not reliable, the last packet of the response carries the END bit and closes the association. If the transaction is reliable, the transmitter closes the connection with a control packet carrying the END bit.

Option bits, in concert with packet exchange patterns, are the mechanisms that allow XTP to provide services appropriate to application requirements.

4 Performance

The Transport Layer Interface (TLI) is designed to be System V Unix's standard interface to transport layer protocols. Mentat Inc., of Los Angeles, California[1], has developed a kernel implementation of XTP Version 3.7, called MXTP [10], with a TLI interface. To characterize the performance of XTP, we ran several different tests using MXTP and the implementations of TCP and UDP (also with TLI interfaces) that come standard with SunOS 5.3 (Solaris). The timing tests were run between two Sun SparcStation 10 Model 50 workstations containing Viking processor chips. The machines communicated through a dedicated channel on a 100 Mbps DEC FDDI Gigaswitch.

We tested five different protocols—XTP, TCP, UDP, R-UDP, and T-UDP—for three different communication paradigms. R-UDP is a simple approximation of what a reliable datagram protocol should do. We built R-UDP over UDP using a three-way positive acknowledgement packet exchange with two timers to catch lost packets. This three-way handshake ensures that the transmitter and the receiver are aware of each other's states as quickly as possible. If a two-way exchange is used to construct reliable delivery, the receiver would need a dally timer to catch duplicate data packets from the transmitter in case the acknowledgement got lost. The third packet, from the transmitter back to the receiver, assures the receiver that the transmitter knows the data was received. T-UDP is a similarly constructed transaction primitive built over UDP using two timers, one for the request and one for the response, and a single acknowledgement of the response. The response acts as an acknowledgement for the request. Since no packets happened to be dropped in any of the R-UDP or T-UDP tests, our measurements include the overhead of providing the mechanisms for recovery of lost data, but not the overhead of actually recovering any lost data.

We refer to the three communication paradigms as "Preestablished," "One-Shot," and "Transaction." Preestablished measures the one-way end-to-end latency of a message sent on an already established connection. One-shot measures the end-to-end one-way latency, including the time for connection setup and teardown, of a single message delivered reliably. This test is designed to estimate the communication performance that can be achieved when connections cannot be held open throughout the computation. Transaction measures the performance of the protocols performing with request-response behavior similar to RPC.

We took measurements of these paradigms for message sizes of 4, 16, 64, 256, 512, 1024, 2048, and 4096 bytes. We chose these message sizes to reflect the size of data that is appropriate for various parallel algorithms. Our sample size was twenty separate timings per data point. Each timing run consisted of 50 iterations of the experiment; if the experiments used roundtrip times to measure the latency, the total time was divided by 100, otherwise the total time was divided by 50. As a consequence, 1000 samples were taken for each point plotted. The results we show estimate the mean within plus or minus 1% at a confidence level of 95%. For each of the experiments we established a steady state before taking measurements to avoid various artifacts such as ARP lookups.

The results of the Preestablished experiments are shown in Figure 4. For each protocol we measured the time to send and fully receive the data, giving true end-to-end latency. Analysis of the data shows that there is an overhead component and a per-byte component to each protocol's latency, and that the per-byte component for each protocol is essentially the same. Fitting a line to the data shows the overhead as the y-intercept and the per-byte cost as the slope:

$$\begin{aligned}
\text{TCP Overhead} &= 0.650137 \text{ ms, Per-byte cost} = 0.344\ \mu\text{s}\\
\text{XTP Overhead} &= 0.649867 \text{ ms, Per-byte cost} = 0.344\ \mu\text{s}\\
\text{RUDP Overhead} &= 1.851929 \text{ ms, Per-byte cost} = 0.347\ \mu\text{s}\\
\text{UDP Overhead} &= 0.554301 \text{ ms, Per-byte cost} = 0.334\ \mu\text{s}
\end{aligned}$$

The difference between the UDP and R-UDP curves, about 1.3 ms, is the cost of adding two acknowledgement packets to make R-UDP reliable. This cost is constant since the acknowledgement packets are a fixed size. TCP and XTP add about a tenth of a millisecond to the overhead of a UDP packet. This is likely due to the state information that must be updated with the arrival of new packets, and an amortized cost of data acknowledgement.

[1] Mentat Inc. is unrelated to the University of Virginia's parallel processing system, also called Mentat.

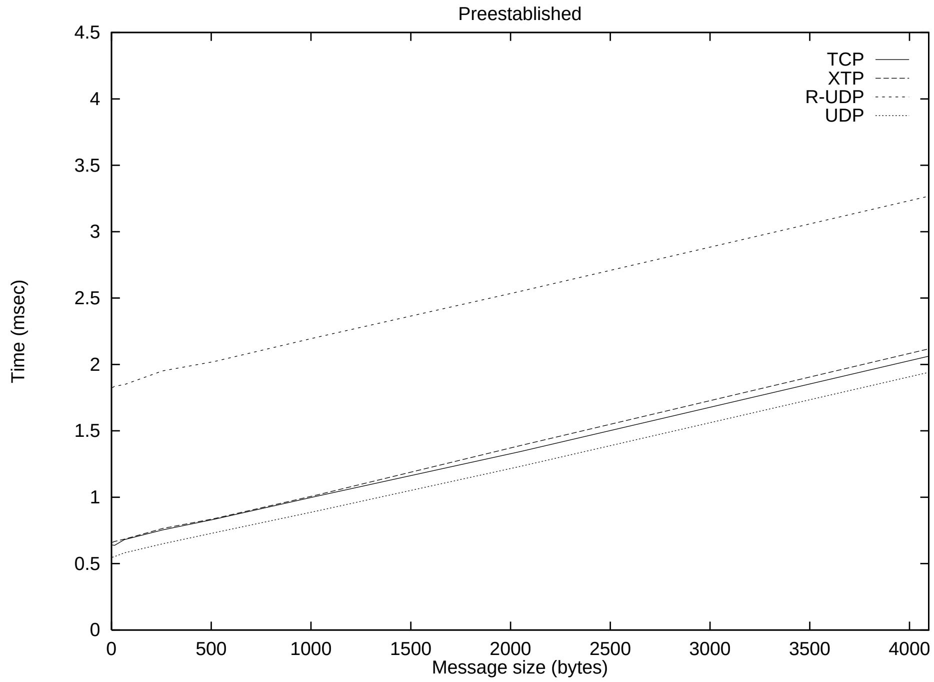

Figure 4: Time for one-way send on preestablished connection

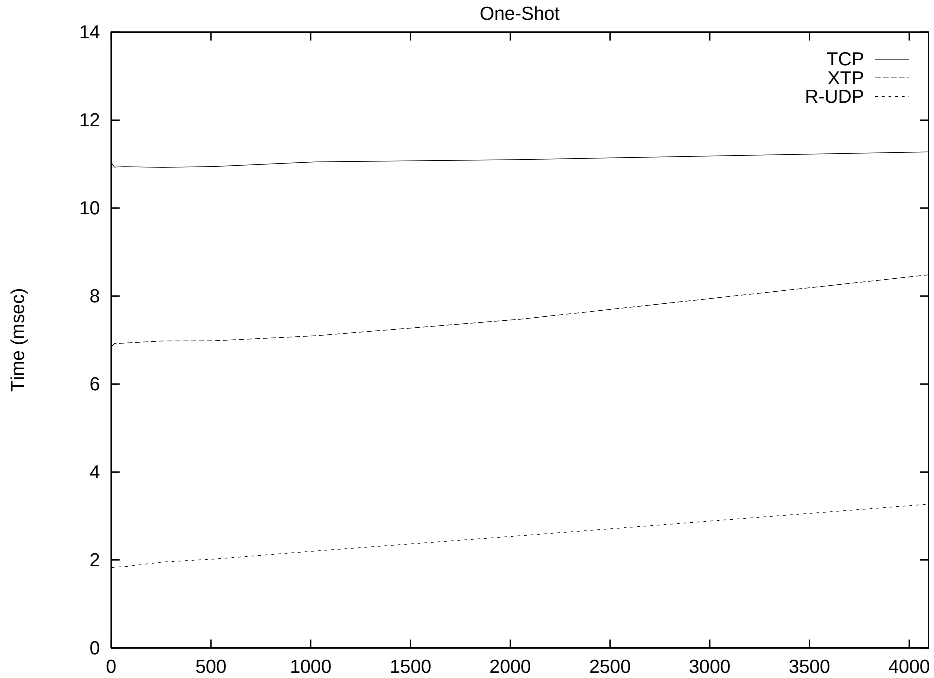

Figure 5: Time for one-way send, including connection setup and teardown

$\boxed{\text{Sender}}$
Fill in a call structure
Connect using this call structure
Send data
Send and orderly release
Block on a receive; when it fails (we expect it to),
    recieve the disconnect message

$\boxed{\text{Receiver}}$
Listen on the previously opened file descriptor
When a connect request arrives, open another file descriptor
Bind to it
Accept the connection on the new file descriptor
While there is data, receive
When the receive fails (we expect it to),
    receive the orderly release message
Send a disconnect message

Figure 6: Pseudo-code for TCP One-Shot

$\boxed{\text{Sender}}$
Fill in a call structure
Connect using this call structure and a data buffer
Send and orderly release
Block on a receive; when it fails (we expect it to),
    recieve the disconnect message

$\boxed{\text{Receiver}}$
(Same as in Figure 6)

Figure 7: Pseudo-code for XTP One-Shot

The One-Shot experiments, shown in Figure 5, expose the costs of connecting, sending, and disconnecting during a single communication with an arbitrary receiver. The open call, which allocates a file descriptor, is done only once; a call structure is filled in on the fly with the intended receiver. This file descriptor can then be reused for communication with another receiver.

We show only the reliable protocols in this graph. R-UDP does not have an explicit connect call, so the times shown by it's curve are the same as in the Preestablished graph. This protocol is the least expensive because it has the least to do. TCP and XTP both have explicit connect calls, but XTP allows the user to send data in the FIRST packet. This reduces the number of packets necessary, which is reflected in the timings. XTP should need only two packets and a dally timer (built into the protocol), or three packets without a dally timer, to reliably send one piece of data, but MXTP uses five: the FIRST packet and it's acknowledgement, then a three-way handshake to disconnect. This is more the fault of the TLI interface than of MXTP. TLI does not allow a connect, send, and disconnect to be combined in a single primitive. TCP does not couple the connect and send primitives; as shown in Figure 1 the connect, send, and disconnect phases are separate.

The psuedo-code for the TCP One-Shot data transfer is shown in Figure 6. We use the failure of the normal receive to detect a release message. This is necessary because TCP has no notion of a message bounary, so either the receiver must know the size of the message *a priori*, or the release message must be used to mark the end of transmission. Also notice that a disconnect message is used. This is normally used to abort a connection. By the time the disconnect function is called, the receiver has gotten all the outstanding data. Further, we could not call the connect function immediately after an orderly release because of some delay imposed by TCP to ensure all packets for this connection are received. Timings with the two-way orderly release showed times in the one-half

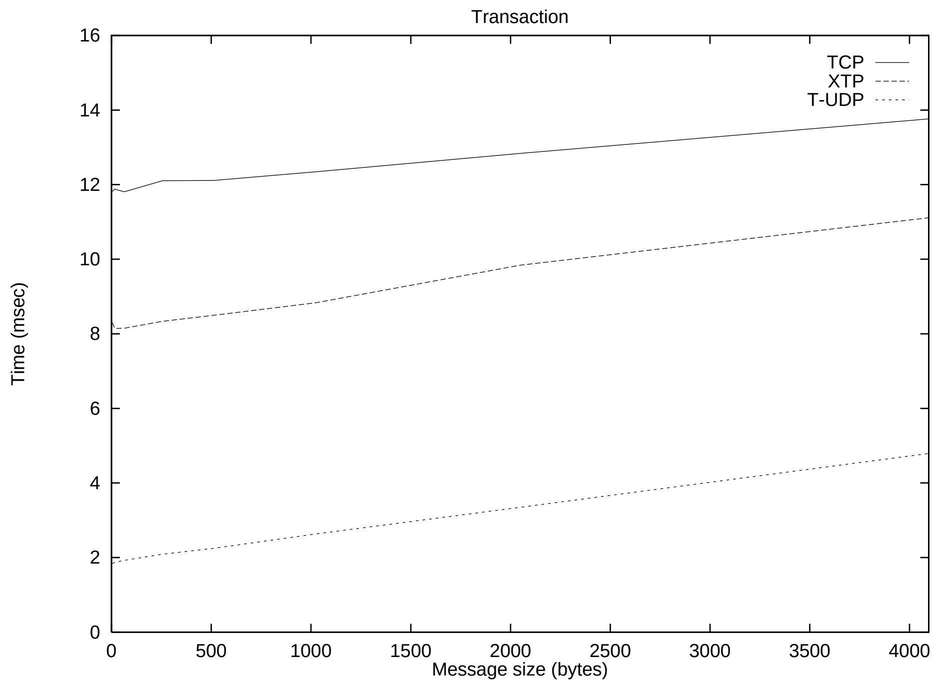

Figure 8: Time per transaction, including connection setup and teardown

to one second range. The psuedo-code for the sender for the XTP One-Shot data transfer is shown in Figure 7.

Figure 5 clearly shows that the XTP connect with data, done via a FIRST packet, helps reduce the one-way latency. According to [10], the number of packets used for this One-Shot send is five: the FIRST packet with DATA from sender to receiver, a CNTL packet in reply, and a three way CNTL packet exchange to close the connection. There are some efficiencies missing in the MXTP implementation, such as piggybacking the beginning of the connection disconnect on the FIRST packet, but the TLI interface does not expose that level of control.

The Transaction results are shown in Figure 8. As with the One-Shot results, these numbers reflect the cost of connecting, reliably executing a transaction, and disconnecting so that the file descriptor can be reused. The response messages in this experiment are the same size as the request messages; we acknowledge that this will rarely be the case, but there is no standard ratio of response size to request size. Also, as with the One-Shot results, reliable UDP is the least expensive because it uses only three packets and two timers for assuring reliable delivery. The overheads for XTP and UDP are roughly four and six times that of T-UDP, respectively; there are more packet exchanges taking place, and the protocol state machines for XTP and TCP are much more complex than that of T-UDP. MXTP's t_connect() uses the two call structure pointers to send the request and receive the response; the transaction, except for the disconnect, is done in one function call. After this a three-way exchange closes the connection. Again, more efficiency can be achieved by coupling some of the connection release indications with the packet exchange used to send the request and response, as shown in Figure 3.

5 Conclusions

Comparing the performance of various protocols, even on the same platform with the same interface, must be done cautiously. So much depends on implementation decisions and programming skill that the inherent advantages of one protocol over another can be lost. Nonetheless, we endeavor to examine TCP, UDP, and XTP for several paradigms important for cluster computing. Implementation differences aside, a protocol must be flexible and efficient in order to serve distributed parallel processing environments. High latency is extremely damaging to parallel processing performance. Well designed and well implemented protocols are essential in any environment.

XTP is designed to handle common virtual circuits as well as datagrams. It provides reliability as an orthogonal attribute; with the packet types and options bits in XTP one can construct reliable datagrams, unreliable virtual circuits, or any number of combinations of attributes and paradigms. In theory XTP provides the tools for constructing communication primitives appropriate for many types of environments. We obtained a commercial implementation of XTP from Mentat, Inc. and instrumented several experiments based on paradigms that arise in parallel processing in order to compare this XTP with the native TCP and UDP.

We make several observations based on the results of these experiments and our experience running them. XTP compares favorably with TCP for one-way end-to-end latency on a preestablished connection. For the situations where a paradigm such as reliable datagrams or reliable transactions had to be constructed, TCP performed worse than XTP. If we assume that the results from the latency tests over preestablished connections suggest that the protocols are comparably implemented, XTP's advantage must be due to reduced packet exchanges. For the one-shot reliable datagram and the reliable transaction, XTP carries data in the FIRST packet while TCP requires the three-way connect handshake before data can be exchanged. MXTP exposes these aspects of XTP through its t_connect() call. Both TCP and XTP, through the TLI interface, required a fairly extensive sequence of calls for reliably closing the connection, and even then the orderly release for TCP caused delays of as much as a second before releasing the resources. Our experience in working with MXTP suggests that it either can not or does not expose the full flexibility of XTP to the user. Some of this is design decision, some is due to the TLI interface. At any rate, more economy could have been gained from XTP in both the reliable datagram and reliable transaction.

In spite of the advantages of XTP over TCP in the experiments where connections where established and released on the fly, the clear winner in reducing latency is a preestablished connection. When the communication topology is static enough to allow this, a preestablished connection provides the lowest latency of reliable protocols. For connect-on-the-fly communication, it seems that our R-UDP and T-UDP solutions are faster than TCP or XTP; we caution that these sample protocols hold virtually no state information or do other protocol processing. Still, these constructed protocols show what is possible when a minimum number of packets are used to construct a particular communication paradigm.

Acknowledgments

The authors wish to thank John Fenton, of Mentat, Inc., for the valuable technical assistance and insight, and Bruce Carneal, also of Mentat, Inc., for providing MXTP for evaluation. We also thank Tuesday Armijo, Tim Cody, and Alden Jackson of Sandia.

References

[1] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[2] D. E. Comer. *Internetworking with TCP/IP, Vol: I, Principles, Protocols, and Architecture: Second Edition*. Prentice Hall, Edgewood Cliffs, New Jersey, 1991.

[3] W. A. Doeringer, D. Dykeman, M. Kaiserswerth, B. W. Meister, and H. Rudin. A Survey of Light-Weight Transport Protocols for High-Speed Networks. *IEEE Transactions on Communications*, 38(11):2025–2039, November 1990.

[4] J. Postel (editor). Transmission Control Protocol, rfc-793, September 1981.

[5] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, Massachusetts, 1989.

[6] M. J. Lewis and R. E. Cline Jr. PVM Communication Performance in Switched FDDI Heterogeneous Distributed Computing Environments. *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 13–19, October 1993.

[7] M. Padovano. *Networking Applications on UNIX System V Release 4*. Prentice Hall, Edgewood Cliffs, New Jersey, 1993.

[8] C. Partridge and S. Pink. A Faster UDP. *IEEE/ACM Transactions on Networking*, 1(4):429–440, August 1993.

[9] W. T. Strayer, B. J. Dempsey, and A. C. Weaver. *XTP: The Xpress Transfer Protocol.* Addison-Wesley, Reading, Massachusetts, 1993.

[10] Mentat XTP Internals Manual, Mentat, Inc., Los Angeles, California, 1994.

[11] XTP Protocol Definition, Revision 3.6. Technical Report PEI 92-10, Protocol Engines, Inc., January 1992.