SCTP versus TCP for MPI

Humaira Kamal
Department of Computer Science
University of British Columbia
Vancouver, BC
kamal@cs.ubc.ca

Brad Penoff
Department of Computer Science
University of British Columbia
Vancouver, BC
penoff@cs.ubc.ca

Alan Wagner
Department of Computer Science
University of British Columbia
Vancouver, BC
wagner@cs.ubc.ca

Keywords
Middleware, SCTP, TCP, MPI, Wide Area Network, Protocol, Message-Passing

ABSTRACT
SCTP (Stream Control Transmission Protocol) is a recently standardized transport level protocol with several features that better support the communication requirements of parallel applications; these features are not present in traditional TCP (Transmission Control Protocol). These features make SCTP a good candidate as a transport level protocol for MPI (Message Passing Interface). MPI is a message passing middleware that is widely used to parallelize scientific and compute intensive applications. TCP is often used as the transport protocol for MPI in both local area and wide area networks. Prior to this work, SCTP has not been used for MPI.

We compared and evaluated the benefits of using SCTP instead of TCP as the underlying transport protocol for MPI. We re-designed LAM-MPI, a public domain version of MPI, to use SCTP. We describe the advantages and disadvantages of using SCTP, the necessary modifications to the MPI middleware to use SCTP, and the performance of SCTP as compared to the stock implementation that uses TCP.

1. INTRODUCTION
TCP is widely used as the underlying transport protocol in the implementation of parallel programs that use MPI. It was available in the first public domain versions of MPI (LAM [10] and MPICH [29]) for the execution of programs in local area network environments. More recently the use of MPI with TCP has been extended to computing grids [17], wide area networks, the Internet and meta-computing environments that link together diverse, geographically distributed, computing resources. The main advantage to using an IP-based protocol (i.e., TCP/UDP) for MPI is portability and ease with which it can be used to execute MPI programs in diverse network environments.

(c) ACM, (2005). This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version will appear in SC|05 November 12-18, 2005, Seattle, Washington, USA http://doi.acm.org/10.1145/nnnnnn.nnnnnn

One well-known problem with using TCP or UDP for MPI is the large latencies and difficulty in exploiting all of the available bandwidth. Although applications sensitive to latency suffer when run over TCP or UDP, there are latency tolerant programs such as those that are embarrassingly parallel, or almost so, that can use an IP-based transport protocol to execute in environments like the Internet. In addition, the dynamics of TCP is an active area of research where there is interest in better models [5] and tools for instrumenting and tuning TCP connections [19]. As well, TCP itself continues to evolve, especially for high performance links, with research into new variants like TCP Vegas [9, 21]. Finally, latency hiding techniques and exploiting trade-offs between bandwidth and latency can further expand the range of MPI applications that may be suitable to execute over IP in both local and wide area networks. In the end, the ability for MPI programs to execute unchanged in almost any environment is a strong motivation for continued research in IP-based transport protocol support for MPI.

TCP and UDP have been the two main IP protocols available for wide-spread use in IP networks. Recently however, a new transport protocol called SCTP (Stream Control Transmission Protocol) has been standardized [27]. SCTP is message oriented like UDP but has TCP-like connection management, congestion and flow control mechanisms. In SCTP, there is an ability to define streams that allow multiple independent message subflows inside a single association. This eliminates the head-of-line blocking that can occur in TCP-based middleware for MPI. In addition, SCTP associations and streams closely match the message-ordering semantics of MPI when messages with the same context, tag and source are used to define a stream (tag) within an association (source).

SCTP includes several other mechanisms that make it an attractive target for MPI in open network environments where secure connection management and congestion control are important. It makes it possible to offload some MPI middleware functionality onto a standardized protocol that will hopefully become universally available. Although new, SCTP is currently available for all major operating systems and is part of the standard Linux kernel distribution.

The contribution of our work is the design and evaluation of using SCTP for MPI. We have re-designed the LAM-MPI middleware to take advantage of the features of SCTP. We describe the overall design, the advantages we found as

well as the disadvantages and limitations of using SCTP. One novel advantage to using SCTP that we investigated is the elimination of the head-of-line blocking present in TCP-based MPI middleware. We evaluated our LAM-SCTP module and report the results of several experiments using real world applications as well as standard benchmark programs and compare the performance with LAM-TCP. Our experiments show that the SCTP module outperforms TCP under loss, especially when latency tolerant applications are used. The advantages of using SCTP are evident even in the case of a single-stream and single path between endpoints and the benefits are greater when SCTP's multistreaming capability is utilized. The performance of SCTP is comparable to that of TCP for the benchmark programs and is better in the case of latency tolerant applications. A second indirect contribution of our work is with respect to SCTP. Our MPI middleware makes very aggressive use of SCTP. In using the NAS benchmarks [6] along with programs of our own design, we were able to uncover problems in the FreeBSD implementation of the protocol that led to improvements in the stack by the SCTP developers.

Overall, the use of SCTP makes for a more resilient implementation of MPI that avoids many of the problems present in using TCP, especially in open wide area network environments such as the Internet. Of course, it doesn't eliminate the performance issues of operating in that environment nor does it eliminate the need for tuning connections. Many of the same issues remain, however, as mentioned this is an active area of research in the networking community where numerous variants of TCP have been proposed. Because of the similarity between the two, SCTP will be able to take advantage of improvements to TCP. Although the advantages of SCTP have been investigated in other contexts, this is the first use of it for MPI. It is an attractive replacement for TCP in an open, wide-area network environment, and adds to the set of IP transport protocols that can be used for MPI. The release of Open MPI [11], and its ability to mix and match transport mechanisms, is an ideal target of our work where an SCTP module can further extend MPI across networks that require a robust TCP-like connection.

2. OVERVIEW

2.1 SCTP

SCTP is a general purpose unicast transport protocol for IP network data communications, which has been recently standardized by the IETF [27]. It was initially introduced as a means to transport telephony signaling messages in commercial systems, but has since evolved for more general use to satisfy the needs of applications that require a message-oriented protocol with all the necessary TCP-like mechanisms. SCTP provides sequencing, flow control, reliability and full-duplex data transfer like TCP, however, it provides an enhanced set of capabilities not in TCP that make applications less susceptible to loss.

Like UDP, SCTP is message oriented and supports the framing of application data. But like TCP, SCTP is session-oriented and communicates by establishing an association between two endpoints. Unlike TCP, in SCTP it is possible to have multiple logical streams within an association where each is an independent stream of messages which are delivered in-order.

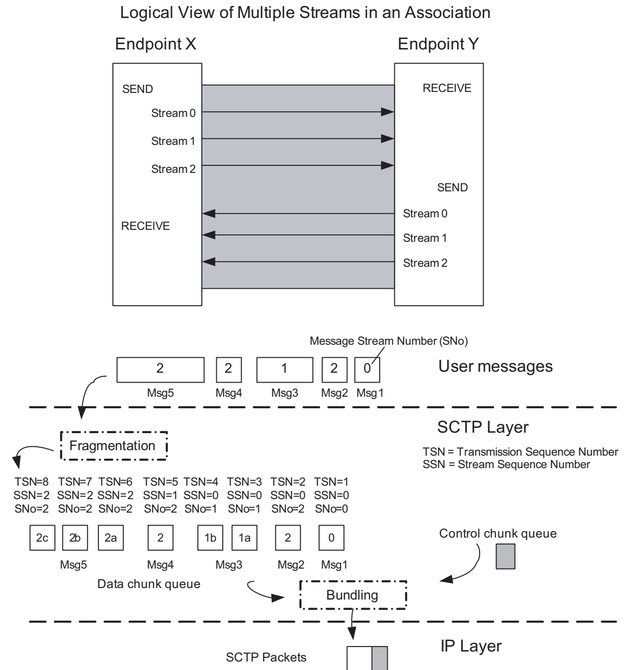

Figure 1: Single Association with Multiple Streams

Figure 1 shows an association between two endpoints with three streams identified as 0, 1 and 2. User level messages are broken into data chunks (SCTP sequences data chunks and not bytes as in TCP) and each chunk is assigned a `SNo`, `SSN` and `TSN`. The `SNo` identifies the stream, `SSN` is used to sequence messages within the stream, and `TSN` is used to sequence the data chunks. Together `SNo`, `SSN`, and `TSN` are used to assemble messages and guarantee the ordered delivery of messages within a stream but not between streams. For example, receiver `Y` in Figure 1 can deliver `Msg2` before `Msg1` should it happen to arrive at `Y` before `Msg1`.

In contrast, when a TCP source sends independent messages to the same receiver at the same time, it has to open multiple independent TCP connections. It is possible to have multiple streams by having parallel TCP connections and parallel connections can also improve throughput in congested and uncongested links. However, in a congested network parallel connections claim more than their fair share of the bandwidth, thereby affecting the cross-traffic. One approach to making parallel connections TCP-friendly is to couple them all to a single connection [12]. SCTP does precisely this by ensuring that all the streams within a single SCTP association share a common set of congestion control parameters. It obtains the benefits of parallel TCP connections while keeping the protocol TCP-friendly [24].

Another difference between SCTP and TCP is that endpoints in SCTP are multihomed and can be bound to multiple IP addresses (i.e., interfaces). If a peer is multihomed, then an SCTP endpoint will select one of the peer's destination addresses as a primary address and all other addresses of the peer become alternate addresses. During normal operation, all data is sent to the primary address, with the exception of retransmissions, for which one of the active alternate addresses is selected. Congestion control variables

are path specific. When the primary destination address of an association is determined to be unreachable, the multihoming feature can transparently switch data transmission to an alternate address. SCTP, currently, does not support simultaneous transfer of data across interfaces, but this will likely change in future. Researchers at the University of Delaware are investigating the use of SCTP's multihoming feature to provide simultaneous transfer of data between two endpoints through two or more end-to-end paths [14, 13], and this functionality may become part of the SCTP protocol.

SCTP supports both one-to-one style and one-to-many style sockets [25]. One-to-one socket corresponds to a single SCTP association and was developed to allow porting of existing TCP applications to SCTP with little effort. In the one-to-many style a single socket can communicate with multiple SCTP associations similar to a UDP socket that can receive datagrams from different UDP endpoints.

2.2 MPI middleware

MPI has a rich variety of message passing routines. These include `MPI_Send` and `MPI_Recv` along with various combinations such as blocking, nonblocking, synchronous, asynchronous, buffered, unbuffered versions of these calls. The matching of an `MPI_Send` to its `MPI_Recv` is based on three values inside the message envelope: (i) context, (ii) source/destination rank and (iii) tag (see Figure 2). Context identifies a set of processes that can communicate with each other. Within a context, each process has a unique identification called rank. A user can specify the type of information being carried by a message by assigning a tag to it. MPI also allows the source and/or tag of a message to be a wildcard in `MPI_Recv` request. For example if `MPI_ANY_SOURCE` is used then that specifies that the process is ready to receive messages from any sending process. Similarly if `MPI_ANY_TAG` is specified, a message arriving with any tag would be received.

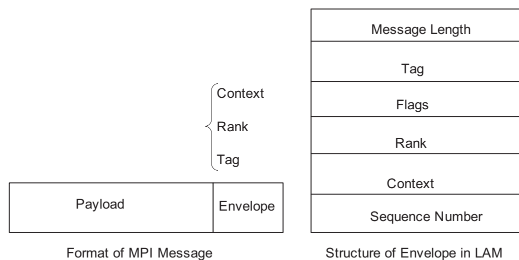

Figure 2: MPI and LAM envelope format

2.2.1 *Message Progression Layer*

Implementations of the MPI standard typically provide a request progression module that is responsible for progressing requests from initialization to completion. One of the main functions of this module is to support asynchronous and nonblocking message progression in MPI. The request progression module must maintain state information for all requests, including how far the request has progressed and what type of data/event is required for it to reach completion. We re-designed LAM's request progression interface (RPI) layer module for TCP to make use of SCTP. LAM's request progression mechanism is representative of the way requests are handled in other MPI middleware implementations and, due to its modular design, provided a convenient platform for us to work in.

2.2.2 *Message Delivery Protocol*

In LAM, each message body is preceded by an envelope that is used to match a message. We can broadly classify the messages in three categories with respect to the way LAM treats them internally; short messages, which are, by default, of size 64K bytes or less, long messages which are of size greater than 64K bytes and synchronous short messages. Figure 2 shows the format of an envelope in LAM. The flag field in the envelope indicates what type of message body follows it.

Short messages are passed using eager-send and the message body immediately follows the envelope. If the receiving process has posted/issued a matching receive buffer the message is received and copied into the buffer and the request is marked as done. If no matching receive has been posted, then this is an unexpected message and it is buffered by LAM in an internal hash table. Whenever a new request is posted, it is first checked against all of the buffered unexpected messages for a possible match. If a match is found, then the message is copied in the request receive buffer and the corresponding buffered message is discarded.

Long messages are handled differently than short messages and are not sent eagerly, but instead sent using the following rendezvous scheme: Initially only the envelope of the long message is sent to the receiver, if the receiver has posted a matching receive request then it sends back an acknowledgment (ACK) to the sender to indicate that it is ready to receive the message body. The sender sends back an envelope followed by the long message body in response to the ACK received. If no matching receive was posted at the time the initial long envelope was received, it is treated as an unexpected message. Later when a matching receive request is posted, it sends back an ACK and the rendezvous proceeds as above. Use of eager send for long messages is a topic that has been under research and people have found that an eager protocol for long messages out-performs a rendezvous protocol only if a significant number of receives have been pre-posted [3]. In [31] the authors report that protocols like eager send can lead to resource exhaustion in large cluster environments.

Synchronous short messages are also communicated using eager-send, however, the send is not complete until the sender receives an ACK from the receiver. The discussion above about the handling of unexpected messages also applies here. Collectives in the TCP module of LAM are implemented on top of point-to-point communication.

2.3 Overview of using SCTP for MPI

SCTP promises to be particularly well-suited for MPI due to its message-oriented nature and provision of multiple streams in an association. As shown in Figure 3, there are some striking similarities between SCTP and MPI. Contexts in an MPI program identify a set of processes that communicate with each other, and this grouping of processes can be represented as a one-to-many socket in SCTP that establishes associations with that set of processes. SCTP can map each

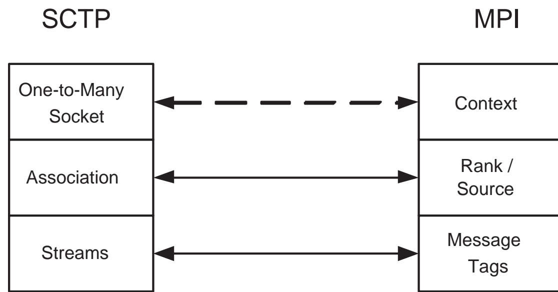

Figure 3: Similarities between the message protocol of MPI and SCTP

association to the unique rank of a process within a context and thus use an association number to determine the source of a message arriving on its socket. Each association can have multiple streams which are independently ordered and this property directly corresponds with message delivery order semantics in MPI. In MPI, messages sent with different tag/rank/context to the same receiver are allowed to overtake each other. This permits direct mapping of streams to message tags.

This similarity between SCTP and MPI is workable at a conceptual level, however, there are some implementation issues, to be discussed, that make the socket-context mapping less practical. Therefore, we preserved the mappings from associations to ranks and from streams to message tags but not the one from socket to context in our implementation. Context creation within an MPI program can be a dynamic operation and if we map sockets to contexts, this requires creating a dynamic number of sockets during the program execution. Not only does this add complexity and additional bookkeeping to the middleware, there can be an overhead to managing a large number of sockets. Creation of a lot of sockets in the middleware counteracts the benefits we can get from using a single one-to-many SCTP socket that can send/receive messages from multiple associations. Due to these reasons, instead of mapping sockets to contexts, the context and tag pair was used to map messages to streams. There is, however, an alternative way of dealing with contexts and that is to map them to PID (Payload Identifier) present in the common SCTP header of each packet. The PID is a 32-bit field that can be used at the application level to label the contents of an SCTP packet and is ignored by the SCTP layer. Using the PID field gives us the flexibility of dynamic context creation in our application without the need for maintaining a corresponding number of sockets. Also, the PID mapping can be easily incorporated in our module, with minor modifications.

3. DESIGN AND IMPLEMENTATION

In this section we discuss the design and implementation details of our SCTP module and also compare it to the LAM-TCP design. In Sections 3.1 and 3.2 we discuss how multiple streams are used in our module and the enhanced concurrency obtained as a result. In Section 3.3 we discuss resource management issues in the middleware and provide a comparison with LAM-TCP. Section 3.4 describes race conditions in our module and the solution adopted to fix them. In Section 3.5 we discuss enhanced reliability and fault tolerance in our module due to the features provided by SCTP. In Section 3.6 we discuss some limitations of using SCTP.

3.1 Message Demultiplexing

SCTP provides a one-to-many UDP-like style of communication that allows a single socket descriptor to receive messages from multiple associations, eliminating the need for maintaining a large number of socket descriptors. In the case of LAM's TCP module there is a one-to-one mapping between processes and socket descriptors, where every process creates individual sockets for each of the other processes in the LAM environment.

In SCTP's one-to-many style, the mapping between processes and sockets no longer exists. With one-to-many sockets there is no way of knowing when a particular association is ready to be read from or written to. At any time, a process can receive data sent on any of the associations through its sole socket descriptor. SCTP's one-to-many socket API does not permit reception of messages from a particular association, therefore, messages are received by the application in the order they arrive and only afterwards is the receive information examined to determine the association it arrived on.

In our SCTP module, each message goes through two levels of demultiplexing; first on the association number the message arrived on and secondly on the stream number within that association. These two parameters allow us to invoke the correct state function for that request, which directs the incoming message to the proper request receive buffer. If no matching receive was posted, it is treated like an unexpected message and buffered.

3.2 Concurrency and SCTP streams

MPI send/receive calls without wildcards define an ordered stream of messages between two processes. It is also possible, through the use of wildcards or appropriate tags, to relax the delivery order of messages. Relaxing the ordering of messages can be used to make the program more message driven and independent from network delay and loss. However, when MPI is implemented on top of TCP, the stream-oriented semantics of TCP with one connection per process precludes having unordered message streams from a single process. This restriction is removed in our implementation by mapping tags onto SCTP streams, which allows different tags from the same source to be independently delivered.

3.2.1 Assigning Messages to Streams

As discussed in the previous section, message matching in MPI is based on the tag, rank and context (TRC) of the message. MPI semantics require that messages with the same TRC must be delivered in order, i.e., they may not overtake each other, whereas MPI messages with different TRCs are not required to be ordered. In SCTP, the number of streams is a static parameter (short integer value) that is set when an association is initially established. For each association, we use a fixed sized pool of stream numbers, 10 by default, that is used for sending and receiving messages between the endpoints of that association. Messages with different TRCs are mapped to different stream numbers within an association to permit independent delivery.

Of course, since the number of streams is fixed, the degree of concurrency achieved depends on the number of streams.

3.2.2 *Head-of-Line Blocking*

Head-of-line blocking can occur in the LAM TCP module when messages have the same rank and context but different tags. Our assignment of TRC to streams alleviates this problem by allowing the unordered delivery of these messages.

It is often the case that MPI applications are loosely synchronized and as a result alternate between computation and bursts of communication. We believe this characteristic of MPI programs makes them more likely to be affected by head-of-line blocking in high loss scenarios, which underpins our premise that SCTP is better suited as a transport mechanism for MPI than TCP on WANs.

3.2.3 *Example of Head-of-Line Blocking*

As an illustration of our SCTP module's benefits, consider the two communicating MPI processes P0 and P1 shown in Figure 4. Process P1 sends two messages `Msg-A` and `Msg-B`

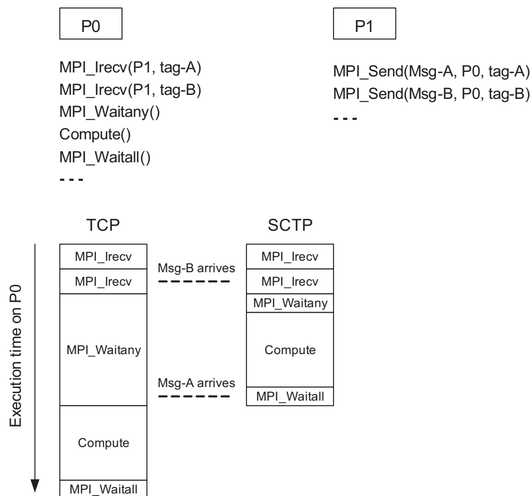

Figure 4: Example of multiple streams between two processes

to P0 in order, using different tags. P0 does not care what order it receives the messages and, therefore, posts two non-blocking receives. P0 waits for any of the receive requests to complete and then carries out some computation. Assume that part of `Msg-A` is lost during transmission and has to be retransmitted while `Msg-B` arrives safely at the receiver. In the case of TCP, its connection semantics require that `Msg-B` stay in TCP's receive buffer until the two sides recover from the loss and the lost parts are retransmitted. Even though the programmer has specified that the messages can be received in any order, P0 is forced to receive `Msg-A` first and incur increased latency as a result of TCP's semantics. In the case of the SCTP module, since different message tags are mapped to different streams and each stream can deliver messages independently, `Msg-B` can be delivered to P0 and the process can continue executing until `Msg-A` is required. SCTP matches the MPI semantics and makes it possible to take advantage of the concurrency that was specified in the program. The TCP module offers concurrency at process level, while our SCTP module adds to this with enhanced concurrency at the TRC level.

Even when blocking communication takes place between a pair of processes and there is loss, there are still advantages to using SCTP. Consider Figure 5 where blocking sends and

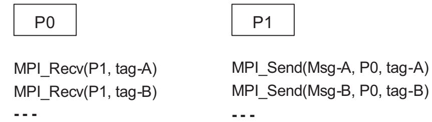

Figure 5: Example of multiple streams between two processes using blocking communication

receives are being used with two different tags. In SCTP, if `Msg-A` is delayed or lost, and `Msg-B` arrives at the receiver, it is treated as an unexpected message and buffered. SCTP, thus, allows the receive buffer to be emptied and hence does not slow down the sender due to the flow control mechanism. In TCP, however, `Msg-B` will occupy the socket receive buffer until `Msg-A` arrives.

3.2.4 *Maintaining State*

The LAM TCP module maintains state for each process (mapped to a unique socket) that it reads from or writes to. Since TCP delivers bytes in strict sequential order and the TCP module transmits/ receives one message body at a time per process, there is no need to maintain state information for any other messages that may arrive from the same process since they cannot overtake the message body currently being read. In our SCTP module, this assumption no longer holds true since subflows on different stream numbers are only partially ordered with respect to the entire association. We, therefore, maintain state for each stream number that a message can arrive on from a particular process. We only need to maintain a finite amount of state information per association since we limit the possible number of streams to our pool size. For each stream number, the state holds information about how much of the message has been read, what stage of progression the request is in and what needs to be done next. At the time when an attempt is made to read an incoming message, we cannot tell in advance from where the message will arrive and how large it will be, therefore, we specify a length equal to the socket receive buffer. SCTP's receive function `sctp_recvmsg`, however, takes care of message framing by returning the next message and not the number of bytes specified by the size field in the function call. This frees us from having to look through the receive buffer to locate the message boundaries.

3.3 Resource Management

LAM's TCP RPI module uses TCP as its communication mechanism and employs a socket based interface in a fully connected environment. It uses a one-to-one connection oriented scheme and maintains *N* sockets, one for each of the *N* processes in its environment. The `select` system call is used to poll these sockets to determine their readiness for reading any incoming messages or writing outgoing messages. Polling of these sockets is necessary because operating systems like Linux and UNIX do not support asynchronous communication primitives for sockets. It has been shown that the time taken for the `select` system call has a cost

associated with it that grows linearly with the number of sockets [20]. Socket design was originally based on the assumption that a process initiates a small number of connections, and it is known that performance is affected if a large number of sockets are used. Implications of this are significant in large scaled commodity clusters with thousands of nodes that are becoming more frequently used. Use of collective communications in MPI applications also strengthens the argument, since nearly all connections become active at once when a collective communication starts, and handling a large number of active connections may result in performance degradation.

SCTP's one-to-many communication style eliminates the need for maintaining a large number of socket descriptors. In our implementation each process creates a single one-to-many SCTP socket for communicating with the other processes in the environment. An association is established with each of the other processes using that socket and since each association has a unique identification, it maps to a unique process in the environment. We do not use the `select` system call to detect events on the socket, instead an attempt is made to retrieve messages at the socket using `sctp_recvmsg`, as long as there are any pending receive requests. In a similar way, if there are any pending send requests, they are written out to the socket using `sctp_sendmsg`. If the socket returns `EAGAIN`, signaling that it is not ready to perform the current read or write operation, the system attempts to advance other outstanding requests.

The use of one-to-many sockets in SCTP results in a more scalable and portable implementation since it does not impose a strong requirement on the system to manage a large number of socket descriptors and the resources for the associated buffers [4]. We investigated a process's limits for the number of socket descriptors it can manage and the number of associations it can handle on a single one-to-many socket. In our setup, a process reached its maximum socket descriptors' limit much earlier than the limit on maximum number of associations it can handle. The number of file descriptors in a system usually has a user limit of 1024 and needs root privileges to be changed.

3.4 Race Conditions

In the SCTP module it was necessary to change the long message protocol. This occurred because even though SCTP is message-based the size of message that can be sent in a single call to `sctp_sendmsg` function is limited by the send buffer size. As a result large messages had to broken up into multiple smaller messages of size less than that of send buffer. These messages then had to reassembled at the receiving side.

All pieces of the large message are sent out on the same stream number to ensure in-ordered delivery. As a refinement to this scheme, we considered interleaving messages sent on different streams with portions of a message larger than the send buffer size, at the time it is passed to the SCTP transport layer. Since reassembly of the large message is done at the RPI level, and not at the SCTP level, this may result in reduced latency for shorter messages on other streams especially when processes use non-blocking communication.

While testing our SCTP module we encountered race conditions that occurred due to the rendezvous mechanism of the long message protocol. Consider the case when two processes simultaneously exchange long messages using the same stream number. According to the long message protocol, both processes send a rendezvous initiating envelope to the other and wait for an acknowledgment before starting the transmission of the body of the long message (see Figure 6).

As shown in Figure 6, after P1 receives the `ACK` for its long

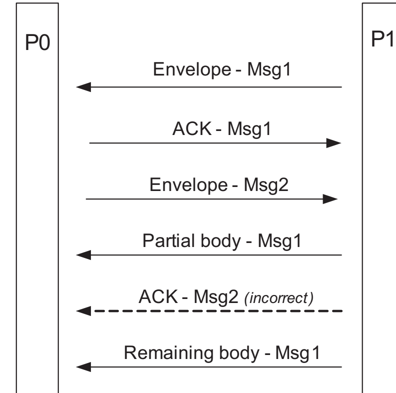

Figure 6: Example of long message race condition

message `Msg1`, P1 begins transmitting the message. If P1 successfully writes only part of `Msg1` to the socket, then P1 saves its current state, which includes the number of bytes sent to P0. Process P1 now continues trying to advance other active requests and returns later to writing the rest of `Msg1`. While advancing other messages, if P1 sends an `ACK` to P0 for long message `Msg2` using the same stream number that `Msg1` used, then P0, which has recieved part of `Msg1`, now receives a message from the same process on the same stream number as `Msg1`. Since P0 uses the association and stream number to decide on the communication request to be advanced, P0 incorrectly takes the `ACK` for `Msg2` as part of the body of `Msg1`.

3.4.1 Option A

In order to fix the above race condition we considered several options, and the tradeoffs associated with them. The first option was to stay in a loop while sending `Msg1` until all of it has been written out to the socket. One advantage of this is that the rendezvous mechanism introduces latency overhead and once the transmission of long message body starts, we do not want to delay it any longer. The disadvantage is that it greatly reduces the amount of concurrency that would otherwise have been possible, since we do not receive from or send to other streams or associations. Also, if the receiver of the long message had issued a non-blocking receive call, then it might continue to do other computations, and, as a result, sending the data in a tight loop may simply cause the sender to stall waiting for the receiver to remove data from the connection. Another disadvantage that can arise is when the receiver is slow and we overload it by sending a lot of data in a loop, instead of advancing other requests on other streams or associations.

3.4.2 *Option B*

The second option was to disallow P1 from writing a different message, if a previous message writing operation was still in progress, on the same stream number to the same process. The advantage of this was simplicity in design, but we reduce the amount of overlap that was possible in the above case if we allowed both processes to exchange long messages at the same time. However, it is still possible to send/receive from other streams or associations. This option is the one we implemented in our module.

3.4.3 *Option C*

A third option was to treat acknowledgments used within LAM to synchronize message exchange, such as those used in the long message rendezvous mechanism, as control messages. These control messages would be treated differently from other messages containing actual data. Whenever a control message carrying an ACK arrives, the middleware would know it is not part of any unfinished message, e.g. the long message in our example, and would invoke the correct state function to handle it. This option introduces more complexity and bookkeeping to the code, but may turn out to be one that offers the most concurrency.

There was one other race condition that occurred in implementing the SCTP module. Because the one-to-many SCTP style does not require any of the customary `accept` and `connect` function calls before receiving messages, we had to be careful about `MPI_Init` to ensure that each process establishes associations with all the other processes before exchanging messages. In order to ensure that all associations are established before messages are exchanged, we implemented a barrier at the end of our association setup stage and before the message reception/ transmission stage. This is especially important in an heterogeneous environment with varying network and machine speeds. It is easier with TCP because the `MPI_Init` connection setup procedure automatically takes care of this issue by its use of `connect` and `accept` function calls.

3.5 Reliability

In general, MPI programs are not fault tolerant and communication failure typically causes the programs to fail. As well, network loss which results in added latency and reduced bandwidth can severely impact the overall performance of programs. There are several mechanisms in SCTP that improve the overall reliability of executing an MPI program in a WAN environment.

3.5.1 *Multihoming Feature*

MPI applications have a strong requirement for the ability to rapidly switch to an alternate path without excessive delays in the event of failure. MPI processes in a given application's environment tend to be loosely synchronized even though they may not all be communicating directly with each other. Delays occurring in one process due to failure of the network can have a domino effect on other processes and they can potentially get delayed as well. It would be highly advantageous for MPI processes if the underlying transport protocol supports fast path failover in the event of network failure.

SCTP's multihoming feature provides an automatic failover mechanism where a communication failure between two endpoints on one path will switch over to an alternate path. Of course, this is useful only when there are independent paths, but having multiple interfaces on different networks is not uncommon and SCTP makes it possible to exploit the possibility when it exists. SCTP provides several user-adjustable controls that can be used to change the amount of time it takes to determine network problems [25].

Ideally, these control parameters need to be tuned to a particular network, and perhaps even the application, to ensure fast failover but to avoid unnecessary failovers due to network delay. Note, there is no similar mechanism in TCP. It could be accomplished by using multiple sockets and managing them in the middleware but this introduces added complexity to the middleware.

3.5.2 *Added Protection*

SCTP has several in-built features that provide an added measure of protection against flooding and masquerade attacks. They are discussed below.

As SCTP is connection oriented, it exchanges setup messages at the initiation of the communication, for which it uses a robust four-way handshake as shown in Figure 7. The receiver of an `INIT` message does not reserve any resources until the sender proves that its IP address is the one claimed in the association setup message. The handshake uses a signed state cookie to prevent use of IP spoofing for a `SYN` flooding denial of service attack. The cookie is authenticated by making sure that it has a valid signature and then a check is made to verify that the cookie is not stale. This check guards against replay attacks [27]. Some implementations of TCP also use a cookie mechanism, but those typically do not use signed cookies and also the cookie mechanism is supplied as an option and it is not an integral part of the TCP protocol as is the case with SCTP. In TCP, a user has to validate that both sides of the connection have the required implementation. SCTP's four-way handshake may be seen to be an overhead, however, if a one-to-many style socket is used, then user data can be piggy-backed on the third and fourth leg of the handshake.

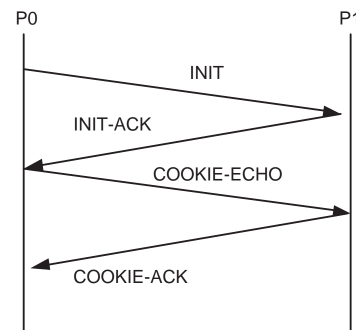

Figure 7: SCTP's four-way handshake

Every SCTP packet has a common header that contains, among other things, a 32-bit verification tag. This verification tag protects against two things: (1) it prevents an

SCTP packet from a previous inactive association from being mistaken as a part of a current association between the same endpoints, and (2) it protects against blind injection of data in an active association [27].

A study [30] done on TCP Reset attacks has shown that TCP is vulnerable to a denial of service attack in which the attacker tries to prematurely terminate an active TCP connection. The study has found that this type of attack can utilize the TCP window size to reduce the number of sequence numbers that must be guessed for a spoofed packet to be considered valid and accepted by the receiver. This kind of attack is especially effective on long standing TCP flows such as BGP peering connections, which if successfully attacked, would disrupt routing in the Internet. SCTP is not susceptible to the reset attacks due to its verification tag.

SCTP is able to use large socket buffer sizes by default because it is not subject to denial of service attacks that TCP becomes susceptible to with large buffers [26].

SCTP also provides an autoclose option that protects against accidental denial-of-service attacks where a process opens an association but does not send any data. Using autoclose, we can specify the maximum number of seconds that an association remains idle, i.e., no user traffic in either direction. After this time the association is automatically closed [25].

There is another difference between SCTP's and TCP's close mechanisms; TCP allows an "half-closed" state, which is not allowed in SCTP. In the half-closed state, one side closes a connection and cannot send data, but may continue to receive data until the peer closes its connection. SCTP avoids the possibility of a peer failing to close its connection by disallowing a half-closed state.

In this section we have highlighted the additional security features that are present in SCTP. These features are especially important in open network environments, such as the Internet, where security is an issue and therefore the use of SCTP adds to the reliability of the environment for MPI.

3.5.3 Use of a Single Underlying Protocol

One issue not directly related to SCTP was the mixed use of UDP and TCP in LAM. LAM uses user level daemons that by default employ UDP as their transport protocol. These daemons serve several purposes such as enabling external monitoring of running jobs and carrying out cleanup when a user aborts an MPI process. LAM also provides remote I/O via the LAM daemon processes. We modified the LAM daemons to use SCTP so that the entire execution now uses SCTP and all the components in the LAM environment can take advantage of the features of SCTP.

3.6 Limitations

As discussed in Section 3.4, SCTP has a limit on the size of message it can send in a single call to `sctp_sendmsg`. This limits us from taking full advantage of the message-framing property of SCTP since our long message protocol divides a long message into smaller messages and then carries out message framing at the middleware level.

Both TCP and SCTP use a flow control mechanism where a receiver advertises its available receive buffer space to the sender. Normally when a message is received it is kept in the kernel's protocol stack buffer until a process issues a `read` system call. When large sized messages are exchanged in MPI applications and messages are not read out quickly, the receiver advertises less buffer space to the sender, which slows down the sender due to flow control. An event driven mechanism that gets invoked as soon as a message arrives is currently not supported in our module.

Some factors that can affect the performance of SCTP are as follows:

- SCTP uses a comprehensive 32 bit CRC32c checksum which is expensive in terms of CPU time, while TCP typically offloads checksum calculations to the network interface card (NIC). However, CRC32c provides much greater data protection at the cost of additional CPU time and it is likely that hardware support for it may become available in a few years.
- TCP can always pack a full MTU, but SCTP is limited by the fact that it bundles different messages together, which may not always fit to pack a full MTU. However, in our experiments this was not observed to be a factor impacting the performance.
- TCP performance has been fine-tuned over the past decades, however, optimization of the SCTP stack is still in its early stages and will improve over time.

4. EXPERIMENTAL EVALUATION

In this section we describe the experiments carried out to compare the performance of our SCTP module with the LAM-TCP module for different MPI applications. Our experimental setup consists of a dedicated cluster of eight identical Pentium-4 3.2GHz, FreeBSD-5.3 nodes connected via a layer-two switch using 1Gbit/s Ethernet connections. Kernels on all nodes are augmented with the Kame.net SCTP stack [16]. The experiments were performed in a controlled environment and Dummynet was configured on each of the nodes to allow us to vary the amount of loss on the links between the nodes. We compare the performance under different loss rates of 0%, 1% and 2% between each of the eight nodes.

In Section 4.1, we evaluate the performance of the SCTP module using two standard benchmark programs. In Section 4.2, we look at the effect of using a latency tolerant program that overlaps communication with computation. We compare the performance using a real world program that uses a manager-worker communication pattern commonly found in MPI programs. We also discuss the effects of head-of-line blocking. In order to make the comparison between SCTP and TCP as fair as possible, the following settings were used in all the experiments discussed in subsequent sections:

1. By default, SCTP uses a larger `SO_SNDBUF/SO_RCVBUF` buffer size than TCP. In order to prevent any possible effects on performance due to this difference, the send and receive buffers were set to a value of 220 Kbytes in both the TCP and SCTP modules.

2. Nagle's algorithm is disabled by default in LAM-TCP and this setting was used in the SCTP module as well.

3. An SCTP receiver uses Selective Acknowledgment `SACK` to report any missing data to the sender, therefore, `SACK` option for TCP was enabled on all the nodes used in the experiment.

4. In our experimental setup, the ability to multihome between any two endpoints was available since each node is equipped with three gigabit interface cards and three independent paths between any two nodes were available. In our experiments, however, this multihoming feature was not used so as to keep the network settings as close as possible to that used by the LAM-TCP module.

5. TCP is able to offload checksum calculations on to the NICs on our nodes and thus has zero CPU cost associated with its calculation. SCTP has an expensive CRC32c checksum which can prove to be a considerable overhead in terms of CPU cost. We modified the kernel to turn off the CRC32c checksum in SCTP so that this factor does not affect the performance results.

4.1 Evaluation of Benchmark Programs

We evaluated our implementation using two benchmark programs; MPBench ping-pong test and NAS parallel benchmarks. The NAS benchmarks approximate the performance of real applications. These benchmarks provided a point of reference for our measurements and the results are discussed below.

4.1.1 MPBench Ping-Pong Test

We first report the output obtained by running the MPBench [22] ping-pong test with no message loss. This is a standard benchmark program that reports the throughput obtained when two processes repeatedly exchange messages of a specified size. All messages are assigned the same tag. Figure 8 shows the throughput obtained for different mes-

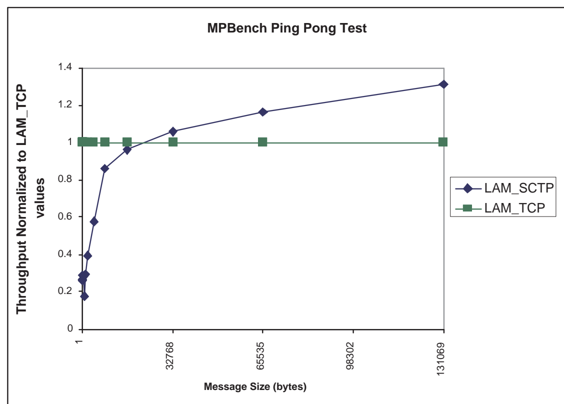

Figure 8: The performance of SCTP using a standard ping-pong test normalized to TCP under no loss

sage sizes in the ping-pong test under no loss. The throughput values for the LAM-SCTP module are normalized with respect to LAM-TCP. The results show that SCTP is more efficient for larger message sizes, however, TCP does better for small message sizes. The crossover point is approximately at a message size of 22 Kbytes, where SCTP throughput equals that of TCP. The SCTP stack is fairly new compared to TCP and these results may change as the SCTP stack is further optimized.

We also used the ping-pong test to compare SCTP to TCP under 1% and 2% loss rates. Since the LAM middleware treats short and long messages differently, we experimented with loss for both types of messages. In the experiments short messages were 30 Kybtes and long messages were 300 Kbytes. For both short and long messages under 1% and 2% loss, SCTP performed better than TCP and the results are summarized in Table 1.

MPI Message Size (bytes)	Throughput(bytes/second) Loss:1% SCTP	TCP	Loss:2% SCTP	TCP
30K	54,779	1,924	44,614	1,030
300K	5,870	1,818	2,825	885

Table 1: The performance of SCTP and TCP using a standard ping-pong under loss

SCTP shows that it is more resilient under loss and can perform rapid recovery of lost segments. Work done in [1] and [24] compares the congestion control mechanisms of SCTP and TCP and shows that SCTP has a better congestion control mechanism that allows it to achieve higher throughput in error prone networks. Some features of SCTP's congestion control mechanism are as follows:

- Use of SACK is an integral part of the SCTP protocol, whereas, for TCP it is an option available in some implementations. In those implementations SACK information is carried in IP options and is, therefore, limited to reporting at most four TCP segments. In SCTP, the number of gap ACK blocks allowed is much larger as it is dictated by the PMTU [27].

- Increase in the congestion window in SCTP is based on the number of bytes acknowledged and not on the number of acknowledgments received [1]. This allows SCTP to recover faster after fast retransmit. Also, SCTP initiates slow start when the congestion window is equal to the slow start threshold. This helps in achieving a faster increase in the congestion window.

- The congestion window variable `cwnd` can achieve full utilization because when a sender has 1 byte of space in the `cwnd` and space available in the receive window, it can send a full PMTU of data.

- The FreeBSD KAME SCTP stack also includes a variant called *New-Reno* SCTP that is more robust to multiple packet losses in a single window [15].

- When the receiver is multihomed, an SCTP sender maintains a separate congestion window for each transport address of the receiver because the congestion status of the network paths may differ from each other.

SCTP's retransmission policy helps in increasing the throughput, since retransmissions are sent on one of the active alternate transport addresses of the receiver. The effect of multihoming was not a factor in our tests.

4.1.2 NAS Benchmarks

In our second experiment, we used the NAS parallel benchmarks (NPB version 3.2) to test the performance of MPI programs with SCTP and TCP as the underlying transport protocol. These benchmarks approximate the performance that can be expected from a portable parallel application. The suite of benchmarks consists of eight programs, however, we tested the following seven of those: LU (LU Factorization), IS (Integer Sort), MG (Multi-Grid Method), EP (Embarrassingly Parallel), CG (Conjugate Gradient), BT (Block Tridiagonal ADI) and SP (Scalar Pentadiagonal ADI). The eighth benchmark FT (Fourier Transform) was not used because it does not compile with mpif77. Performance of the benchmarks is reported using Mop/s total value. Dataset sizes S, W, A and B were used in the experiments with the number of processes equal to eight. The Dataset sizes increase in order S, W, A, and B with S being the smallest. We have done an analysis of the type of messages being exchanged in these benchmarks and we have found that in datasets 'S' and 'W', short messages (as defined in LAM middleware to be messages smaller than or equal to 64 Kbytes) are predominantly being sent/received. In datasets 'A' and 'B' we see a greater number of long messages (i.e., messages larger than 64 Kbytes) being exchanged.

Figure 9 shows the results for dataset size 'B' under no loss.

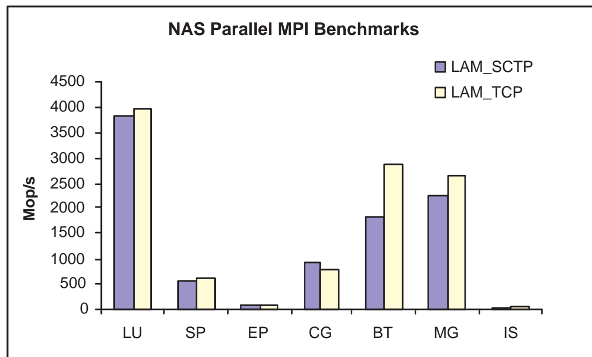

Figure 9: TCP versus SCTP for the NAS benchmarks

The results for the other datasets are not shown in the figure, but as expected from the ping-pong test results, TCP does better for the shorter datasets. These benchmarks use single tags for communication between any pair of nodes and the benefits of using multiple streams in the SCTP module are not being utilized. The SCTP module, in this case, reduces to a single stream per association and as the results show, the performance on average is comparable to TCP. TCP shows an advantage for the MG and BT benchmarks and we believe the reason for this is that these benchmarks use greater proportion of short messages in dataset 'B' than the other benchmarks.

4.2 Evaluation of a Latency Tolerant Program

In this section we compare the performance of the SCTP module with LAM-TCP using a latency tolerant program. In order to take advantage of highly available, shared environments that have large delays and loss, it is important to develop programs that are latency tolerant and capable of overlapping computation with communication to a high degree. Moreover, SCTP's use of multiple streams can provide more concurrency. Here, we investigate the performance of such programs with respect to TCP and also discuss the effect of head-of-line blocking in these programs.

4.2.1 Comparison for a Real World Program

In this section we investigate the performance of a realistic program that makes use of multiple tags. Our objectives were two-fold: first, we wanted to evaluate a real-world parallel application that is able to overlap communication with computation, and secondly, to examine the effect of introducing multiple tags which, in the case of the SCTP module, can map to different streams. We describe the experiments performed using an application which we call the *Bulk Processor Farm* program. The communication characteristics of this program is typical of real-world manager-worker programs.

The Bulk Processor Farm is a request driven program with one manager and several workers. The workers ask the manager for tasks to do, and the manager is responsible for creating tasks, distributing them to workers and then collecting the results from all the workers. The manager services all task requests from workers in the order of their arrival (`MPI_ANY_SOURCE`). Each task is assigned a different tag, which represents the type of that task. There is a maximum number of different tags (`MaxWorkTags`) that can be distributed at any time. The workers can make multiple requests at the same time and when they finish doing some task they send a request for a new one, so at any time, each of the workers have a fixed number of outstanding job requests. This number was chosen to be ten in our experiments. The workers use non-blocking MPI calls to issue send and receive requests and all messages received by the workers are expected messages. The workers use `MPI_ANY_TAG` to show that they are willing to perform a task of any type.

The manager has a total number of tasks (`NumTasks`) that need to be performed before the program can terminate. In our experiments we set that number to 10,000 tasks. We experimented with tasks of two different sizes; short tasks equal to 30 Kbytes and long tasks equal to 300 Kbytes. The size of the task represents the message sizes that are exchanged between the manager and the worker.

In case of the SCTP module, the tags will be mapped to streams and in case of congestion if a message sent to a worker is lost, then it would still be possible for messages on the other streams to be delivered and the worker program can continue working without blocking on the lost message. In LAM-TCP, however, since all messages between the manager and a worker must be delivered in order, there is less overlap of communication with computation. The experiments were run at loss rates of 0%, 1% and 2%. The farm program was run six times for each of the different combinations of loss rates and message sizes and the average

value of total run-times are reported. The average and the median values of the multiple runs were very close to each other. We also calculated the standard deviation of the average value, and found the variation across different runs to be very small.

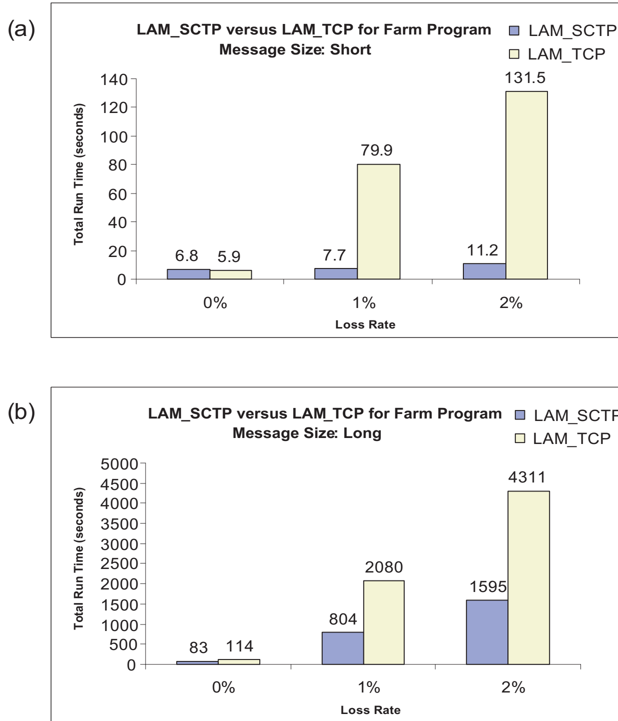

Figure 10: TCP versus SCTP for (a) short and (b) long messages for the Bulk Processor Farm application

Figure 10 shows the comparison between LAM-SCTP and LAM-TCP for short and long message sizes under different loss rates. As seen in Figure 10, SCTP outperforms TCP under loss. For short messages, LAM-TCP run-times are 10 to 11 times higher than that of LAM-SCTP at loss rates of 1 to 2%. For long messages, LAM-TCP is slower than LAM-SCTP by 2.58 times at 1% and 2.7 times at 2% loss. Although SCTP performs substantially better than TCP for both short and long messages, the fact that the difference is more pronounced for short messages is very positive. MPI implementations, typically, try to optimize short messages for latency and long messages for bandwidth [3]. In our latency tolerant application, the workers, at any time, have a number of pre-posted outstanding receive requests. Since short messages are sent eagerly by the manager, they are copied to the correct receive buffer as soon as they are received. Since messages are being transmitted on different streams in LAM-SCTP, there is less variation in the amount of time the worker might have to wait for a job to arrive, especially under loss conditions, and results in more overlap of communication with computation. The rendezvous mechanism in the long messages introduces synchrony to the transmission of the messages, and the payload can only be sent after the receiver has acknowledged its readiness to accept it. This reduces the amount of overlap possible, compared to the case when messages are sent eagerly. The long messages are also more likely to be affected by loss by virtue of their size. Long messages are typically used for bulk transfers, and the cost of the rendezvous is amortized over the time required to transfer the data.

We also introduced a tunable parameter to the program described above, which we call `Fanout`. `Fanout` represents the number of tasks a manager will send to a worker in response to a single task request from that worker. In Figure 10, the `Fanout` is 1.

We experimented with a `Fanout` value of 10 as shown in Figure 11. We anticipated that the settings with `Fanout`=10

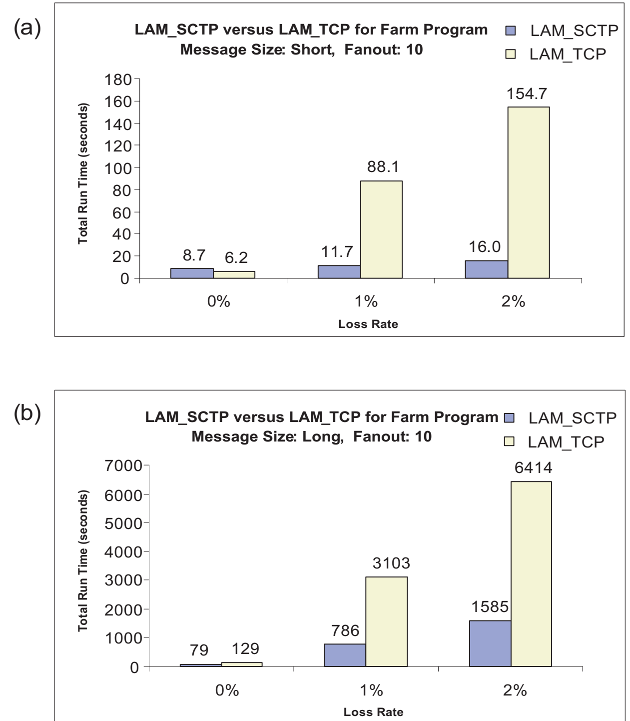

Figure 11: TCP versus SCTP for (a) short and (b) long messages for the Bulk Processor Farm application using `Fanout` of 10

would create more possibilities for head-of-line blocking in the LAM-TCP case, since in this case ten tasks are sent to a worker in response to one task request, and are more prone to be affected by loss. Figure 11 shows that with larger `Fanout`, the average run-time for TCP increases substantially for long messages, while there are only slight changes for short messages. One possibility for this behavior is TCP's flow-control mechanism. We are sending ten long tasks in response to a job request, and in case of loss, TCP blocks delivery of all subsequent messages until the lost segment is recovered. If it does not empty the receive buffer quickly enough, it can cause the sender to slow down. Moreover, as discussed in Section 4.1.1, SCTP's better congestion control also aids in faster recovery of lost segments, and this is also reflected in all our results. Also, we expect the performance of SCTP over TCP to further increase when multihoming is present and retransmissions are sent on an alternate path.

4.2.2 Investigating Head-of-line Blocking

In the experiments presented so far, we have shown that SCTP has superior performance than TCP under loss, and there are two main factors affecting the results: improvements in SCTP's congestion control mechanism, and the ability to use multiple streams to reduce head-of-line blocking. In this section we examine the performance improvement obtained as a result of using multiple streams in our SCTP module. In order to isolate the effects of head-of-line blocking in the farm program, we created another version of the SCTP module, one that uses only a single stream to send and/or receive messages irrespective of the message tag, rank and context. All other things were identical to our multiple-stream SCTP module.

The farm program was run at different loss rates for both short and long message sizes. The results are shown in Figure 12. Since multihoming was not present in the experiment, the results obtained show the effect of head-of-line blocking and the advantage due to the use multiple tags/streams. The reduction in average run-times when using multiple streams compared to a single stream is about 25% under loss for long messages. In the case of short messages, the benefit of using multiple streams becomes apparent at 2% loss where using a single stream shows an increase in run-time of about 35%. This shows that head-of-line blocking has a substantial effect in a latency tolerant program like the Bulk Processor Farm. The performance difference between the two cases, can increase in a network where it takes a long time to recover from packet loss [26].

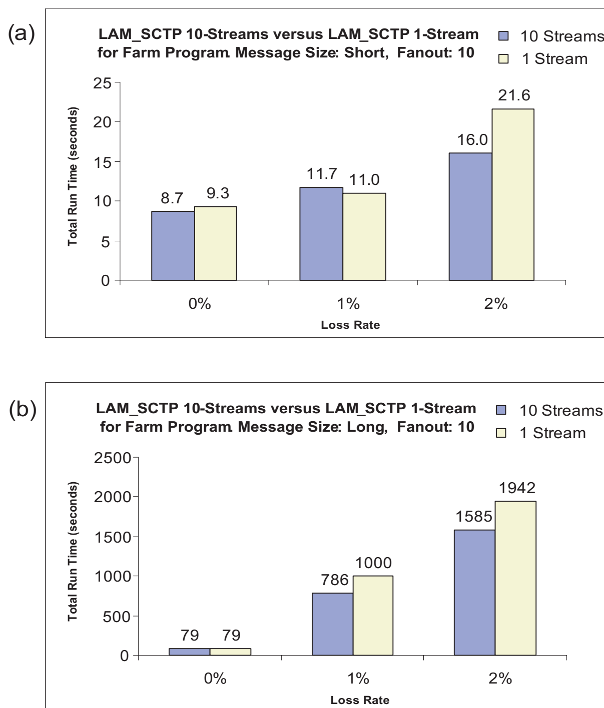

Figure 12: Effect of head-of-line Blocking in the Bulk Processor Farm program for (a) short and (b) long messages

5. RELATED WORK

The effectiveness of SCTP has been explored for several protocols in high latency, high-loss environments. Researchers have investigated the use of SCTP in FTP [18], HTTP [23], and also over wireless [7] and satellite networks [1]. Using SCTP for MPI has not been investigated.

There are a wide variety of projects that use TCP in an MPI environment. MPICH-G2 [17] is a multi-protocol implementation of MPI for the Globus environment that was primarily designed to link together clusters over wide area networks. MPICH-G2 can use a custom transport for inside the cluster and a TCP connection between clusters. LAM as well has a TCP-based Globus component to support meta-computing. Our work provides the opportunity to add SCTP for transport in a Grid-environment where we can take advantage of the improved performance in the case of loss.

Several projects have used UDP rather than TCP. As mentioned, UDP is message-based and one can avoid all the "heavy-weight" mechanisms present in TCP to obtain better performance. However, when ones adds reliability on top of UDP the advantages begin to diminish. For example, LA-MPI, a high-performance, reliable MPI library that uses UDP, reports performance of their implementation over UDP/IP to be similar to TCP/IP performance of other MPI implementations over Ethernet [2]. WAMP [28] is an example of UDP for MPI over wide area networks. Interestingly, WAMP only wins over TCP in heavily congested networks where TCP's congestion avoidance mechanisms limit bandwidth. This is a problem but hopefully research in this area will lead to better solutions for TCP that could also be incorporated into SCTP.

Another possible way to use streams is to send eager short messages on one stream and long messages on a second one. LA-MPI followed this approach with their UDP/IP implementation of MPI [2]. This potentially allows for the fast delivery of short messages and a more optimized delivery for long messages that require more bandwidth. It also may reduce head of line blocking for the case of short messages waiting for long messages to complete. However, one has to maintain MPI message ordering semantics and thus, as is the case in LA-MPI, sequence numbers were introduced to ensure that the messages are received strictly in the order they were posted by the sender. Thus, in the end, there is no added opportunity for concurrent progression of messages as there is in the case of our implementation.

There are a number of MPI implementations that explored improving communication through improved socket interfaces [20] or by changes to operating system. In general, these will not have the same degree of support, which limits their use. Although the same can be said about SCTP, the fact that SCTP has been standardized and implementations have begun to emerge are all good indications of more wide-spread support.

Recently Open MPI has been announced which is a new public domain version of MPI-2 that builds on the experience gained from the design and implementation of LAM/MPI, LA-MPI and FT-MPI [8]. Open MPI takes a component-based approach to allow the flexibility to mix and match

components for collectives and for transport and link management. We hope that our work could be incorporated into Open MPI as a transport module. TEG [31] is a fault-tolerant point-point communication module in Open MPI that supports multi-homing and the ability to stripe data across interfaces. The ability to schedule data over different interfaces has been proposed for SCTP and may provide an alternative way to provide the functionality of TEG in environments like the Internet. A group at the University of Delaware is researching Concurrent Multipath Transfer (CMT) [14, 13], which uses SCTP's multihoming feature to provide simultaneous transfer of data between two endpoints, via two or more end-to-end paths. The objective of using CMT between multihomed hosts is to increase an application throughput. CMT is at the transport layer and is thus more efficient, compared to multipath transfer at application level, since it has access to finer details about the end-to-end paths. CMT is currently being integrated into FreeBSD KAME SCTP stack and will be available as a `sysctl` option by the end of year 2005 [26].

6. CONCLUSIONS

In this paper we discuss the design and evaluation of using SCTP for MPI. SCTP is better suited as a transport layer for MPI because of its several distinct features not present in TCP. We have designed the SCTP module to address the head-of-line blocking problem present in LAM-TCP middleware. We have shown that SCTP matches MPI semantics more closely than TCP and we have taken advantage of the multistreaming feature of SCTP to provide a direct mapping from streams to MPI message tags. This has resulted in increased concurrency at the TRC level in the SCTP module compared to concurrency at process level in LAM-TCP.

Our SCTP module's state machine uses one-to-many style sockets and avoids the use of expensive `select` system calls, which leads to increased scalability in large scale clusters. In addition, SCTP's multihoming feature makes our module fault tolerant and resilient to network path failures.

We have evaluated our module and reported the results of experiments using standard benchmark programs as well as a real world application and compared the performance with LAM-TCP. Simple ping-pong tests, under no loss, have shown that LAM-TCP outperforms the SCTP module for small message sizes but SCTP does better for large messages. In addition, SCTP's performance is comparable to TCP's for standard benchmarks such as the NAS Benchmarks when larger dataset sizes are used. The strengths of SCTP over TCP become apparent under loss conditions as seen in the results for the ping-pong tests and our latency tolerant *Bulk Processor Farm* program. When different tags are used in a program, the advantages due to multistreaming in our module can lead to further benefits in performance.

In general, the performance requirements of different MPI programs will vary. There will be those programs that can only achieve satisfactory performance on dedicated machines, with low-latency and high-bandwidth links. On the other hand, there will be those latency tolerant programs that will be able to run just as well in highly available, shared environments that have larger delays and loss. Our contribution is to the latter type of programs, for extending their performance in open environments such as the Internet. Ideally, we want to increase the portability of MPI and in doing so, encourage programmers to develop programs more towards this end of the spectrum.

7. ACKNOWLEDGMENTS

We wish to thank Randall Stewart, one of the primary designers of SCTP, for answering our questions and providing timely fixes to the protocol stack that made it possible to provide the performance evaluation.

We would also like to thank the developers at LAM-MPI for their assistance on their mailing list. We would particularly like to thank Jeff Squyres for always providing extensive help regarding our questions about the LAM-MPI implementation.

8. REFERENCES

[1] Alamgir, R., Atiquzzaman, M., and Ivancic, W. Effect of congestion control on the performance of TCP and SCTP over satellite networks. In *NASA Earth Science Technology Conference* (Pasadena, CA, June 2002).

[2] Aulwes, R. T., Daniel, D. J., Desai, N. N., Graham, R. L., Risinger, L. D., Taylor, M. A., Woodall, T. S., and Sukalski, M. W. Architecture of LA-MPI, a network-fault-tolerant MPI. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04)* (Sante Fe, New Mexico, April 2004).

[3] Brightwell, R., and Underwood, K. Evaluation of an eager protocol optimization for MPI. In *PVM/MPI* (2003), pp. 327–334.

[4] Burns, G., and Daoud, R. Robust message delivery with guaranteed resources. In *Proceedings of Message Passing Interface Developer's and User's Conference (MPIDC)* (May 1995).

[5] Cardwell, N., Savage, S., and Anderson, T. Modeling TCP latency. In *INFOCOM* (2000), pp. 1742–1751.

[6] Center, N. A. R. Numerical aerodynamic simulation (NAS) parallel benchmark (NPB) benchmarks. http://www.nas.nasa.gov/Software/NPB/.

[7] Choi, Y., Lim, K., Kahng, H.-K., and Chong, I. An experimental performance evaluation of the stream control transmission protocol for transaction processing in wireless networks. In *ICOIN* (2003), pp. 595–603.

[8] Fagg, G. E., and Dongarra, G. E. Building and using a fault-tolerant MPI implementation. *International Journal of High Performance Computing Applications 18*, 3 (2004), 353–361.

[9] Feng, W., and Tinnakornsrisuphap, P. The failure of TCP in high-performance computational grids. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)* (Washington, DC, USA, 2000), IEEE Computer Society, p. 37.

[10] G. Burns, R. Daoud and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Supercomputing Symposium '94* (Toronto, Canada, June 1994).

[11] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. Open MPI: Goals, concept and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting* (Budapest, Hungary, September 2004).

[12] Hacker, T. J., Noble, B. D., and Athey, B. D. Improving throughput and maintaining fairness using parallel TCP. In *IEEE INFOCOM* (2004).

[13] Iyengar, J. R., Amer, P. D., and Stewart, R. Retransmission policies for concurrent multipath transfer using SCTP multihoming. In *ICON 2004, Singapore* (November 2004).

[14] Iyengar, J. R., Shah, K. C., Amer, P. D., and Stewart, R. Concurrent multipath transfer using SCTP multihoming. In *SPECTS 2004, San Jose* (July 2004).

[15] Jr., A. L. C., Shah, K., Iyengar, J. R., Amer, P. D., and Stewart, R. R. SCTP and TCP variants: Congestion control under multiple losses. Technical Report TR2003-04, CIS Dept, U of Delaware, February 2003.

[16] KAME. SCTP stack implementation for FreeBSD. http://www.kame.net.

[17] Karonis, N. T., Toonen, B. R., and Foster, I. T. MPICH-G2: A grid-enabled implementation of the message passing interface. *CoRR cs.DC/0206040* (2002).

[18] Ladha, S., and Amer, P. Improving multiple file transfers using SCTP multistreaming. In *Proceedings IPCCC* (April 2004).

[19] Mathis, M., Heffner, J., and Reddy, R. Web100: Extended TCP instrumentation for research, education and diagnosis. *SIGCOMM Comput. Commun. Rev. 33*, 3 (2003), 69–79.

[20] Matsuda, M., Kudoh, T., Tazuka, H., and Ishikawa, Y. The design and implementation of an asynchronous communication mechanism for the MPI communication model. In *IEEE Intl. Conf. on Cluster Computing* (Dana Point, Ca., Sept 2004), pp. 13–22.

[21] Medina, A., Allman, M., and Floyd, S. Measuring the evolution of transport protocols in the Internet, April 2005. To appear in ACM CCR.

[22] Mucci, P. J. MPBench: Benchmark for MPI functions. Available at: http://icl.cs.utk.edu/projects/llcbench/mpbench.html.

[23] Rajamani, R., Kumar, S., and Gupta, N. SCTP versus TCP: Comparing the performance of transport protocols for web traffic. Technical report, University of Wisconsin-Madison, May 2002.

[24] S. Fu, M. A., and Ivancic, W. SCTP over satellite networks. In *IEEE Computer Communications Workshop (CCW 2003)* (Dana Point, Ca., October 2003), pp. 112–116.

[25] Stevens, W. R., Fenner, B., and Rudoff, A. M. *UNIX Network Programming, Vol. 1, Third Edition*. Pearson Education, 2003.

[26] Stewart, R. Primary designer of SCTP, private communication, 2005.

[27] Stewart, R. R., and Xie, Q. *Stream control transmission protocol (SCTP): a reference guide*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[28] Vinkat, R., Dickens, P. M., and Gropp, W. Efficient communication across the Internet in wide-area MPI. In *Conference on Parallel and Distributed Programming Techniques and Applications* (Las Vegas, Nevada, USA, 2001).

[29] W. Gropp, E. Lusk, N. Doss and A. Skjellum. High-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing 22*, 6 (Sept. 1996), 789–828.

[30] Watson, P. A. Slipping in the window: TCP reset attacks, October 2003. CanSecWest Security Conference.

[31] Woodall, T., Graham, R., Castain, R., Daniel, D., Sukalski, M., Fagg, G., Gabriel, E., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J., Sahay, V., Kambadur, P., Barrett, B., and Lumsdaine, A. TEG: A high-performance, scalable, multi-network point-to-point communications methodology. In *Proceedings, 11th European PVM/MPI Users' Group Meeting* (Budapest, Hungary, September 2004).