# Towards MPI Progression Layer Elimination with TCP and SCTP

by

Bradley Thomas Penoff

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

**Master of Science**

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

**The University of British Columbia**

January 2006

© Bradley Thomas Penoff, 2006

# Abstract

MPI middleware glues together the components necessary for execution. Almost all implementations have a communication component also called a message progression layer that progresses outstanding messages and maintains their state. The goal of this work is to thin or eliminate this communication component by pushing the functionality down onto the standard IP stack in order to take advantage of potential advances in commodity networking. We introduce a TCP-based design that successfully eliminates the communication component. We discuss how this eliminated TCP-based design doesn't scale and show a more scalable design based on the Stream Control Transmission Protocol (SCTP) that has a thinned communication component. We compare the designs showing why SCTP one-to-many sockets in their current form can only thin and not completely eliminate the communication component. We show what additional features would be required of SCTP to enable a practical design with a fully eliminated communication component.

# Contents

**Abstract** **ii**

**Contents** **iii**

**List of Tables** **v**

**List of Figures** **vi**

**1 Introduction** **1**

**2 MPI Middleware** **9**

2.1 Middleware . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.2 MPI Runtime Execution Environment . . . . . . . . . . . . . . . . . 9

2.3 MPI Library . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.4 Message Progression Layer . . . . . . . . . . . . . . . . . . . . . . . 12

2.4.1 Message Matching . . . . . . . . . . . . . . . . . . . . . . . . 14

2.4.2 Expected/Unexpected Messages . . . . . . . . . . . . . . . . . 14

2.4.3 Short/Long Messages . . . . . . . . . . . . . . . . . . . . . . . 15

**3 TCP Socket-based Implementations** **17**

3.1 The LAM-TCP Message Progression Layer Implementation . . . . . 18

3.1.1 Message Matching . . . . . . . . . . . . . . . . . . . . . . . . 19

3.1.2 Expected/Unexpected Messages . . . . . . . . . . . . . . . . . 20

3.1.3 Short/Long Messages . . . . . . . . . . . . . . . . . . . . . . . 21
3.2 Socket per Message Stream MPI Implementation . . . . . . . . . . . 22
3.2.1 Message Matching . . . . . . . . . . . . . . . . . . . . . . . 24
3.2.2 Expected/Unexpected Messages . . . . . . . . . . . . . . . . . 25
3.2.3 Short/Long Messages . . . . . . . . . . . . . . . . . . . . . . . 27
3.3 Critique of the Socket per Stream Implementation . . . . . . . . . . 28
3.3.1 Number of Sockets . . . . . . . . . . . . . . . . . . . . . . . . 28
3.3.2 Number of System Calls . . . . . . . . . . . . . . . . . . . . . 29
3.3.3 Select System Call . . . . . . . . . . . . . . . . . . . . . . . . 30
3.3.4 Flow Control . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

**4 SCTP-based MPI Middleware** **32**

4.1 SCTP Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.2 SCTP-based Implementation of the Message Progression Layer . . . 36
4.2.1 Message Matching . . . . . . . . . . . . . . . . . . . . . . . 37
4.2.2 Expected/Unexpected Messages . . . . . . . . . . . . . . . . . 38
4.2.3 Short/Long Messages . . . . . . . . . . . . . . . . . . . . . . . 39

**5 Related Work** **41**

**6 Conclusions** **46**

**Bibliography** **48**

# List of Tables

6.1 Summary of presented MPI implementations . . . . . . . . . . . . . . 47

# List of Figures

1.1 Communication component of MPI middleware . . . . . . . . . . . . 5
1.2 IP based implementation of the middleware . . . . . . . . . . . . . . 5

2.1 Overview of an executing MPI program . . . . . . . . . . . . . . . . . 10
2.2 Splitting the communication middleware between the library and the transport protocol. . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

3.1 MPI and LAM envelope format . . . . . . . . . . . . . . . . . . . . . . 19
3.2 Comparison of MPI send/recv and socket library send/recv . . . . . 22
3.3 Motivation for expected queues . . . . . . . . . . . . . . . . . . . . . . 26

4.1 SCTP association versus TCP connection . . . . . . . . . . . . . . . 33
4.2 One-to-many sockets contain associations which contain streams . . 35
4.3 Mapping of SCTP streams to MPI tag, rank and context . . . . . . 36

# Chapter 1

# Introduction

Performance has always played a vital role in the analysis of computer applications. Advances in processor speeds have resulted in a corresponding increase in application performance. In order to gain further performance improvements, computer scientists have researched other optimization techniques than increasing the processor's clock speed. One area of research has involved the use of parallelism. Initially, systems were developed that could themselves automatically exploit parallelism through, for example, the concurrent use of multiple functional units. In addition to systems-based parallel approaches, it has become apparent that further performance improvements could occur through user-level parallelism. An example of this is the use of parallel programming models. Such models let applications be designed so that they can exploit multiple resources at once in a distributed manner.

One parallel programming model is to have independently executing processes, communicating by explicitly passing messages between them. Within the message passing programming model, the Message Passing Interface (MPI) is one API defined to implement parallel applications. It has been in existence for over a decade now. In that time, MPI has become a de-facto standard for message-passing programming in scientific and high-performance computing, which has resulted in the creation of a large body of parallel code that can execute unchanged on a variety

of systems.

Middleware is an essential part of any MPI system. MPI was designed from the onset to be independent from the execution environment. Its communication primitives were designed general enough so that implementations could allow for all possible interconnect technologies. Since no assumptions can be made about the execution environment nor the properties of the interconnect, MPI has needed middleware for mediating amongst the operating system and the interconnect hardware. Overall, the middleware is responsible for coordinating between the application and the underlying components; components exist for starting the application execution, for runtime support with managing processes, and for communicating within the application across different interconnect technologies. The focus of the work in this thesis is primarily on the communication middleware component of the MPI middleware.

The public domain versions of MPI (MPICH [54], LAM [9], and Open-MPI [15]), implement communication components for several interconnects. For example, each have communication component implementations that work on shared memory for SMPs. Also, components exist for specialized interconnects common in dedicated clusters; examples of such interconnects include technologies like Infiniband and Myrinet.

In addition to specialized interconnects, the public domain versions have always supported the IP-based transport protocol TCP in their available communication components. TCP/IP is used to allow MPI to operate in local area networks. More recently the use of MPI with TCP has been extended to computing grids [29], 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.

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 are an active area of research where there is interest in better models [11] and tools for instrumenting and tuning TCP connections [33]. As well, TCP itself continues to evolve, especially for high performance links, with research into new variants like TCP Vegas [14, 35]. 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 widespread use in IP networks. Recently however, a new transport protocol called SCTP (Stream Control Transmission Protocol) has been standardized [49]. 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 type 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, Mac OS X, FreeBSD as well as Sun Solaris 10.

In previous work [26], we investigated the effect latency and loss had on benchmark programs using TCP. After this initial work, we designed a communication component for MPI within LAM atop of SCTP [27] and later implemented and evaluated it, comparing the design and performance to TCP [28]. Our initial experiments were conducted using standard benchmark programs as well as a synthetic task farm program. The task farm was designed to take advantage of our component's specific features which in turn leverage the features offered by SCTP. Our tests introduced loss through the use of Dummynet. Loss in Dummynet resulted in out-of-order packet delivery, a characteristic that could result in real wide area networks during congestion or when intermediate routes change during a flow. Generally, our module's performance behaved similarly to TCP under no loss but as loss increased, the strengths of SCTP versus TCP became apparent showing large performance benefits. In later work [25], we further studied a latency tolerant task farm template and incorporated real applications into its framework so that they could take advantage of the additional features SCTP provides. Using this template, we modified two real applications: the popular protein alignment search tool, mpiBLAST [13], as well as a robust correlation matrix computation. This demonstrated the benefits our SCTP-based communication component can have for real applications under loss and latency.

Using SCTP for MPI has performance benefits[1] but use of SCTP required a different design for the communication component. One repeated theme when concentrating on the design of SCTP-based middleware is how one could push functionality previously within the communication component down on to the protocol

[1]In her masters thesis, my partner Humaira Kamal focused on the performance results of our SCTP-based module. This thesis focuses on the design of the module and comparison to other MPI designs.

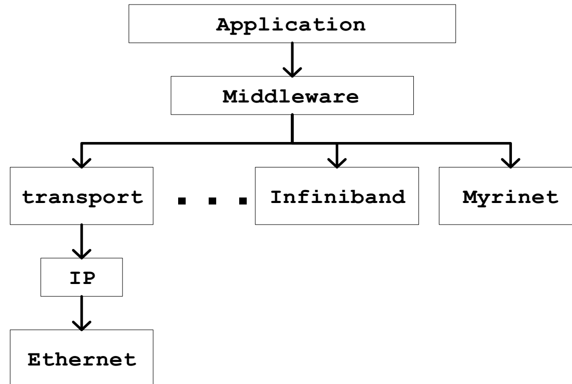

Figure 1.1: Communication component of MPI middleware

stack, effectively thinning the communication component within the MPI middleware. Traditionally, middleware has had their communication component designed as in the hourglass-type network architecture shown in Figure 1.1. In this thesis, we focus on the alternative network architecture for MPI's communication middleware shown in Figure 1.2; such a design attempts to not only thin the communication component, but to eliminate it altogether.

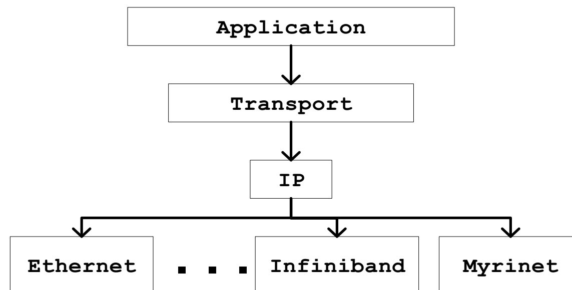

Figure 1.2: IP based implementation of the middleware

Rather than including extensive support for different interconnects in the middleware, we investigated middleware designs that could take advantage of standard transport protocols that use IP for interoperability. One advantage to the network architecture shown in Figure 1.2 is that it simplifies the middleware, which could lead to a more standardized MPI library that could improve the interoperability between different MPI implementations. MPI middleware has never been

standardized and each implementation uses its own design. For example, MPICH's ADI [53, 52] and LAM's RPI [44] describe their layering of the middleware. Open-MPI [16] is a more recent project with the goal of modularizing the middleware to support mixing and matching modules according to the underlying system and interconnect. The use of IP substantially reduces the complexity of designing MPI middleware in comparison to designs that try to exploit the particular features of an interconnect, which is one source of incompatibilities between implementations[2].

The advantage of the usual implementation of the communication middleware component is that it can fully exploit the performance of specialized interconnects. However, these interconnects all have implementations of IP and the affordability and performance of Ethernet make TCP/IP widely used for MPI. With the increased use of more commodity processors and commodity OSes, an interesting research topic becomes how much the use of commodity networking can be exploited if this commoditization trend is to continue. One initial concern is performance. If one compares microbenchmarks, TCP/IP achieves less bandwidth and significantly higher latency than customized stacks and interconnects. But microbenchmarks can be misleading so they should be carefully interpreted; improved bandwidth and latency do not always produce the same gains at the application level. For example, the performance of TCP/IP over Ethernet is within 5% of the performance of Myrinet on the NAS benchmarks and even closer with more efficient TCP implementations [32]. The advantage of using commodity networking (i.e., TCP/IP) is that it can continue to leverage the advances in mainstream uses of networking that share many of the same performance demands as MPI programs.

Given the network architecture shown in Figure 1.2, then it is interesting to investigate re-designs of the middleware that can leverage standardized transport protocols over IP with a view to simplifying and eliminating the communication

[2]Recognizing this is an important problem, other attempts to standardize the interoperability of MPI implementations have also been made such as the industry-led IMPI effort [55].

middleware. This approach differs from the design of current public domain implementations of MPI in that it attempts to push functionality down into the transport layer rather than pull it up into the middleware. For example, OpenMPI sequences all its data within the middleware and stripes it across all available interconnects, managing message assembly and connection failures within the communication component. If one were to limit the communications to those only atop IP, it could be possible to prevent doing such management in the middleware if networking research such as Concurrent Multipath Transfer [24] were exploited, where the data is sequenced and striped within the transport layer instead.

We compare and contrast three different designs of the MPI middleware. We briefly describe an existing TCP-based MPI, we then consider a TCP socket per message stream design, and finally after a brief SCTP introduction, we consider an SCTP-based design. We will compare and contrast the SCTP-based design with the original TCP-based and socket per message stream designs described in the first part of the thesis.

The contribution of this work is that it identifies the extent to which we can take advantage of standardized transport protocols to simplify and eliminate MPI middleware functionality. We discuss the requirements of MPI messaging in terms of demultiplexing, flow control, and the communication management properties of the underlying transport protocol. The socket per message stream design leads to a simple implementation of MPI that completely eliminates the communication middleware layer, however, it does not scale. We show how using SCTP thins the middleware and avoids the scalability problems, however SCTP one-to-many sockets in their current form present limitations that make it difficult to completely eliminate the communication middleware[3]. We discuss what would be required of SCTP one-to-many sockets for full elimination[4].

[3]Elimination is through the use of the more scalable one-to-many socket API described in Section 4.1. SCTP one-to-one style elimination is possible but would have the same problems present in Section 3.3 so it is not examined in this thesis.

[4]A conference paper [39] describing these results will be presented in April 2006 at

In Chapter 2, we look at the MPI library and its requirements for correct implementation. In Chapter 3, we introduce a standard TCP-based implementation and later look at a socket per message stream design comparing and contrasting it with the standard TCP-based design. In Chapter 4, we look at our SCTP-based design of the middleware to see how this brings us closer to the goal of eliminating the communication middleware, and what would be required of SCTP for full elimination. In the end, after surveying related work in Chapter 5, we finish by offering some conclusions in Chapter 6.

---

HIPS(High-Level Parallel Programming Models and Supportive Environments).

# Chapter 2

# MPI Middleware

## 2.1 Middleware

Middleware is an essential piece of MPI. As with any parallel programming model, an MPI application requires a variety of services in order to properly run. These services are each glued together by the middleware [10]; they can be conveniently split into three interacting components:

- a job scheduler to determine which resources will be used to run the MPI job,
- a process manager for process initialization/shutdown, signal delivery, error detection as well as process stdin/stdout/stderr redirection, and finally
- a parallel library such as MPI to provide a mechanism for parallel processes to effectively exchange messages.

Although the focus will be on the MPI library, we briefly discuss the remaining pieces.

## 2.2 MPI Runtime Execution Environment

As shown in Figure 2.1, an MPI message-passing program is a collection of processes executing on one or more processors exchanging messages using an underlying com-

munication medium. The MPI standard was purposely designed to avoid assump-

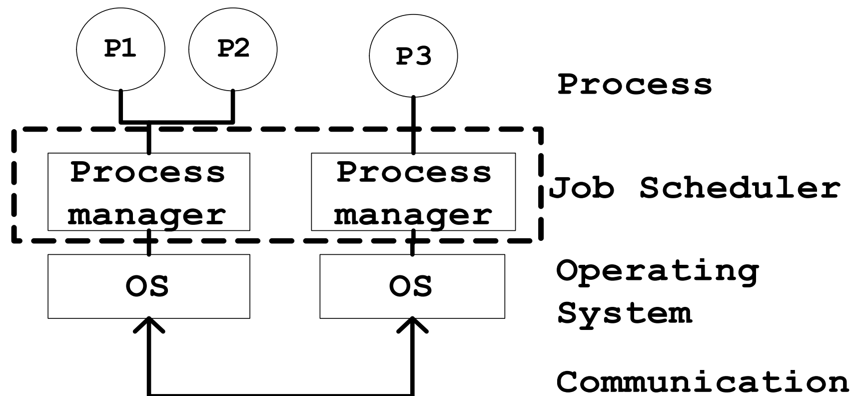


Figure 2.1: Overview of an executing MPI program

tions about the runtime environment. As a result, MPI applications themselves need an environment to run in and thus MPI releases typically provide middleware to manage the interaction of three components: job scheduler, process manager, and the parallel library (MPI itself). Some MPI releases implement the components directly themselves; others make use of externally developed versions of the components and tie their functionalities into their own framework. For example, LAM has implementations of many components including all three of these yet alternatively leverages external components by interfacing job schedulers and process managers like Globus [19], OpenPBS [36], and BProc [5]. For a communications component, LAM implements it as a library linked to by applications as pictured in Figure 2.2; this library makes calls into the other two aforementioned components as well as to another communication component that is within its middleware managing all MPI messages. MPICH and OpenMPI also leverage external components yet similarly always manage the state of MPI messages within communication components in their middleware. All of these MPI releases can run over a variety of interconnects and follow the network architecture that was shown in Figure 1.1. As mentioned, the focus of this work is on the MPI library and the communication middleware that is used to support it.

## 2.3 MPI Library

The MPI library offers a variety of routines to help application programmers effectively develop parallel programs. The middleware exports the API as defined by the MPI standard and the application links to this library. The MPI API routines are typically implemented with the help of other functions which in turn call the appropriate lower layer system and device routines. Often MPI implementations further modularize these other functions into components to allow for maximum flexibility of hardware and algorithm choice. For example, amongst other things, OpenMPI allows you to select at runtime which hardware to run on, which datatype implementation to use, and which reduction operation algorithm to perform.

As shown in Figure 2.2, one typically obtains a layered system where the

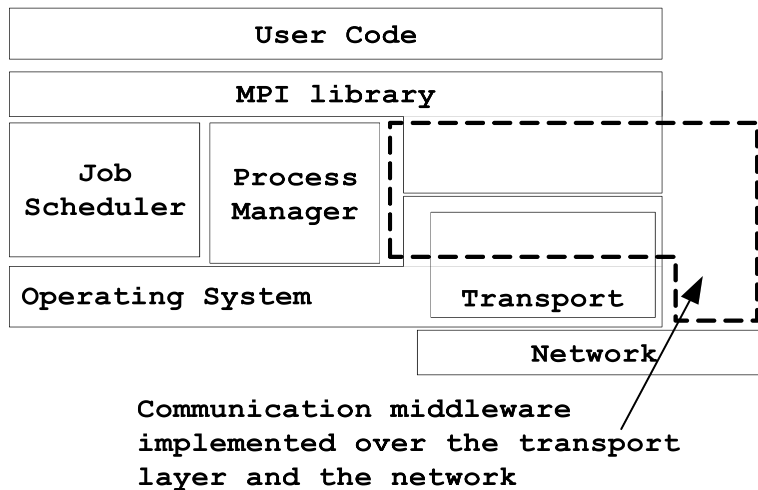


Figure 2.2: Splitting the communication middleware between the library and the transport protocol.

MPI library code calls into lower level modules for job scheduling, process management and communication (represented by the dashed line). The communication middleware consists of one or more modules that call the operating system in the case of TCP and possibly lower level network layer functions in the case of specialized interconnects like Infiniband. Our intent is to remove the direct access to the

network layer and remove the state maintained in the communication middleware by dividing its functionality between the MPI library and an IP transport protocol. The management of active MPI messages and their progress will now be handled by the transport protocol.

The message progression layer within the communication middleware is responsible for progressing messages to ensure that all messages sent are eventually received. The existence of non-blocking communication, in addition to other types of messaging, and the need to match sends and receives makes it difficult to completely execute the MPI communication routine as a simple library function. Typically message progression state has to be maintained in-between MPI calls. One specific question then becomes what state has to be maintained and to what extent can message progression be accomplished by the library versus a middleware layer versus the transport protocol. Another question is when exactly should message progression occur. These questions are answered in the next section. We later eliminate this communications component by storing some state in the arguments passed into the MPI communication routine and then rely on the transport layer protocol to actively progress messages.

## 2.4 Message Progression Layer

The message progression layer copies the message, allocates and frees memory when necessary, enforces the MPI semantics for the different message types, performs message matching, executes the lower level communication protocol, constructs the appropriate message header, and finally sends and receives data from the underlying transport layer.

How the message progression layer is invoked depends on an implementation's interpretation of the Progress Rule for MPI [7]. This rule is defined in the MPI Standard for asynchronous communication operations. Unfortunately, it is ambiguous so therefore its exact meaning is left for interpretation. When a weak in-

terpretation is taken, messages are advanced only during an MPI call implying that this layer is only invoked then. However, with a strict interpretation, the message progression layer can be entered at any point in the application's execution. In this thesis, we focus on implementations that assume a weak message progression rule where messages are progressed only during MPI calls and there is not a separate thread, process, or processor that progresses messages. Additionally, we also do not discuss interrupt driven schemes [51] that use signals or some other type of call-back mechanism.

The various public domain versions of MPI define their interfaces for message progression in a variety of ways. In LAM, there is one layer for this purpose called the request progression interface (RPI), while in MPICH it is a combination of two layers, an abstract device handling MPI semantics and a lower level channel that acts as a byte mover. Similarly, OpenMPI splits the MPI semantics off into its own layer called the PML (point-to-point management layer). However, underneath this layer it has two other layers for message progression, a simple byte transfer layer (BTL) as well as a BTL management layer (BML) that attempts to stripe data across all underlying interconnects([56, 43]).

A communication is initiated in MPI when a send or receive communication routine is called (i.e., posted). The actual completion of the call depends on the semantics of the communication primitive. In the case of non-blocking communication, the call returns immediately and the user must test or wait for the completion of the communication. The send call completes once the user send buffer can be re-used; the receive call completes once the message has been copied into the user receive buffer. In general, the message progression layer manages the life-cycle of a message.

The three main tasks we will focus on in the message progression layer are the message matching, managing of expected and unexpected messages, and the long/short message protocols that are typically used. We will describe the MPI

calls in terms of actions required by the transport layer.

### 2.4.1 Message Matching

Message matching in MPI is based on three values specified in the send and receive call: (i) context, (ii) source/destination rank, and (iii) tag. The context identifies a group of processes that can communicate with each other. These values all appear inside the message envelope, which is part of the header sent from source to destination. Within a context, each process has a unique identification called rank that is used to specify the source and destination process for the message. Finally there is a user defined tag value that can be used to further distinguish messages. A receive call matches a message when the tag, source rank and context (TRC) specified by the call matches the corresponding values in a message envelope. MPI semantics dictate that messages belonging to the same TRC must be received (i.e., completed) in the order in which the sends were posted.

Message matching is a form of demultiplexing that identifies the particular receive call that is the target of the sent message. The destination rank identifies the target process whereas the combination of context, source rank and tag identifies the particular receive call. Because the TRC is provided dynamically at runtime, a receive call can potentially receive any message sent to that destination matching the TRC. There are also wildcards that can be used in a receive call to match messages from any source `MPI_ANY_SOURCE` and/or any tag `MPI_ANY_TAG`. Wildcards provide a form of non-deterministic choice where a process can receive from some collection of messages sent to the process.

### 2.4.2 Expected/Unexpected Messages

One way to view messaging in MPI is in terms of send and receive requests that need to be matched. We ignore the payload for now and also consider non-blocking communication where there can be any number of send and receive requests posted.

Some protocol must be used to ensure that two matching send and receive requests, after being posted, eventually synchronize and complete the communication. Most implementations of MPI do this by managing two structures; one for expected messages (local receive requests) and one for unexpected messages (remote send requests). When the message progression layer processes a receive request, it can search the unexpected queue for a matching request and, if not found, add itself to the expected message structure. In the case of a send request, the message progression layer sends the request to the remote process where the message progression layer on the remote process checks its expected message structure for a match and, if not found, adds the request to the unexpected message structure.

The management of the expected and unexpected message structures is simple in principle but there are several subtle issues that complicate it. First, in order to avoid potential deadlock, the message progression layer must be able to accept all requests. The MPI standard defines a "safe program" as a program that correctly finishes when every communication is replaced by a synchronous communication, which effectively implies that the progression layer needs to be able to accept only a single request. The guaranteed envelope resource protocol (GER) [8] extends this guarantee to a bounded number of requests. Second, because of weak message progression, each MPI call needs to progress messages by updating the state of the expected and unexpected message structures and sending and receiving data to and from the transport layer below. The message progression layer needs to maintain state between the MPI calls since it will need to progress messages for other calls; for example, it needs to keep checking the transport layer for remote send requests from other processes to be matched or added to the unexpected message structure.

### 2.4.3 Short/Long Messages

In terms of network level functions, long and short messages in MPI introduce problems associated with message framing and flow control.

The previous discussion only considers requests and assumes that once requests have been matched the transfer can now be completed. In MPI this is complicated by the fact that messages can in principle be arbitrarily long. Since the system does not have an unlimited amount of buffers, at some point in the transfer, parts of the message must use the user level buffer space given as a parameter to the receive call. As a result, for a sufficiently large message, the communication becomes synchronous where the send call cannot complete before the receive has been posted. Furthermore, depending on the transport layer used, large messages may need to be fragmented and reassembled on the receive side. Fragmentation may also be necessary for fair message delivery to ensure that the message progression layer doesn't block for long periods of time while engaged in the transfer of one large message, especially given that the transfer could occur unexpectedly (i.e., advanced by the message progression layer) on any MPI call.

Given that the message progression layer must accept requests and also manage the transfer of large messages, then a natural solution is to handle short and long messages differently. Short messages are bundled with the request and thus can be immediately copied into the user's memory when matched. Long messages, which may require a synchronous communication, use a rendezvous protocol that first matches the two requests and then arranges a transfer of the message from the user's send buffer on the remote side to the user's receive buffer. Thus, short messages are sent eagerly in most MPI implementations, whereas long messages use a rendezvous [6]. There is an additional benefit to this approach for performance. As Gropp describes for MPICH [53], if one considers memory copying and the cost of the rendezvous, then when message copying costs exceed rendezvous costs it is more efficient to use rendezvous, assuming it can avoid the extra copying, rather than using eager send.

# Chapter 3

# TCP Socket-based Implementations

In the first part of this chapter, Section 3.1, we focus on the LAM-TCP RPI in order to illustrate a typical implementation of the message progression layer. In the second part, Section 3.2, we describe a TCP-based implementation of the MPI library that eliminates the message progression layer altogether, following Figure 2.2. By elimination, we mean that such a layer is no longer necessary to maintain the hidden state required to progress messages. Since messages still need to be advanced, we eliminate the message progression layer by moving some functionality into the MPI library routines and the remainder down into TCP as illustrated in Figure 2.2. Although this design is not very scalable, for reasons to be discussed, it serves to illustrate the features that are needed in the library and transport layer when having an MPI implementation where the message progression layer is eliminated. After sketching this design, in Section 3.3 we discuss problems and possible techniques to alleviate some of these problems.

## 3.1 The LAM-TCP Message Progression Layer Implementation

LAM-MPI is a popular open source implementation of MPI. It supplies message progression layers for various interconnects, one of which is TCP. Our work in [27] and [28] involved carefully analyzing this particular TCP message progression layer (referred to as LAM-TCP from here forward); we summarize LAM-TCP here to demonstrate a typical message progression layer implementation.

When an MPI application begins, each MPI process executes `MPI_Init()`. Within this call for LAM-TCP, each process initializes a `struct` that contains the state between itself and each other process. A given process will know its own rank and how large its `MPI_COMM_WORLD` is. Other than setting initial states between itself and other processes, it also connects to each other process by way of a socket. This is accomplished by opening a listener port and then making this port number available by way of the out-of-band daemons offered by LAM[1]. If a given process is creating a connection with a process whose global rank is less than its own, then it acts as a client issuing a `connect()` call to the other process's listening port. On the other hand, if the other global rank is greater, then the lesser ranked process will issue an `accept()` creating a new connection on the resulting socket and port once the client connects.

At the end of `MPI_Init`, each of the `N` processes has opened `N-1` additional TCP sockets; the total topology is fully connected from the beginning of the MPI application. The listening socket is then closed. While performing connection establishment, a global map is created going from each socket's file descriptor value to the other process's global rank value. This map is an array of size `FD_SETSIZE` which is typically defined to be on the order of 1024 for standard users on most systems.

[1]The LAM daemons serve the role of the job scheduler and process manager components introduced in Chapter 2.

Having outlined how connections amongst processes are established in LAM-TCP, next we consider the message progression layer functionality with respect to its main tasks described in Section 2.4.

### 3.1.1 Message Matching

When messages are passed from one process to another, they are sent together with the appropriate control information, or envelope, containing information about the associated message. As mentioned earlier in Section 2.4.1, message matching in MPI is based on three values specified in the send and receive call: (i) context, (ii) source/destination rank, and (iii) tag. In LAM, as shown in Figure 3.1, these are included in the envelope in addition to the message length, a sequence number, and a flag field that indicates what type of message body follows the envelope. Each of these fields are 32 bits in LAM. Not all MPI implementations use the same envelope structure. This is in part because different interconnects can specify more or less information. The Interoperable MPI (IMPI) project [55] attempts to standardize the envelope packet format so that MPI implementations can interact.

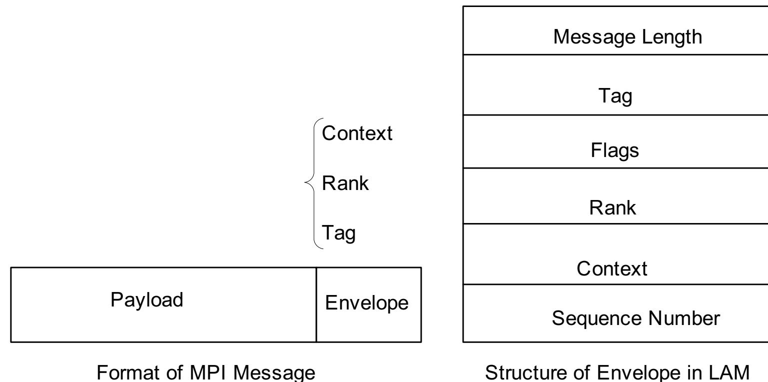


Figure 3.1: MPI and LAM envelope format

Since there is a connection per MPI process pair, for a standard receive call it is clear where the progression layer should read a message from: the appropriate socket for the source rank. The information read from the envelope can be compared

to the tag, rank, and context (TRC) of the receive call. The presence of a wildcard for the tag (`MPI_ANY_TAG`) doesn't differ from the standard case except that the arrived message's tag value is not important. However, the presence of a wildcard for the source (`MPI_ANY_SOURCE`) complicates matters. With such a wildcard, messages from any of the ranks in `MPI_COMM_WORLD` are legitimate. The implementation handles this by adding all sockets to a set and then calling `select()` on that set.

### 3.1.2 Expected/Unexpected Messages

As introduced in Section 2.4.2 for general message progression layer implementations, the expected and unexpected queues are also vital within LAM-TCP. For LAM-TCP, the queues are each implementated as a linked list within the message progression layer. When a request is posted, it is added to the end of a linked list of expected messages which have not arrived yet. When the progression layer is entered, the expected message queue is traversed and the socket associated with each request's rank is inserted into one of two file descriptor sets, depending on whether the request is for a send or receive operation. In the end, if the sum of the sets' sizes is greater than one, the sets are passed in to a `select()` call and afterwards the appropriate `read()` and `write()` calls are made based on its results. If the sum of the sets' sizes is one then no `select()` call is necessary so the one call can operate directly on the socket. When the progress layer is entered, then `setsockopt` is used prior to these I/O calls to set the socket to blocking or non-blocking depending on the whether the MPI call was blocking or non-blocking.

In LAM-TCP, all tags are sent over the same connection. The ordering semantics of MPI could result in a scenario where a socket is being read for a message with one tag but the obtained message envelope shows that it contains a different tag that has yet to be pre-posted. Here, this message is considered unexpected. When a message is unexpected, it is buffered and stored inside an internal hash table. Later, whenever a new request is posted, before being added to the expected queue, it is

first checked against all of the buffered unexpected messages for a possible match, in case the message had already arrived.

Unexpected messages can arrive other ways too. For example, collective operations share the same connection but have a different context than point-to-point messages; when expecting one context, it could receive another. Similarly, if `MPI_ANY_SOURCE` is used for a specific tag and it is the only outstanding request, data that has arrived on any connection for any tag will be read since the `select()` call will return their sockets as ready to be read. Messages read for tags other than the specified tag will be considered unexpected.

### 3.1.3 Short/Long Messages

LAM-TCP splits the messages into three types internally: asynchronous short messages which are of size 64K bytes[2] or less, synchronous short messages, and long messages which are greater than 64K. The type of message is specified in the flag field in the LAM envelope (see Figure 3.1).

Short messages are passed using eager-send where the message body immediately follows the envelope. When received, they are handled as mentioned in the previous section depending on whether or not it is an expected or unexpected message. However, while synchronous short messages (i.e., `MPI_Ssend`) also communicate using eager-send, the send is not complete until the sender receives an acknowledgment (ACK) from the receiver. The state of these short protocols is maintained within the message progression layer.

Long messages are sent using a rendezvous scheme to avoid potential resource exhaustion problems and also in order to have the potential of a zero-copy as mentioned in Section 2.4.3. 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 ACK to the sender to indicate that it is ready to receive the message

[2] 64K bytes is the default cutoff point however this can be specified by the user at runtime.

body. Once the ACK is received, the long message is sent. If no matching receive request 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. Again, the state of the rendezvous is stored within the message progression layer.

## 3.2 Socket per Message Stream MPI Implementation

Now that we've introduced a typical implementation of a message progression layer, we'll describe a TCP-based approach of the MPI library that attempts to eliminate the message progression layer altogether.

The standard socket library provides routines that can be used to implement the MPI communication primitives. Figure 3.2 compares the standard MPI send and receive calls to the corresponding socket library calls. Some of the differences

```
int MPI_Send(void* msg, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
int MPI_Recv(void* rcvBuf, int count, MPI_Datatype datatype,
       int source, int tag, MPI_Comm comm,
       MPI_Status *status)

int send(int socket, const void *msg, int msgLen, int flags);
int recv(int socket, void *rcvBuf, int bufLen, int flags)
```

Figure 3.2: Comparison of MPI send/recv and socket library send/recv

can be easily accommodated. For example, message length is expressed in terms of MPI types, which within the MPI library can be converted into its actual byte length using packing information contained in the `MPI_Datatype`. The other parts of the message are the communicator, source/destination, and tag fields. The design attempts to map these into a socket descriptor. As a result, every TRC will have its own socket and TCP connection, which is consistent with the message ordering semantics of MPI.

In general, since tags are user defined variables, sockets will need to be created dynamically when communication occurs. Although this may require a large number of descriptors, it is possible to mitigate this problem by closing connections that are not being used. Typically, depending on the program, there are few TRC combinations active at any one time.

The `MPI_Init()` routine can be used to exchange the information necessary to initiate the system. In particular, at a minimum, at runtime we need the IP addresses of all machines running MPI processes. We assume there is a pre-assigned control port for each MPI process. This is used for the `accept()` socket call when creating new connections initiated by a `connect()` call to that port from another MPI process. When a new connection is created on this control port using `accept()`, a new socket using its own port is created and the control port can continue to be reused as a control port to initiate other connections. The control IP and port are made available for each rank by way of the LAM out-of-band daemons. The `MPI_Init()` routine creates the `MPI_COMM_WORLD` communicator, an opaque data structure, which for each process, contains the table of the machines in `MPI_COMM_WORLD`. More specifically, the table gives a mapping from MPI ranks to IP addresses and control port. Since the communicator object is an argument to all communication calls, every call has access to this table.

As well as the rank-IP table, communicators will also maintain a separate "connections" table that is a mapping from a TRC to a socket descriptor. Each MPI send or receive uses the TRC value as a key to find the corresponding socket for the send and receive socket call. If there is no entry for that key, then it uses the rank to determine the IP address and port. The control port is then used to connect to the remote machine and create a new connection for that TRC. To create a new connection, one end must execute `accept()` whereas the other end must execute `connect()`. Because of wildcards, the receive side executes the `accept()` and the send side executes a corresponding `connect()`. Note, this is only done for the first

connection.

Having sketched out the basic scheme, we will now consider how the message progression layer functionality requirements described in Section 2.4 will be handled in this socket per stream implementation of the MPI Library that has eliminated the message progression layer.

### 3.2.1 Message Matching

The basic scheme works for the standard send and receive since in this implementation, a TRC defines a message stream in MPI and a TCP socket per TRC provides a full ordering between endpoints. However, matching is complicated by the existence of wildcards. Wildcards are only used in MPI receive calls and are `MPI_ANY_SOURCE` and/or `MPI_ANY_TAG`. The `select()` socket call allows us to create a set of socket descriptors that can be used to block while waiting for data on one of the connections. The MPI library code scans the connections table to create a set of sockets that can match the receive and can then be used in a `select()` call.

It is possible that there is a matching TRC for which a connection does not yet exist so a new connection will need to be established. New connections can be handled by adding the socket associated with the control port to the `select()` call. However, there may be connect requests from one or more processes not associated with the receive call. As a result, once a new connection is accepted, the TRC information from the remote send side needs to be sent to the receive side in order for it to process the connection request and update the connections table associated with the communicator. This was the rationale for having receive do `accept()`, since these calls can receive from multiple processes and need to use `select()`, whereas the sends do not use wildcards and they either already have a pre-assigned socket or the send needs to create a new connection using `connect()`.

The receive call that executes the `select()` may be notified of data on an existing connection or notification for a new connection, which must be added to the

connection table. Depending on the implementation, it is possible that the request is for a different communicator, which implies that the tables for `MPI_COMM_WORLD` and all communicators derived from it must be accessible. An alternative approach exists that avoids requiring access to all communicators. When creating a new communicator, an old communicator is provided to the `MPI_Comm_create` call; the new communicator is some subset of the old communicator. Creating a new communicator is a global operation over the ranks in the old communicator, even if they are not going to be within the resulting new communicator. As a result, it is during this global operation that it is possible to negotiate a new control port to be stored for each communicator. This eliminates the possibility of receiving a connection request for a communicator different from the one specified in the receive call since every communicator will have its own control port and communicators can never be wildcarded.

### 3.2.2 Expected/Unexpected Messages

The global expected and unexpected message structures are no longer needed under this design. Because each MPI message stream (TRC) now has its own connection, for the unexpected queue, the implementation relies instead on the socket buffers and the TCP flow control mechanism to manage the flow of messages on a connection once it has been created.

Non-blocking is possible with sockets by setting the non-blocking option on the socket (`O_NONBLOCK`). For most socket libraries, it is possible to set the option on a per call basis, which corresponds to MPI, rather than having it as part of the connection [46]. Non-blocking communication in MPI returns an opaque message request object (`MPI_Request`) that is then passed to all calls that check for completion. The request objects can be used to maintain the information about what is needed to complete the MPI call. For example, in the case of a synchronous communication, the request object stores whether the message corresponding to the

send or receive has been posted.

The presence of blocking and non-blocking calls in MPI creates potential scenarios where the state of a particular message's progress will be dependent on the progress of other previously posted messages. As a result, a local form of the expected message queues must be kept. To motivate the necessities for these queues, let's assume we had no expected message queues and we had the following scenario illustrated in Figure 3.3. Say messages A and B were successfully put on the network

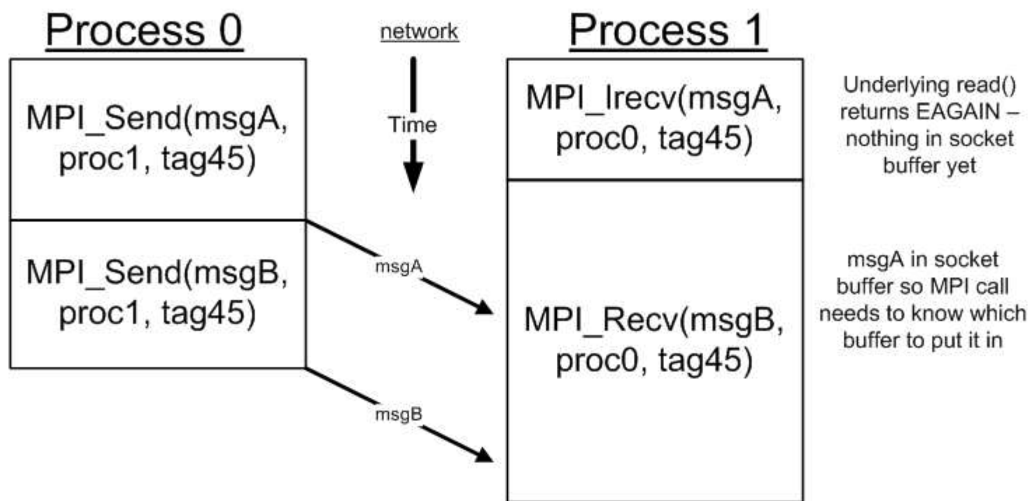


Figure 3.3: Motivation for expected queues

from process rank 0 to process rank 1 on tag 45 using blocking sends. Process rank 1 has a non-blocking receive for message A and a blocking call for message B. When process rank 1 pre-posts a request for message A, if that message isn't in the socket buffer, an `EAGAIN` would be returned and the application would proceed. Next, during the blocking receive call for message B, process rank 1 is going to have to be aware of the request for message A since it is going to arrive to the socket first due to TCP's full sequencing of a connection. The library must know which buffer to put it in since in this design, no global expected queue exists [3].

In order to process requests correctly, another table is maintained within the

[3]This example may seem contrived but the same scenario could be illustrated with any combination of non-blocking calls together with `MPI_Wait`.

opaque communicator object. This table maps a TRC value to the head of a linked list of request objects. LAM-TCP uses an expected message queue with messages from all TRCs in it. In this socket per TRC design, effectively the table contains more local expected message queues. A linked list itself is outstanding requests on a given TRC; due to MPI semantics, these requests must be completed in order. For receive requests, access to this linked list is required so that data obtained from the socket buffer can be placed in the correct user-space buffer as specified in the posted MPI receive call. For send requests, messages need to be sent in the order in which they were posted so this linked list is also required to be available. This table is essentially an expected queue within the opaque object, the only difference being that instead of a linked list of requests for all TRCs as in standard message progression layers, it is a table indexed by the TRC returning a smaller, more local linked list.

### 3.2.3 Short/Long Messages

In this design, a connection corresponds to a particular TRC. TCP's in-order semantics ensure that messages are completed in the order they are sent. As well, TCP's flow control mechanism ensures that the eager sending of messages will not over-run the receiver's buffer and exhaust memory resources. This eliminates one of the motivations for short and long messages and the use of a separate protocol for each kind. The important point is that we can take advantage of TCP's flow control mechanisms.

This design simplifies the middleware since it is no longer necessary to have a short and long protocol that restricts the eager sending of larger messages. In the message progression layer, flow control is handled by complicating the protocol, restricting the eager sending of large messages. Memory resources in our case are allocated on a per TRC basis in the socket buffer and not on a per message basis in the middleware.

It is still necessary to do message framing for TCP streams to determine message boundaries since otherwise one send may receive part of a message destined for some other user level receive buffer with the same TRC. This is accomplished by passing the envelope first, which contains the message's length.

## 3.3 Critique of the Socket per Stream Implementation

There are several issues that arise in using a socket for each TRC. These issues generally arise in any TCP implementation of the middleware, including the standard TCP implementation used in public domain implementations of MPI([34, 28]). The major issues are the following:

- limitations on the number of sockets,
- large number of system calls,
- the `select()` system call, and
- TCP flow control.

### 3.3.1 Number of Sockets

The number of allowed file descriptors is determined by `FD_MAX` (typically 1024 on Linux). On larger systems, even the standard TCP implementation can exhaust the supply of file descriptors, when it opens one socket for each MPI process in the system. For this reason, both MPICH and OpenMPI open connections as they are needed rather than opening all connections during a call to `MPI_Init()`, as is the case for LAM-TCP. The socket per TRC design makes this problem even worse. The design may be acceptable for small clusters, but has difficulty scaling to a large number of processes or with applications that may use a large number of tags and contexts. For example, consider a simple farm program with a master and `N-1` workers. The master would need `N-1` sockets to communicate between the workers.

Typically, three tags are used with task farms: task requests, tasks, and results. Thus, a simple farm would require `3(N-1)` sockets.

The number of file descriptors can be reconfigured in the kernel, however, there is also a significant memory cost associated with each TCP connection, which again prohibits having large number of connections. Recent work by Gilfeather and Maccabe [18] addresses the problem of the scalability of connection management in TCP and introduces some techniques for alleviating these problems in clusters. The origin of their work was from similar problems that arise in web servers, which also have to manage a large number of connections and where techniques have been developed to efficiently handle these connections. This work is a good example of an advantage of using commodity transport protocols like TCP for MPI since one can take advantage of state-of-the-art research on other applications (like web servers) that place similar demands on the underlying operating system.

Even with scalable connection management, there is no easy way to know when a connection is no longer needed. A simple approach would be to time out connections, or to release them shortly after a new context is detected. But again, tags that are used rarely or for a short amount of time would result in extremely large communication times waiting to create connections. For example, if tags are used as job numbers in a task farm, then it is quite possible that every message in the system results in a new connection.

### 3.3.2 Number of System Calls

The second issue is a performance issue that arises because of the way in which the middleware must poll the sockets that results in a large number of system calls. As described in [34], the message progression layer usually needs to execute a `select()` call to determine the next socket with available data and then needs one or more `read()` calls to obtain the message. The same is true of the implementations outlined above. This results in several system calls for each MPI call, which not only requires

a context switch, but also extra processing because each system call is a call into the kernel [46]. Matsudo et al. [34] addresses this problem by managing the expected and unexpected message structures inside the kernel.

### 3.3.3 Select System Call

A third issue is the performance of the `select()` call, which is known to scale linearly with the number of sockets. For example, as shown in Matsudo et al. [34] on a 3.06GHz dual Xeon machine, the cost of `select()` increases linearly from under 100 microseconds for less than 200 sockets to 900 microseconds for 1000 sockets. Newer system calls such as `epoll()` for Linux, and similar calls in other operating systems, have tried to improve the event handling performance of web-servers [17]. The socket per TRC design still requires frequent `select()` calls particularly with MPI calls having wildcards.

### 3.3.4 Flow Control

The final issue is flow control in TCP, which is a concern in any TCP-based MPI implementation. Since buffering is being done by the transport layer, it is possible that when MPI receive calls are delayed, the socket buffers fill and as a result trigger TCP flow control to close the receive window. Closing and opening the advertised receive window in TCP restricts bandwidth because of TCP built-in timers and slow-start mechanisms. This is particularly serious for high bandwidth connections where only a small fraction of the bandwidth could be utilized.

There is a mismatch between the event driven operation of the transport layer and the sequential control operation of the MPI program. Flow control at the transport layer attempts to match the sender's data rate to that of the receiver. Messages in MPI are bursty and can easily trigger TCP's flow control mechanism. Flow control is an end-to-end mechanism and thus not only reduces bandwidth but also significantly adds to message latency. The standard TCP implementation of

the progression layer constantly empties the socket buffers to reduce the chances of it filling. Also, since all traffic between two processors is aggregated on a single connection it tends to smooth out some of the burstiness. The socket per TRC does not aggregate traffic and is far more likely to trigger TCP flow control.

On the other hand, our socket per TRC design does make it possible for the MPI program to take advantage of TCP's flow control mechanism since it possible to push back on an individual flow by delaying receiving data on a particular socket. Under this design, flow control allows the user to configure and throttle the data rates on a particular TRC. However, the end-to-end nature of TCP's flow control mechanism makes such fine-grain flow control expensive in high latency network environments.

For LAM-TCP, research has been conducted where they have flow control mechanisms within the message progression layer itself rather than only using the flow control of the underlying transport protocol [8]. Such schemes attempt to do a user level version of push back on message flows in order to guarantee resources and to prevent exhaustion. By pushing this functionality down into TCP, the socket per TRC design prevents the MPI implementor to have to design a user level flow control mechanism and instead lets them leverage the transport layer's flow control mechanism, including potential advances that may occur here by research in the networking community.

# Chapter 4

# SCTP-based MPI Middleware

SCTP is a general purpose unicast transport protocol for IP network data communications; it has been recently standardized by the IETF [49, 57, 48]. SCTP is well suited as a transport layer for MPI because of some its distinct features not present in other transport protocols like TCP or UDP. In this chapter, we first give an overview of the SCTP protocol in Section 4.1. After this, we present the implementation of an SCTP-based message progression layer in Section 4.2. We show that use of SCTP one-to-many sockets thins the message progression layer, taking advantage of SCTP's additional features. We show that in order to completely eliminate the progression layer's functionality using a one-to-many socket, additional features of SCTP would be required.

## 4.1 SCTP Overview

SCTP is a general purpose unicast transport protocol for IP network data communications [49]. 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.

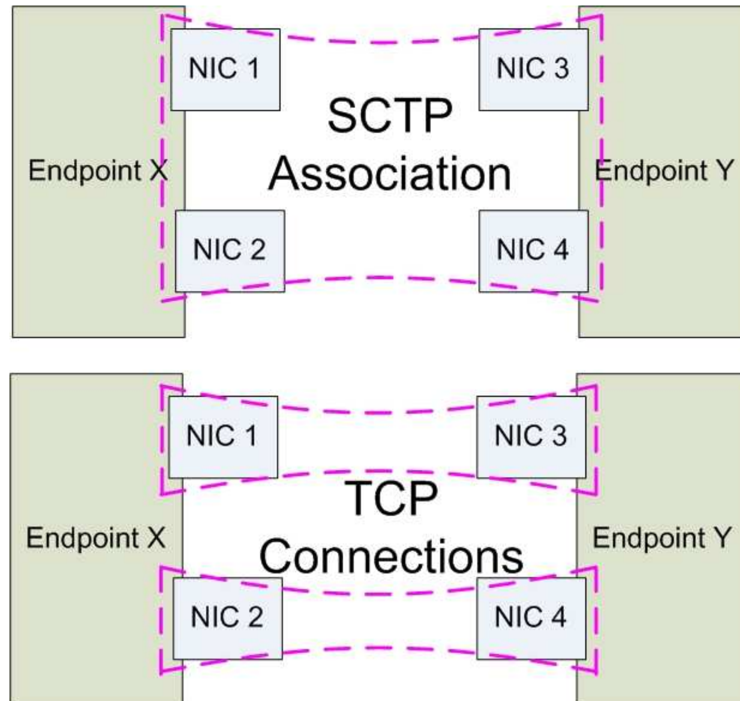


Figure 4.1: SCTP association versus TCP connection

Like UDP, SCTP is message oriented and supports the framing of application data. However, more like TCP, SCTP establishes a reliable session between a pair of endpoints. SCTP calls its sessions *associations* whereas TCP terms them *connections*. There is however a significant difference between a TCP connection and an SCTP association. As shown in Figure 4.1, an SCTP association can simultaneously be between endpoints with multiple IP addresses, i.e., multihomed hosts. On the other hand, a TCP connection is only capable of being between exactly one pair of IP addresses.

In SCTP, if a peer is multihomed, then an 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. Currently SCTP 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 [24, 23], and this functionality may become part of the SCTP protocol.

In addition to the multihoming feature SCTP offers, there is also a multistreaming feature where it is possible to have multiple logical streams within an association. Each stream is an independent flow of messages which are delivered in-order. If one of the streams gets delayed, it doesn't affect the other streams so, in other words, it avoids head-of-line blocking.

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 that operate in parallel. Each connection would then be analogous to an SCTP stream. Having parallel TCP 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 [20]. SCTP does precisely this by ensuring that all the streams within a single SCTP association share a common set of congestion control and flow control parameters. It obtains the benefits of parallel TCP connections while keeping the protocol TCP-friendly [41].

SCTP supports both one-to-one style and one-to-many style sockets. A 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 with the

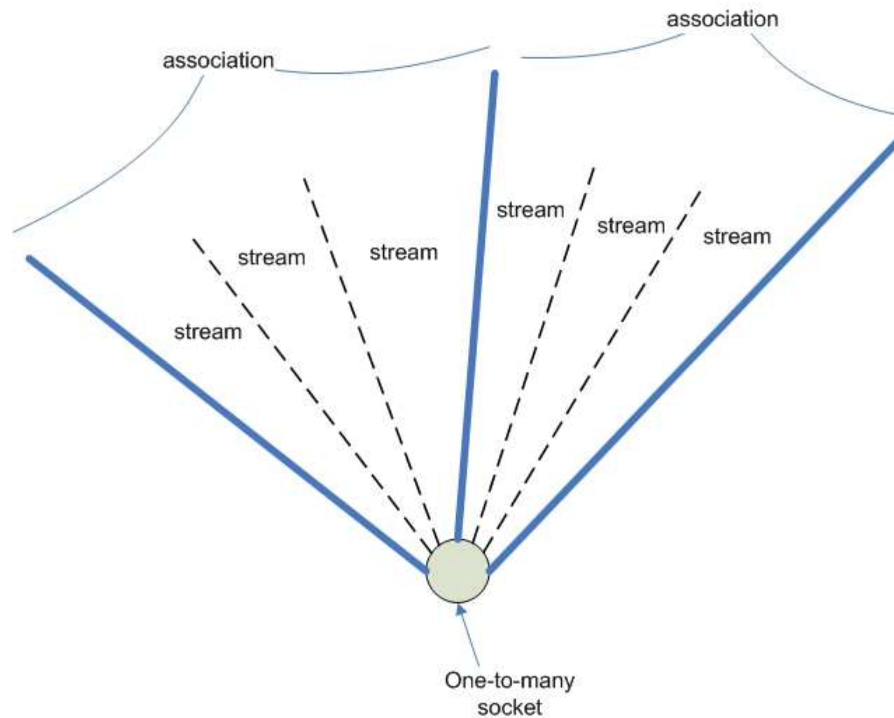


Figure 4.2: One-to-many sockets contain associations which contain streams

added benefit of consisting of reliable associations. A one-to-many socket with two associations is depicted in Figure 4.2.

Figure 4.2 shows how multiple associations and their contained streams are all funneled into a single one-to-many socket. The SCTP API provides a `sctp_recvmsg()` method that obtains data from the one-to-many socket. It returns a message and a populated `sctp_sndrcvinfo struct` that contains information about which association and stream the message came in on. The API does not provide a means of specifying which association or stream to obtain a message from. So for applications that previously used multiple TCP sockets, porting it to use a single one-to-many socket requires more of an effort because the application needs to be more reactive since the `sctp_recvmsg()` call itself does not let you specify exactly where to get the next message from.

## 4.2 SCTP-based Implementation of the Message Progression Layer

We have implemented a version of the message progression layer, as a new RPI for LAM, that takes advantage of SCTP's new features [28]. Here, we briefly describe our implementation and compare it to the middleware implementations described previously.

In SCTP, each association between endpoints can have multiple streams. We take advantage of streams in our SCTP-based implementation of MPI by mapping different MPI context and tag combinations onto different streams (as shown in Figure 4.3). The use of an association for each rank and a stream for each context

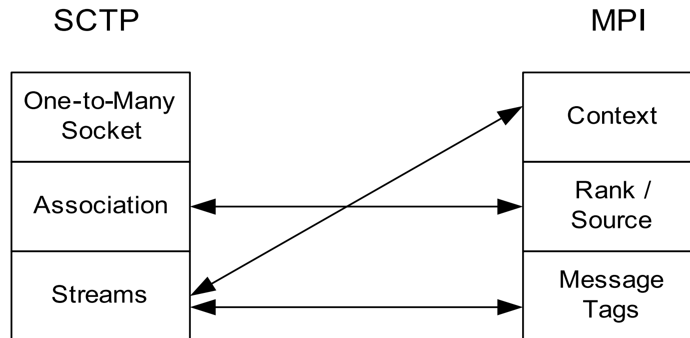


Figure 4.3: Mapping of SCTP streams to MPI tag, rank and context

and tag combination results in a mapping of each TRC to its own SCTP stream. This uses only a single one-to-many socket. The mapping of each TRC to its own stream is semantically equivalent to the mapping of TRCs to TCP connections discussed in Section 3.2. Both satisfy the MPI semantics of in-order message delivery, however, SCTP avoids many of the disadvantages discussed in Section 3.3.

In the remainder of this section, we discuss our implementation of the message progression layer with respect to message matching, expected and unexpected messages, and the short/long message protocol outlined in Section 2.4. Effectively, under this design some of the traditional features of a message progression layer are

pushed down into the transport layer. We show what features would be required of SCTP if one-to-many sockets were to be used to fully eliminate the message progression layer, splitting its traditional functionalities into both the library and the transport protocol.

### 4.2.1 Message Matching

In SCTP, we used a one-to-many socket that is similar to a UDP socket. Using socket calls, it is possible to specify the association (destination machine) and stream on which to send messages. Again, just as in the case of one socket per TRC, we can store this information in the communicator that is a parameter to each send call. Ideally, on the receive side we would like to receive the next message on a given association and stream; this is because TRCs are mapped to streams which maintain the order which messages were sent, adhering to MPI messaging semantics. However, SCTP does not support this functionality for streams or associations in the one-to-many style and simply returns the next available message on any association and any stream.

Semantically, the provided `sctp_recvmsg()` socket call is equivalent to an `MPI_Recv` with `MPI_ANY_RANK` and `MPI_ANY_TAG`. As a result, for MPI receive calls that do not use the wildcards, it is necessary to do message matching. It is thus not possible to eliminate message matching from the design without maintaining global expected and unexpected message queues.

SCTP one-to-many style does simplify the socket calls needed to receive messages since no `select()` call is required because only one socket is used; implementing it using a single one-to-many socket makes the message progression layer thinner. Each `sctp_recvmsg()` socket call returns a full message that has either to be delivered or added to the unexpected message structure. In comparison to the standard TCP implementation introduced in Section 3.1, it avoids the multiple system calls needed to read a message. In comparison to the socket per TRC design

in Section 3.2, it avoids the costly `select()` call.

### 4.2.2 Expected/Unexpected Messages

Since we need to do message matching, our message progression layer still needs to maintain expected and unexpected message structures. Each of these queues needs to be global across all TRCs because the SCTP API does not let one specify the stream and association on the `sctp_recvmsg()` call. In the socket per TRC design, these queues were eliminated from within the progression layer; their functionalities were effectively split. For the expected message queue, a more local form of the queue was pulled up into the MPI library, having a hash of queues keyed by the TRC. The ability to fully specify the TRC within the `recv()` call was a trait of the transport protocol so expected message queues were split between the MPI library and the transport protocol. For unexpected message queues in the socket per TRC design, they were pushed fully down onto the transport protocol. Traditional designs such as LAM-TCP place the queues in the message progression layer. In any design, while possible to maintain these queues in the MPI library and attach them to a communicator, their mere existence and necessity shows that progression layer elimination with SCTP using a one-to-many socket can not push as much functionality down onto the transport as is possible in the socket per TRC design unless the SCTP API provided additional capabilities; these capabilities include the ability to receive messages on a particular association and stream as well as a means to ask the socket which streams currently have data that can be read without blocking (i.e., something similar to the functionality of `select()`).

Although, every `sctp_recvmsg()` call returns a message, we continue reading from the socket if there are outstanding requests in order to obtain and deliver as many messages as possible. Like TCP, SCTP has a socket option that can be set so that I/O functions called on that socket return immediately when the socket call would block (returning `errno EWOULDBLOCK`); for non-blocking MPI calls, we

can exit the message progression layer when this `errno` is returned. In addition, the SCTP receive socket buffer size can be set by the user, but all streams and associations share the same buffer, congestion control settings, and flow control settings. To avoid triggering the flow control mechanisms, it is better to empty the buffer as often as possible; this is why we read as much as we can from the socket when we enter the message progression layer. Given no other user input, this design decision is likely the best thing since it progresses all messages that are already sitting at the receiver.

### 4.2.3 Short/Long Messages

The need to manage unexpected messages implies that we will need a short and long message protocol to avoid resource exhaustion.

The first issue that arises is handling messages which exceed SCTP's socket send buffer size. Messages that exceed this size need to be fragmented and re-assembled by the message progression layer. We use a rendezvous mechanism where, once the rendezvous has occurred, the sender fragments the MPI message into fragments and the receiver assembles these fragments in the user's receive buffer as specified in the MPI receive call. The sender could loop sending each fragment, however, this does not allow other messages to advance. As a result, we used round robin scheduling of all out-going SCTP messages and recorded their progress in a structure pointed to from the `MPI_Request`. This allows MPI messages with different TRCs to progress concurrently on an association between two machines. Special care had to be taken to eliminate race conditions that could occur when MPI messages had the same TRC [28].

The second issue is flow control. As previously mentioned, it is not possible in SCTP to request the next message on a particular stream and association. A reason for not providing this functionality is that streams share the receive socket buffer inside the transport layer. Providing a mechanism for obtaining messages on

a particular stream introduces the possibility of messages not being selected and eventually exhausting the socket buffer resulting in deadlock. The potential for deadlock would be difficult to detect because it depends on the possible ordering of data segments received by the transport layer.

The socket per TRC design was able to eliminate the short/long message protocol because it could rely on TCP's flow control mechanism. Although the consequences of closing the advertised receive window makes the use TCP's flow control costly (as discussed in Section 3.3.4), a push back mechanism would give the MPI program control over which message to advance. This would allow users the ability to advance messages on its critical path and potentially improve the overall execution time of the program.

SCTP cannot provide the level of fine grain flow control that would be necessary to allow the user to push back on a particular stream. SCTP streams and thus TRCs share flow control values. As a result, one TRC can affect the performance of another. To try to combat this, it is possible in the SCTP one-to-many socket style to allocate socket buffer space on a per association basis, which would give more control over flow control from a particular machine, but there still is no call provided in the API to tell which associations or streams have data ready to be read.

# Chapter 5

# Related Work

Computing systems in the Top 500 list [50] increasingly are clusters which use commodity parts such as operating systems and processor hardware. In the past, the list had previously been occupied more by massively parallel processing (MPP) systems, each with a set of customized parts. Using components from various vendors, clusters present a cost effective alternative. For example, the share of clusters in November 2000 was a mere 5% but it was well over 40% by November 2003. In this same span of time, the share of MPP systems dropped from 69% to 33%. Generally this shows a trend towards commoditization.

This thesis explores the scenario where advances in commodity networking are to be made, and what MPI implementation re-designs could be made to better leverage commodity networking. Use of the transport protocols atop IP for MPI is quite common however each design and implementation is unique to the MPI distribution. For example, the public domain versions of MPI each have their own components and interfaces. LAM defines its main component interface through its System Services Interface (SSI) [45]. The component performing message progression within SSI is the Request Progression Interface (RPI) [44]. A TCP implementation comes with the distribution and in previous work, we have developed an RPI for SCTP [28]. For MPICH, a TCP component exists within its defined compo-

nent interface called ADI, or Abstract Design Interface [53]. Under this framework, message progression is achieved through a combination of two layers, an abstract device handling MPI semantics and a lower level channel that acts as a byte mover. Similarly, OpenMPI [16] under its MPI Component Architecture (MCA) splits the MPI semantics off into its own layer called the PML (point-to-point management layer). However, underneath this layer it has two other layers for message progression, a simple byte transfer layer (BTL) as well as a BTL management layer (BML) that attempts to stripe data across all underlying interconnects [56, 43]. A TCP implementation of the BTL comes standard with the distribution.

Each distribution has made independent component interface design decisions, and the costs of their decisions have been analyzed. In [21], they implement MPI over Infiniband at various layers within MPICH2. They found substantial performance benefits when eliminating layers. On the other hand, in [3] they look at the component costs of the MCA within OpenMPI. In this work, they found that eliminating layers did not result in any substantial performance gain. The reasons why these two papers appear contradictory are yet to be explained and remain a topic for future work.

The question remains where definitively should features be placed within a given MPI implementation. One feature that is placed in a variety of levels is the ability to stripe data across multiple networks at once. Within OpenMPI, this is handled within the middleware [56] for all interconnects. An MPICH variation called MPICH-VMI [38] discusses the ability to stripe data using TCP when the OS and network both support channel bonding [42]. In our work with MPI over SCTP [28], data striping could occur solely within the transport protocol by utilizing Concurrent Multicast Transfer([24, 23]).

Another feature that can be placed at different levels is flow control. In this thesis, we showed a design with a TCP socket per message flow i.e., messages that share the same tag, rank, and context, or TRC. This effectively means that the TCP

flow control can throttle an aggressive sender by lowering the advertised window from the receiver. Similar flow control schemes have been implemented within the message progression layer within the middleware. For LAM, the guaranteed envelope resources protocol [8] is implemented in the middleware. This protocol is a flow control mechanism within the middleware that guarantees that only a bounded number of unmatched envelopes can be successfully sent and delivered.

Since message ordering is only mandated on a per TRC basis, messages aggregated over a common link can arrive in a variety of orderings. It is the role of expected and unexpected message queues to match and store messages no matter which ordering presents itself. Where expected and unexpected message queues are implemented varies across MPI implementations. In typical implementations([54, 9, 15]), these queues are within the middleware. In [34], these queues are implemented within a customized kernel that they made. In Section 3.2, the socket per TRC design presented a design where the role of the unexpected queue was pushed down onto the transport layer socket buffer and a more local expected queue was pushed up into the MPI library.

The effectiveness of SCTP has been explored for several protocols in high latency, high-loss environments such as the Internet or a WAN. Researchers have investigated the use of SCTP in FTP [31], HTTP [40], and also over wireless [12] and satellite networks [1]. Prior to our work in [27] and [28], SCTP for MPI had not been explored.

Given the strengths of SCTP under high loss and high latency, our work with MPI over SCTP would be most relevant to projects operating over WANs that aim to link together clusters into a Cluster-of-Clusters (CoC). The approaches researchers have taken to CoCs can be classified into two paradigms: the MetaMPI approach and the unified stack approach.

The MetaMPI approach to CoCs relies on an MPI implementation to be operational for intra-cluster communication and then, for inter-cluster communi-

cation, another layer is used as glue. One MetaMPI example is MPICH-G2 [29]. MPICH-G2 is a multi-protocol implementation of MPI for the Globus [22]. Globus provides a basic infrastructure for Grid computing and addresses many of issues with respect to the allocation, access and coordination of resources in a Grid environment. The communication middleware in MPICH-G2 is able to switch between the use of vendor-specific MPI implementations and TCP, which is the protocol used within the glue layer. MPICH-G2 maintains a notion of locality and uses this to optimize for both intra-cluster and inter-cluster communications. For inter-cluster communications, the developers report that MPICH-G2 optimizes TCP but there is no description of these optimizations and how they might work in general when all the nodes are non-local. Similar to MPICH-G2, PACX-MPI [30] also takes a MetaMPI approach to CoCs utilizing vendor MPI implementations and using a glue layer between clusters.

The unified stack approach is yet another direction researchers have taken for CoCs. Defining a unified stack eliminates the redundancy and potentially high overheads that a MetaMPI glue layer presents. MPICH-VMI [38] takes this approach. MPICH-VMI utilizes the Virtual Machine Interface (VMI) [37], a middleware optimized for communication within heterogeneous networks. In addition, MPICH-VMI provides profile-guided optimizations where communication characteristics of an application can be automatically captured and used to intelligently allocate nodes. MPICH-VMI also provides optimizations for communication hierarchies by helping collectives become topologically aware. A similar project that presents a unified stack for CoCs is MPICH/MADIII [2], although it provides no profile-guided optimizations nor topologically aware collectives.

In terms of MPI applications executing over WANs, our SCTP RPI is most similar to the unified stack approach for CoCs because there is no meta-layer required to join heterogenous systems. Instead of implementing our stack for various interconnects, we rely on IP to provide us support for heterogenous networks since

most interconnects provide implementations of IP. By using SCTP atop IP, we take advantage of SCTP's loss and latency resilience keeping the MPI implementation thin and to a minimum.

One disadvantage of using SCTP for WANs might be the presence of middleboxes like firewalls or boxes performing network address translation (NAT). Even though SCTP is atop IP, some middleboxes are transport (i.e., L4 layer) aware and typically only work with TCP. However, some firewalls like the IPTables firewall within Linux, support SCTP. Even with full support in middleboxes, the fact that SCTP is multihomed presents some unique challenges. The IETF *behave* working group [4] is currently researching what SCTP NAT traversal considerations are required; an initial draft [47] has been proposed by Randall Stewart, but at the time of writing of this thesis, it is still in the process of becoming a standard solution.

# Chapter 6

# Conclusions

In this thesis, we discussed and compared various MPI designs using either TCP or SCTP. The various implementations are summarized in Table 6.1. The standard LAM-TCP implementation has a message progression layer whereas the other two designs thin or eliminate this communication component from the MPI middleware. This is done by pushing functionality commonly present in this component down onto a transport protocol. As a result, MPI implementations are simplified and in addition, they can leverage advances in networking protocol research instead of having to implement certain functionalities in the middleware. Our designs illustrate some limitations of TCP in terms of scalability and of SCTP in terms of missing features. For SCTP, because of the lack of stream-level flow control and the ability to select on a particular stream, it is not possible to completely eliminate the message progression layer with something more scalable than a socket per TRC. However, SCTP scales better than TCP, it avoids the head of line blocking that can occur in standard TCP implementations and it provides more opportunity for fairer concurrent message transfer in the case of longer messages.

MPI designs that make extensive use of standard protocol stacks may provide a solution to interoperability among interconnects and can take advantage of commodity networking as it continues to gain momentum.

| | Message Progression Layer | Message Matching and Queues | Long/Short Protocol | Scalability |
|---|---|---|---|---|
| LAM-TCP | standard, maintains full message progression state | required due to TRC link aggregation and the presence of wildcards | required to avoid resource exhaustion; has user-level flow control | requires large number of sockets using costly `select()` |
| TCP socket per TRC | eliminated, functionality split between library and transport layer | pushes unexpected queue into transport but a more local expected queue required in the library for non-blocking calls | eliminated, relies on TCP flow control | requires largest number of sockets using costly `select()` |
| SCTP | thinned, requires additional features of SCTP one-to-many API for full elimination | required, cannot receive from a specific stream in SCTP; avoids head of line blocking in TCP implementations | required, SCTP has flow control per association and not per stream | requires only a single one-to-many socket avoiding costly `select()` |

Table 6.1: Summary of presented MPI implementations

# Bibliography

[1] Rumana Alamgir, Mohammed Atiquzzaman, and William Ivancic. 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] Olivier Aumage and Guillaume Mercier. MPICH/MADIII: a cluster of clusters enabled MPI implementation. *IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03)*, 2003.

[3] B. Barrett, J. M. Squyres, A. Lumsdaine, R. L. Graham, and G. Bosilca. Analysis of the component architecture overhead in Open MPI. In *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.

[4] IETF behave working group. Working group charter. Available from http://www.ietf.org/html.charters/behave-charter.html.

[5] BProc. Beowulf distributed process space. Available from http://bproc.sourceforge.net.

[6] Ron Brightwell and Keith Underwood. Evaluation of an eager protocol optimization for MPI. In *PVM/MPI*, pages 327–334, 2003.

[7] Ron Brightwell, Keith Underwood, and Rolf Riesen. An initial analysis of the impact of overlap and independent progress for MPI. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, September 2004.

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

[9] Greg Burns, Raja Daoud, and James Vaigl. LAM: An open cluster environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[10] Ralph Butler, William Gropp, and Ewing Lusk. Components and interfaces of a process management system for parallel programs. *Parallel Computing*, 27(11):1417–1429, 2001.

[11] Neal Cardwell, Stefan Savage, and Thomas Anderson. Modeling TCP latency. In *INFOCOM*, pages 1742–1751, 2000.

[12] Yoonsuk Choi, Kyungshik Lim, Hyun-Kook Kahng, and I. Chong. An experimental performance evaluation of the stream control transmission protocol for transaction processing in wireless networks. In *ICOIN*, pages 595–603, 2003.

[13] Aaron E. Darling, Lucas Carey, and Wu chun Feng. The design, implementation, and evaluation of mpiBLAST. In *ClusterWorld Conference & Expo and the 4th International Conference on Linux Clusters*, 2003.

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

[15] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 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.

[16] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 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.

[17] Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. In *Proceedings of the Linux Symposium*, July 2004.

[18] Patricia Gilfeather and Arthur B. Maccabe. Connection-less TCP. In *IPDPS*, 2005.

[19] Globus. Grid middleware. Available from http://www.globus.org.

[20] Thomas J. Hacker, Brian D. Noble, and Brian D. Athey. Improving throughput and maintaining fairness using parallel TCP. In *IEEE INFOCOM*, 2004.

[21] Wei Huang, Gopalakrishnan Santhanaraman, Hyun-Wook Jin, and Dhabaleswar K. Panda. Design alternatives and performance trade-offs for implementing MPI-2 over Infiniband. In *PVM/MPI*, pages 191–199, 2005.

[22] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl Journal of Supercomputer Applications*, 11(2):115 – 128, 1997.

[23] Janardhan R. Iyengar, Paul D. Amer, and Randall Stewart. Retransmission policies for concurrent multipath transfer using SCTP multihoming. In *ICON 2004, Singapore*, November 2004.

[24] Janardhan R. Iyengar, Keyur C. Shah, Paul D. Amer, and Randall Stewart. Concurrent multipath transfer using SCTP multihoming. In *SPECTS 2004, San Jose*, July 2004.

[25] Humaira Kamal, Brad Penoff, Mike Tsai, Edith Vong, and Alan Wagner. Using SCTP to hide latency in MPI programs. In *Heterogeneous Computing Workshop: Proceedings of the 2006 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006.

[26] Humaira Kamal, Brad Penoff, and Alan Wagner. Evaluating transport level protocols for MPI in the Internet. In *International Conference on Communications in Computing (CIC 2005)*, June 2005.

[27] Humaira Kamal, Brad Penoff, and Alan Wagner. SCTP-based middleware for MPI in wide-area networks. In *3rd Annual Conf. on Communication Networks and Services Research (CNSR2005)*, pages 157–162, Halifax, May 2005. IEEE Computer Society.

[28] Humaira Kamal, Brad Penoff, and Alan Wagner. SCTP versus TCP for MPI. In *Supercomputing '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2005. IEEE Computer Society.

[29] Nicholas T. Karonis, Brian R. Toonen, and Ian T. Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *CoRR*, cs.DC/0206040, 2002.

[30] Rainer Keller, Edgar Gabriel, Bettina Krammer, Matthias S. Mueller, and Michael M. Resch. Towards efficient execution of MPI applications on the

Grid: Porting and optimization issues. *Journal of Grid Computing*, 1:133–149, June 2003.

[31] Sourabh Ladha and Paul Amer. Improving multiple file transfers using SCTP multistreaming. In *Proceedings IPCCC*, April 2004.

[32] Supratik Majumder and Scott Rixner. Comparing Ethernet and Myrinet for MPI communication. In *LCR '04: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, pages 1–7, New York, NY, USA, 2004. ACM Press.

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

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

[35] Alberto Medina, Mark Allman, and Sally Floyd. Measuring the evolution of transport protocols in the Internet, April 2005. To appear in ACM CCR.

[36] OpenPBS. Open-source portable batch system implementation. Available from http://www.openpbs.org.

[37] Scott Pakin and Avneesh Pant. VMI 2.0: A dynamically reconfigurable messaging layer for availability, usability, and management. In *The 8th International Symposium on High Performance Computer Architecture (HPCA-8), Workshop on Novel Uses of System Area Networks (SAN-1)*, Cambridge, Massachusetts, February 2, 2002.

[38] Avneesh Pant and Hassan Jafri. Communicating efficiently on cluster based grids with MPICH-VMI. In *Proceedings of 2004 IEEE International Conference on Cluster Computing*, San Diego, CA, September 2004. IEEE.

[39] Brad Penoff and Alan Wagner. Towards MPI progression layer elimination with TCP and SCTP. In *11th International Workshop on High-Level Programming Models and Supportive Environments (HIPS 2006)*. IEEE Computer Society, April 25 2006.

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

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

[42] Sourceforge. Linux channel bonding project website. http://sourceforge.net/projects/bonding.

[43] Jeff Squyres. Lead developer of OpenMPI, private communication, 2005.

[44] Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. Request progression interface (RPI) system services interface (SSI) modules for LAM/MPI. Technical Report TR579, Indiana University, Computer Science Department, 2003.

[45] Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. The system services interface (SSI) to LAM/MPI. Technical Report TR575, Indiana University, Computer Science Department, 2003.

[46] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *UNIX Network Programming, Vol. 1, Third Edition*. Pearson Education, 2003.

[47] Randall Steward and Michael Tuexen. SCTP network address translation Internet draft. Available from http://www.ietf.org/internet-drafts/draft-stewart-behave-sctpnat-01.txt.

[48] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, M. Kalla I. Rytina, L. Zhang, and V. Paxson. The Stream Control Transmission Protocol (SCTP). Available from http://www.ietf.org/rfc/rfc2960.txt, October 2000.

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

[50] Top500. Top 500 Supercomputing Sites. Available from http://www.top500.org.

[51] Dave Turner, Shoba Selvarajan, Xuehua Chen, and Weiyi Chen. The MP_Lite message-passing library. In *14th IASTED International Conference on Parallel and Distributed Computing and Systems*, Cambridge, Massachusetts, November 2002.

[52] W. Gropp and E. Lusk. MPICH working note: Creating a new MPICH device using the channel interface. Technical Report ANL/MCS-TM-213, Argonne National Laboratory, July 1996.

[53] W. Gropp and E. Lusk. MPICH working note: The implementation of the second generation MPICH ADI. Technical Report ANL/MCS-TM-number, Argonne National Laboratory, 2005.

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

[55] Interoperable Message Passing Interface website. IMPI. http://impi.nist.gov/.

[56] T.S. Woodall, R.L. Graham, R.H. Castain, D.J. Daniel, M.W. Sukalski, G.E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine. 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.

[57] J. Yoakum and L. Ong. An introduction to the Stream Control Transmission Protocol (SCTP). Available from http://www.ietf.org/rfc/rfc3286.txt, May 2002.