Transport Level Features for Commodity Clusters

by

Bradley Thomas Penoff

B.Sc., The Ohio State University, 2001
M.Sc., The University of British Columbia, 2006

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

(Vancouver)

June, 2011

Abstract

There is a need for systems to provide additional processing to extract useful information from the growing amounts of data. High Performance Computing (HPC) techniques use large clusters comprised of commodity hardware and software to provide the necessary computation when a single machine does not suffice. In addition to the increase in data, there have been other architectural changes like the advent of multicore and the presence of multiple networks on a single compute node, yet the commodity transport protocols in use have not adapted. It is therefore an opportune time to revisit the question of which transport features are necessary in order to best support today's applications. Popular in HPC, we use the Message Passing Interface (MPI) to provide support for large scale parallel applications. We propose features to the transport protocol to overcome the problems with reliability, performance, and design simplicity existing in Ethernet-based commodity clusters. We use the Stream Control Transmission Protocol (SCTP) as a vehicle to implement tools having the proposed transport features for MPI. We develop several SCTP-based MPI implementations, a full-featured userspace SCTP stack, as well as enable the execution of unmodified MPI programs over a simulated network and SCTP implementation. The tools themselves provide the HPC and networking communities means to utilize improved transport features for MPI by way of SCTP. The tools developed in this thesis are used to show that the proposed transport features enable further capabilities regarding the performance, reliability, and design simplicity of MPI applications running on Ethernet-based cluster systems constructed out of commodity components.

Preface

[81] Brad Penoff, Mike Tsai, Janardhan Iyengar, and Alan Wagner. Using CMT in SCTP-based MPI to exploit multiple interfaces in cluster nodes. In *Proceedings, 14th European PVM/MPI Users' Group Meeting*, Paris, France, September 2007.

[106] Mike Tsai, Brad Penoff, and Alan Wagner. A hybrid MPI design using SCTP and iWARP. In *Communication Architecture for Clusters: Proceedings of the 2008 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2008.

[84] Brad Penoff, Alan Wagner, Irene Rüngeler, and Michael Tüxen. MPI-NeTSim: A network simulation module for MPI. In *The Fifteenth International Conference on Parallel and Distributed Systems (ICPADS'09)*, 2009.

[80] Brad Penoff, Humaira Kamal, Alan Wagner, Mike Tsai, Karol Mroz, and Janardhan Iyengar. Employing transport layer multi-railing in cluster networks. *J. Parallel Distrib. Comput.*, 70(3):259–269, 2010.

[94] Irene Rüngeler, Brad Penoff, Michael Tüxen, and Alan Wagner. A New Fast Algorithm for Connecting the INET Simulation Framework to Applications in Real-time. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques (SIMUTools2011)*, 2011.

[83] Brad Penoff and Alan Wagner. High performance computing using commodity hardware and software. In Victor Leung, Eduardo Parente Ribeiro, Alan Wagner, and

Janardhan Iyengar, editors, *Multihomed Communications for Heterogeneous Networks: Concepts and Applications of SCTP Multihoming*. Wiley, 2011.

This thesis consists of research I have contributed to during my time at UBC. All projects have been under the supervision of Dr. Alan Wagner; we are co-authors on all work described here. Depending on the project, I have also worked together with a variety of local and non-local collaborators. In this preface, I provide a brief account of the relevant publications used in this thesis.

The initial design of SCTP-based MPI-1 middleware described in Section 3.1 was done together with Humaira Kamal in two papers [53, 54] for LAM/MPI, but it targeted executing in a wide-area environment and was never released to the public; her Masters thesis was an expanded version of this work. My Masters thesis extended this design further to push more functionality of MPI middleware into various transport protocols, expanding upon Penoff and Wagner [82].

The initial LAM/MPI design was extended by Mike Tsai and myself to add MPI-2 functionality to target Ethernet-based clusters. SCTP was added into MPICH2 as `ch3:sctp` during my internship in the summer of 2006 at Argonne National Laboratory. The first official release which included SCTP support in MPICH2 was version 1.0.5. This served as a foundation for several variations of SCTP-based MPI developed for this thesis, as is shown in Table 3.1.

Parts of the thesis were previously published in Penoff's JPDC work [80]. This was work published together with Humaira Kamal, Mike Tsai, Karol Mroz, and Janardhan Iyengar that expanded the original Euro PVM/MPI conference paper [81]. The parts of this work used in this thesis includes (1) the Open MPI design in Section 3.3, (2) the SCTP failover introduction and reliability testing in Section 6.1, (3) multirail performance testing done in Section 7.1, as well as (4) the multirail configuration details in Section 8.2. Portions of this work also are in preparation for a chapter prepared with Alan Wagner in [83].

The work conducted with Michael Tüxen and Irene Rüngeler for MPI-NeTSim has been published in two papers [84, 94]. The original interface for MPI to the simulator and our time dilation technique outlined in Chapter 5 appeared in ICPADS 2009 [84]. The faster, adaptable algorithm in Section 5.2.2 was published in SIMUTools 2011 [94].

Together with Mike Tsai, we published some initial results for SCTP-based RDMA-enabled network cards in IPDPS 2008 [106]. Portions of this work are used in this thesis in Section 8.1. An expanded version formed his Masters thesis.

Finally, some parts of this thesis were done in collaboration with others but have yet to be published. Using MPI-NeTSim, the failover testing for MPI applications in Section 6.1.2 as well as the I-Bit testing in 7.3.2 was done with Irene Rüngeler. For the userspace SCTP stack, the initial design in Section 4.1 was done together with Humaira Kamal and Michael Tüxen. The performance optimizations of the userspace stack discussed in Section 7.2 was work done together with Michael Tüxen and Irene Rüngeler.

Otherwise, all writing and any other work in this thesis was done by myself, with the helpful advice of my peers.

Table of Contents

List of Tables

List of Figures

Acknowledgments

This research has been supported by the UBC NSS Lab through funding from NSERC and Cisco as well as through a UBC University Graduate Fellowship. I would like thank Alan Wagner and the NSS Group at UBC for their support and feedback. I would also like the thank my family for their endless patience and help. And last but not least, I would like to thank God.

Chapter 1

Introduction

The futurist John Naisbitt has said, "We are drowning in information but starved for knowledge" [68]. In an era where we now create more data than we are able to store [34], there is a pressing need in all areas to make the most of our compute resources to turn data into information. Sequential computing alone cannot meet the challenge of extracting the information from the large datasets that are at the core of most enterprises and central to many scientific endeavors such as bio-informatics. The target architecture for programs operating in this domain are parallel and distributed, taking advantage of multiple cores and multiple machines to speed-up the computation. High-performance computing can no longer be the purview of specialists on specialized hardware and systems. Massive data is everywhere and is it important to take advantage of advancements in high performance computing to develop tools for commodity-type hardware and software to allow the more everyday use of parallelism.

Parallel processing uses multi-core, multi-socket boards on a single machine that communicate to other nearby machines in the same compute cluster. To communicate within a cluster, there has been a move away from specialized interconnects and towards commodity networking such as Ethernet or Infiniband in order to support the communication needs of cluster and Internet computing. In high performance computing, the most common way to write distributed parallel applications that scale is to use message passing by way of the Message Passing Interface (MPI) [9].The focus of this thesis is to use Ethernet and MPI technologies to provide an inexpensive yet robust system for the execution of compute and

data intensive parallel applications in a cluster environment.

Prior to the advent of MPI in 1994, different supercomputers and computing clusters each had their own unique libraries [62]. This was quite troublesome as the lifespan of the applications was longer than that of the hardware on which they ran, meaning that application writers needed to port their source code from system to system if they wanted to run their code elsewhere. The MPI standard was created as a portable solution for distributed parallel applications. MPI is a result of a series of forums consisting of researchers from industry, academia, and government. MPI has been very successful and continues to be actively developed, as all major cluster manufacturers deliver a version of MPI [40]. The distributed parallel applications targeted in this thesis are those written using MPI.

Ethernet remains the dominant networking technology used in commodity clusters. Since the inception of Ethernet in the mid 1970s at Xerox PARC and the initial IEEE 802.3 standard of the early 1980s, Ethernet has continued to evolve. Through the years, the line speeds have increased from 3 Mbps to 10 Gbps, with forthcoming plans for 40 GigE and even 100 GigE [41]. Generally for Ethernet, a consortium of several companies has helped to standardize these changes. The standard makes it possible to implement at a low cost, and as a result, Ethernet is ever-present in the majority of systems today, and is relatively performant for the cost. This thesis focuses on Ethernet-based clusters to provide robust systems for MPI applications.

The strength of the MPI standard is its portability as several different implementations exist on a variety of systems and clusters. From one implementation to the next, several different interconnects are supported, but most popular open-source implementations have the ability to run over Ethernet at Layer 2 using the TCP and IP protocols at Layers 3 and 4; this because TCP is a readily available, reliable transport protocol in all major operating systems, and MPI requires reliability.[1] Other Ethernet-based solutions that do

[1]The MPI 3 standard [63], scheduled to be released in late 2011, will relax this requirement giving the application more flexibility to specify its own required level of fault tolerance.

not use TCP at Layer 4 try to remain thin and efficient by implementing reliability on top of Ethernet directly [21, 36] or on top of UDP/IP [3], UDP only providing demultiplexing at Layer 4.

As Ethernet and commodity clusters continue to evolve, it is imperative to rethink the intermediate transport protocol features necessary to implement MPI [55]. The forthcoming Converged Enhanced Ethernet (CEE) [22] extension adds RDMA to Ethernet at Layer 2, but it is unclear how this affects the transport protocols. It is also unclear how the 40 GigE and 100 GigE standards exactly affect the features necessary for the Layer 4 transport protocols, as transport protocols are still necessary to communicate outside of the cluster. Clusters continue to increase in scale and the compute nodes are more commonly equipped with multiple cores and multiple network cards. The changes happening within Ethernet and clusters make it an opportune time to revisit the question of what features are necessary in the transport layer in order to best support MPI in a robust, performant system for compute and data intensive processing in a cost-effective, Ethernet-based commodity cluster.

1.1 Existing Parallel and Distributed Systems

There is a vast ecosystem of systems that execute parallel and distributed computing applications across several computers to enable users to run large problem sets larger than the memory of one machine. These systems vary widely, yet each can be characterized by the following properties: (1) the applications they target, (2) how they are managed, and (3) the make-up of their software and hardware components.

A recent trend of systems that target throughput applications consisting of multiple independent sequential tasks has been to use large clusters consisting of several PCs built using commodity off-the-shelf (COTS) software and hardware components, also known as Beowulf clusters [79]. COTS-based clusters are commonly constructed to use Ethernet as an

interconnect because it yields the best performance/cost ratio; this has been the approach of locally managed, Beowulf-like clusters such as those used by Google's MapReduce [27] and other cloud computing approaches managed remotely like Amazon's EC2 [1].

This thesis targets parallel applications which distributively compute dependent tasks that cooperate to achieve a larger common goal. There is a spectrum of various parallel clusters. On the one extreme, multicore chips are now providing "clusters in a box"; this and other green computing initiatives have expanded the focus of clusters to include low power as an aim because it can be more power-efficient to execute an application on multiple cores in parallel instead of at a higher clock speed [78]. For large datasets, multiple machines become necessary because they exceed the capacity of one machine. On the other extreme, large and completely specialized supercomputers offer customized CPUs, interconnects, memory layouts, and internal protocols. In the middle of these two extremes lies the commodity parallel clusters that are the focus of this thesis.

At the upper end of the Top 500 list [105], typically there are parallel clusters made of commodity CPUs. These can be connected using specialized interconnects and software, as is the case with IBM BlueGene [30], or by using low-latency commodity interconnects like Infiniband that cost more than Ethernet. Lower down the list, Beowulf-like clusters utilize commodity Ethernet typically with standard Linux as the operating system, resulting in an attractive performance/cost ratio. Specifically, this Ethernet-based type of Beowulf-like cluster is the focus of this thesis.

1.2 Problems with Existing Ethernet-based Parallel Systems

Parallel applications require reliability from the systems they run on so when running over a commodity Ethernet-based clusters, they typically make use of TCP which is a reliable IP-based transport protocol unless they implement their own reliability over UDP. Regardless,

the use of UDP or TCP does not come without consequence; there are three main problems with existing Ethernet-based COTS parallel clusters using UDP or TCP:

- Lack of **performance** – Transport protocols running over Ethernet are thought to be too slow for communication intensive applications. Even for applications with less communication, transport protocols can still fail to fully utilize the network's potential and thereby the machines. Such applications are network-limited.

- Lack of **reliability** – Large scale parallel applications can take days to compute on their massive data sets. The longer a system runs and the larger the system becomes, the higher the probability of failure. Failures can occur on a particular message, on the network, or at the nodes. The TCP transport protocol, the most commonly used reliable transport protocol over IP, shows little resilience to network loss of any scale from packet to link loss. Additionally, TCP has even been shown to validate invalid data within the stack itself [103], given its insufficient checksum for modern scales; missing such errors can cause memory in the system to be overrun, causing failure and showing the need for increased message reliability. There is also no protection if an entire link goes down.

- Lack of **design simplicity** – MPI implementations are simplified by utilizing available transport protocol features. In this sense, a system lacks design simplicity when one of its features can be implemented down in the transport protocol in a complete and general manner. MPI implementations are simplified if they can avoid implementing a feature and instead can enable it by instructing the transport protocol.

Despite the previous problems of Ethernet-based COTS systems, Ethernet and transport protocols are still used because their high volume make them ubiquitous and therefore extremely cost effective, providing decent performance inexpensively. As well, Ethernet and IP-based transport are well understood, therefore users and administrators can benefit from

the wealth of knowledge and tools that others have created. Using a standard transport allows for a seamless integration between a cluster's LAN to the WAN. Overall, the use of Ethernet and IP-based transport protocols allows applications to span the continuum from reliable "in the closet" networks to more open geographically diverse, less reliable networks.

1.3 Desired Transport Features for MPI

Despite the promising outlook for the future of Ethernet and IP-based transport protocols, the lack of performance, reliability, and design simplicity still need to be addressed. There are several transport protocol features which, if present, could help increase the levels of performance, reliability, and design simplicity in present MPI-based systems running on commodity clusters. These features are:

- Message Framing – MPI communications involve passing fully formed messages. For a stream-based transport protocol, the message edges need to be delimited by the MPI middleware. It would be ideal if the message framing was generically implemented in the transport protocol.

- Message Reliability – MPI messaging requires full reliability, so the arrival of a sent message needs to be guaranteed.

- Path Redundancy – It is common for an endpoint to have several network interfaces. If a connection between endpoints was aware of the redundant paths, then path failover could be used effectively if the main path were to suffer a complete failure. This additional reliability would be helpful for MPI because aborting the full computation could be avoided.

- Multirailing – To utilize the full bandwidth of a system, multiple network interfaces could be used simultaneously to send MPI messages.

- Strong Checksum – Large scale MPI applications send large numbers of messages. With the increased bandwidth of current network fabrics and the increased scale of applications, additional data integrity is required [103].

The desired transport protocol features for MPI could be added to any IP-based transport. The most common transport protocol, TCP, offers reliability but does not have message framing, and support for a stronger CRC32c checksum has not yet been fully specified [13]. The standardization for multiple networks in TCP began after the work for this thesis had already began, and no standard implementation has appeared yet [45]. UDP provides message framing but none of the other proposed features. It would be non-trivial to add these features to either UDP or TCP.

Most of the proposed transport features for MPI are present in the base Stream Transmission Control Protocol (SCTP) [100, 101, 119], and those that are not can be more easily added to SCTP since it was designed with extensibility in mind. The relative newness of SCTP means we are more likely to influence lasting changes to the protocol. Solutions found with SCTP will be general so that they could be applied to other transport protocols like TCP. This thesis uses SCTP as the vehicle to demonstrate the proposed transport protocol features for MPI to address some of the problems in existing commodity clusters.

After over 20 years of TCP and UDP being the only two options over IP, SCTP became the third transport protocol option in October 2000 [100, 101, 119]. SCTP also has familiar features that TCP has like flow control and congestion control. By default, SCTP is reliable and message-based, like MPI. There is also support for increased reliability within the transport protocol itself through a strong CRC32c checksum and full path-level redundancy. The Concurrent Multipath Transfer (CMT) extension to SCTP provides multirailing across a multihomed SCTP connection (termed an "association") by striping data across all paths at the same time [48]. SCTP implementations exist for most major operating systems and applications can make use of them using Berkeley sockets, the SCTP socket API [99], or

the Java SDK [20]. In this thesis, SCTP is used by MPI in order to create the most robust system.

In addition to the proposed transport features for MPI, SCTP provides some additional features which we have also studied for MPI. These features include:

- Messages In-Order per Message Stream (called "multistreaming") – Independent messages can be sent over different streams in SCTP. Message ordering is only guaranteed on messages with the same stream. MPI messages specify a message type called a tag. Between two endpoints, ordering of messages with the same tag must be maintained. When wildcards are not used, messages using different tags have no ordering dependencies, so in SCTP, streams could be mapped to tags.

- User Adjustability – MPI implementations can be simplified if features can be enabled by instructing the transport protocol rather than having to implement the feature itself.

- Selective Acknowledgment (SACK) – Use of cheap commodity switches that could drop packets more frequently may make out-of-order arrival more common. A multihomed setting will also result in out-of-order arrival as well. Using SACK, gaps of missing data can be expressed to the sender unlike cumulative acknowledgment schemes which can only convey the last received data packet.

Some of these additional features exist in other transports but not all of them. For example, SACK is provided as an extension to TCP, however the number of gaps expressed in SACK is limited to what can fit in the TCP Options field [60]. TCP does not provide multistreaming, and making such an addition would be non-trivial. Message ordering per stream is present over UDP in the Structured Stream Transport (SST) [31], but SST lacks other desired features like multihoming support. Overall, SCTP provides more features than other transport protocols and applications have more control since they can fully

adjust these parameters. For example, applications can adjust the features such that it acts like TCP, or in another way so that it acts like UDP; SCTP also offers new solutions so applications can craft additional transport feature combinations, giving entirely new possibilities. All of these features of SCTP can be beneficial when implementing MPI for an Ethernet-based system.

In this thesis, open-source tools are developed that enable the evaluation of both the advantages and potential disadvantages of these desired transport features for MPI. The tools are also available for others to utilize. The belief in this thesis is that these transport features can overall be beneficial when designing a robust cost-effective Ethernet-based COTS-based parallel cluster targeting MPI applications.

1.4 Thesis Statement

The thesis statement offers a solution to the problems introduced in Section 1.2 that are present in existing Ethernet-based low-end COTS clusters:

The addition of the appropriate transport features enables further capabilities regarding the performance, reliability, and design simplicity of MPI implementations running on Ethernet-based cluster systems constructed out of commodity components.

In this thesis, the goal is to use SCTP as the research vehicle to investigate these transport features, maximizing their advantages with respect to performance and reliability while minimizing the disadvantages. We give examples of the design simplicity of SCTP. Support for this thesis statement is demonstrated through a research program consisting of first creating a series of system implementations, and then using these as tools in order to do experimentation on a combination of real systems as well as emulation. The following tools were developed as part of this thesis:

- SCTP-based MPI Implementation – Since MPI is the de facto standard for writ-

ing distributed parallel applications, SCTP-based MPI middleware was designed and developed in order to provide a necessary infrastructure for the investigation and evaluation for this thesis.

- Userspace SCTP Implementation – The best SCTP implementation is in the FreeBSD kernel however Linux is the most popular commodity operating system used by parallel clusters. The FreeBSD SCTP implementation was extracted from the kernel to userspace so that is can run in Linux and all other operating systems. This userspace SCTP implementation enables easier investigations for SCTP protocol onload as well as eases the integration of the MPI middleware and the SCTP protocol itself.

- Simulated SCTP Networks with Emulated MPI – SCTP has lots of parameters and new features. In order to investigate their impact on MPI programs, we extended the OMNeT++ network simulator [115] to be capable of running real MPI programs using our unique time dilation algorithm. This tool enables the study of SCTP parameter impact on a variety of network topology and link characteristics, which could be rapidly changed more simply with simulated networks than with real networks.

All of these developed tools have been contributed to the active open-source projects which they were created from so the tools themselves can therefore be seen as a major contribution of this thesis to the community. In this thesis, these developed tools will be used to show how to use SCTP's features in MPI middleware in order to enable additional capabilities regarding the performance, reliability, and design simplicity. The different features and mechanisms that will be explored as part of this thesis are summarized as follows:

- Using real networks with the kernel FreeBSD implementation[2], the additional reliabil-

[2] This study was conducted prior to the construction of a userspace implementation of the kernel FreeBSD SCTP stack and its features for use on all commodity systems. See Chapter 4.

ity and performance capabilities provided by SCTP's multihoming feature is demonstrated using MPI bandwidth benchmarks, comparing the data striping approaches done by SCTP in the transport to data striping done within the MPI middleware.

- Using simulated networks and transport protocol, the addition reliability and performance capabilities provided by SCTP's multihoming and I-Bit feature [95] are demonstrated using a variety of real MPI applications.

- The SCTP userspace implementation provides additional performance, reliability, and design simplicity since it enabled the features present in the FreeBSD kernel stack onto multiple platforms, particularly Linux which is the commodity operating system most commonly used by parallel clusters.

- The SCTP userspace implementation enables the creation of a tighter coupling between the MPI application, middleware, the SCTP protocol, and the NIC.

- SCTP provides additional features, extensibility, and user flexibility; the combination means that it can adapt to new environments more quickly. Its design simplicity is exemplified by showing how enabling RDMA over IP required changes to the core TCP protocol, but for SCTP it only required existing features to be tuned a particular way.

- SCTP limitations were studied using the userspace stack. The impact of SACK and CRC32c were of little or no impact, however protocol overheads had an impact on short message bandwidth.

- SCTP multistreaming avoids head-of-line blocking amongst MPI messages.

Investigating these key areas will show that with the appropriate extensions, SCTP can enhance the performance, reliability, and design simplicity of MPI implementations running

on Ethernet-based cluster systems constructed out of commodity hardware, as the thesis statement proclaimed.

The work described in this thesis has resulted in six peer reviewed publications, which are listed in the Preface.

1.5 Thesis Structure

Following this introduction, Part I focuses on the tools developed which are necessary to complete the research. After telling of the tool development experiences in Chapter 2, a chapter is then devoted to each of the tools; Chapter 3 shows our SCTP-based MPI middleware, Chapter 4 shows our userspace SCTP implementation, and Chapter 5 describes how we enabled emulated MPI programs to utilize the simulated SCTP stack of OMNeT++. After that, in Part II, each of the next three chapters of the thesis are devoted to how SCTP provides additional capabilities for parallel applications using Ethernet-based commodity clusters. Chapter 6 is on reliability, Chapter 7 is on performance, and Chapter 8 is on design simplicity. A synopsis ends this document in Chapter 9.

Part I

Tools

Chapter 2

Tool Development Experiences

As this thesis works with clusters built from commodity components, there was a desire to develop open-source software for our tools. Commodity components are more widely used compared to specialized components, so there is a greater opportunity to make a wider impact and more lasting effect when, for example, choosing to work with commodity Ethernet rather than a specialized interconnect. The same is true for commodity software as well. Inherent in the culture of building Beowulf-like commodity clusters is a great value in having COTS components as building blocks so that users can mix technologies together to match their preferences. Providing unique software components and making our tools open-source, distributing them freely as a part of an existing popular production-level tool, helps make their potential impact wider.

The development of the tools in this thesis came with a variety of non-technical challenges. In order to integrate one's code into the main branch of a popular open-source project, trust needs to be established. This is more complicated when the team is geographically disconnected, as was the case for all of the tools developed here, with collaborators being at times in Germany, Pakistan, or the United States of America. Each project had their own unique relationships formed by internships, email correspondence, wikis, video conference, etc. Teamwork for these projects required patience and dedication.

There was no shortage of technical challenges for the non-trivial tools developed in this thesis as well. While the specific design details are described in the chapters to come, there are some technical challenges which spanned more than one of the open-source tools

developed in this thesis. Some of these technical challenges involved project management. Integration into repositories required compliance with project-specific test suites consisting of hundreds of test cases, built and executed on a variety of operating systems. A hierarchy of repositories was necessary to setup and maintain in order to do the appropriate local and non-local gatekeeping, for when the source code changes underneath you due to some external commit. Continued chartering of the technical plan of attack during the development of these tools was necessary because all of the tools in this thesis were based off of code bases containing hundreds of thousands of lines of code developed over decades; while the end-goal was clear, the intermediate milestones often had to be set stepwise in order to give ourselves time to wrap our minds around the best approach to achieve the end-goal. Again, none of these project management related technical challenges were trivial.

Many of the technical challenges that happened during development of the tools in this thesis were related to the difficulties of debugging large-scale, distributed applications. Timing plays a role in the surfacing of bugs, so fixing these would take continued patience to recreate a bug with the appropriate message logging in the application and MPI middleware, kernel memory dumps, and network traces to investigate; even then, patience is needed to continue because the profiling could amass piles of data to sift through to find the root of the problem. Given the infancy of SCTP, the problems were sometimes in the stack itself, so this situation would involve coordinating with the developers of that particular kernel and working together to find the appropriate patch. As one can see, debugging was challenging when developing tools for large-scale, distributed applications.

Several tools were developed to provide a framework for demonstrating this thesis. These tools enabled the use of SCTP by parallel MPI applications in real cluster environments as well as simulated networks. In Part I, the developed tools are described in detail. Later, Part II focuses on the *use* of these tools with respect to proving the thesis itself, but here the *design* of the tools themselves are the focus. Our initial SCTP-based MPI implementations are first introduced in Chapter 3 to study the use of SCTP features for

parallel applications. In Chapter 4, our own userspace SCTP implementation is described. The userspace SCTP implementation enables easier investigations for SCTP protocol onload as well as the tight integration of the MPI middleware and the SCTP protocol itself. Finally, in Chapter 5 our extension of the OMNeT++ network simulator is introduced; it enables us to rapidly change the underlying link topologies and characteristics, allowing complete studies of the impact of SCTP transport features when running real MPI programs over simulated networks. Altogether, these tools form the infrastructure necessary for this thesis.

Chapter 3

SCTP-based MPI Implementations

In order to execute any MPI application over top of SCTP, it was necessary to develop a feature-complete implementation of SCTP-based MPI middleware. SCTP offers features that TCP does not have like multistreaming, multihoming, the ability to be message-based, as well as supporting the option to use UDP-like one-to-many sockets for multiple associations; incorporating these features into a full MPI-2 implementation of MPI middleware enables the use and evaluation of their effectiveness for all MPI applications.

The starting point of my SCTP-based MPI implementations created in this thesis was our Supercomputing (SC) paper [54]. The original implementation [53, 54] was based on LAM/MPI [15], an earlier and now discontinued version of MPI. This initial design supported the MPI-1 standard and not MPI-2. The changes outlined as part of this thesis build on and extend the basic design, resulting in a suitable platform for later experiments.

One goal was to have a fully compliant MPI-2 implementation, which was achieved. This not only was useful for our own research but it allowed us to contribute our MPICH2 implementation to the Argonne National Laboratory distribution of MPICH2 [2]. Since version 1.0.5, our SCTP code has been included in all MPICH2 releases. SCTP-based code has also been committed to the main Open MPI [35] repository, enabling further tests and comparisons particularly with their unique middleware-based approach to multirailing. Together, MPICH2 and Open MPI are the most popular open source MPI implementations so supporting code in their respective code bases has allowed others to make use of SCTP for MPI.

In this chapter, we first highlight the common concepts of SCTP-based MPI implementations in Section 3.1. After this, we present how these common concepts are specifically used to extend the MPI-1 portions of two major open source MPI designs: MPICH2 and Open MPI. In Section 3.2, our MPICH2 implementation is presented which is used in this thesis as our best implementation of SCTP-based middleware, and our only implementation supporting MPI-2 features. Next in Section 3.3, our Open MPI implementation is presented which is used in this thesis to study the comparison of multihoming in the SCTP transport to data striping done in the MPI middleware. A summary is provided in Section 3.4 of all of the SCTP-based MPI implementations.

3.1 Common Design Features

TCP is present in all open source implementations of MPI. Since we are using SCTP features to enable new capabilities for existing implementations of MPI middleware on Ethernet-based clusters, TCP's code is used as a basis for comparison to describe the common design features present across our SCTP-based MPI implementations.

3.1.1 Message-based

A TCP connection is a single stream of bytes passed from one application to another. If a TCP sender were to call `send()` three consecutive times with 30 bytes each time, the `recv()` call on the other side has no guarantee for how many of the bytes will be read at once. For example, `recv()` could return 12 bytes the first time and 78 bytes the second. These bytes are received in the same order they are sent but not necessarily in the same 30-byte bundles, so in TCP-based MPI middleware, it must do some presentation layer processing to parse the message boundaries in the data stream, by either counting bytes or use of an in-stream delimiter. For SCTP, message boundaries are preserved so no such parsing is necessary. As a result, less message framing had to occur inside the MPI middleware.

3.1.2 Multistreaming

A TCP connection between two hosts ensures that messages sent from the one side arrive at the receiver in the exact same order. If packets arrive at the receiver out-of-order, they cannot be delivered to the application; even if out-of-order packets are logically independent, the global ordering of TCP requires that all packets be given to the application in order. For example, if the text of a webpage and then its embedded images are sent to a client thereafter, if the packets of the second image arrive before the packets of the first image, then the packets of the second image cannot be delivered to the application until the data from the first image arrives, despite the data being logically separate.

The multistreaming capability of SCTP weakens the delivery requirements as compared to TCP. Multistreaming enables the sender to express the underlying data dependencies by allowing them to designate a logical stream for a particular message. Data with the same stream will be delivered to the application in order, but data on different streams have no ordering constraints. So for example, if a webpage and its contained images were sent to a client, different images can be sent on different streams. On the client, the delivery of data from images two, three, and four will not be delayed if the first image's packets are yet to arrive.

According to the MPI Standard [62], MPI messages must be received in the order they are sent, for any combination of message tag, rank, and context. A context is a set of processes, a rank is one process in particular amongst that set, and a tag is a particular message type. In TCP, a connection is established between two ranks and messages are sent as they arrive in the MPI code. Given the fact that TCP delivers bytes in a strict FIFO ordering, this indicates that MPI messages are delivered by the application in the same order they are sent to the same rank [54]. This is too strict for MPI. Suppose message A and message B are sent in that order on different tags; if message A is lost in transmission but message B arrives to the receiver, message B could be delivered to the application

according to MPI semantics, but TCP semantics prevents this. This is called the head-of-line blocking problem in TCP [101].

Multistreaming in SCTP can be used to match the requirements of message semantics of the MPI Standard to the MPI implementation. Since there is no dependence on the order the messages are delivered compared to when they were sent from one SCTP stream to the next, we therefore assign a different stream to any tag/context pair of an MPI message in SCTP-based middleware [54]. Assuming the scenario described in the previous paragraph, message B can be delivered to the application before message A arrives, preventing head-of-line blocking.

In work by Buntinas et al. [14], the Nemesis communication system has optimized MPI communications between processes on the same machine by recognizing communication pairs occurring on the same node and using efficient shared memory techniques between them. The success of this project has resulted in Nemesis being the default communication system for MPICH2 since 2008. Its success can be attributed to the intelligent use of topological information, i.e., use of the knowledge of when MPI processes exist on the same node. Under Nemesis, connections between MPI process pairs on separate nodes use a TCP connection per pair; each connection has separate flow and congestion control. Using MPI topological information in a similar way, a future version of Nemesis could aggregate the communications of all MPI process pairs spanning the same two nodes and make use of SCTP's multistreaming to share flow and congestion control, and for example, avoid the costly slow-start for each individual communication pair's congestion window.

In SCTP-based MPI middleware that utilizes multistreaming, more state must be managed. A user does not know what stream the next message it receives will be from,[1] and as

[1]Through the SCTP_RECVNXTINFO socket option that we added to the SCTP socket API [99], an application can enable a look-ahead to the next packet, if it is already queued in the kernel. In practice, no performance benefit was observed however this could change if internode MPI pairs were aggregated over the same association and multistreaming was utilized, as proposed in the previous paragraph with a future

a result, state must be kept in the middleware per stream because the timing of where an out-of-order message occurs cannot be predicted. Multistreaming enables the MPI application to precisely specify the data dependencies of an association using tags, however for this added flexibility, the MPI middleware must be more message-driven and store more state about the status of each stream.

3.1.3 Multihoming

A TCP connection is between exactly two network addresses whereas in SCTP, an association connects two sets of network addresses known as an endpoint. SCTP is the first IETF-specified approach for connecting sets of interfaces [101]. When several NICs are present, SCTP applications use one primary path and are able to transparently failover in the event of primary path failure to a secondary path.

It is typical on TCP-based MPI implementations that a listening socket will call `bind()` with `INADDR_ANY` to indicate that it will accept connections on any incoming address on that machine. If a machine has several network cards, then connections can therefore be established using any of the available paths. However, once a connection is established, it only makes use of a single path in TCP. No failover is provided by the transport in the event of a path failure.

When an SCTP association is established using a socket where `bind()` was called with `INADDR_ANY`, like TCP all the paths are available to first establish the association. However, unlike TCP, in SCTP the entire set of paths is also available after the association is established as well. During the association setup process of SCTP, the available interface set of each endpoint is communicated, and the paths are determined according to the configuration of the routing layer. Using `INADDR_ANY` adds all interfaces to the set of possibilities, however, to more precisely choose which interfaces to use, the application can

version of Nemesis.

call `sctp_bindx()` on whatever subset of addresses it desires; only this subset is included in the set of available interfaces for that socket. The specific ways multihoming was used in MPICH2 is discussed further in Section 3.2 whereas Open MPI's use is further discussed in Section 3.3.

3.1.4 One-to-One and One-to-Many Sockets

SCTP has two types of sockets: one-to-one and one-to-many. An SCTP one-to-one socket is similar to a TCP socket; a TCP socket is used for each TCP connection whereas similarly an SCTP one-to-one socket is used for only a single association. On the other hand, an SCTP one-to-many socket is similar to a UDP socket in that an incoming message can be from any sender from different associations. Unlike UDP sockets, messages on an SCTP one-to-many socket arrive reliably.

Using a TCP or a one-to-one SCTP socket in MPI middleware means that there is a socket for each pair of communicating ranks. A process that is communicating with more than one remote process will have a set of sockets to manage because data could be arriving on any of them at a given time. Typically, this management is done by calling `select()` on the set of sockets and performing the appropriate `send()`/`recv()` calls thereafter.

In addition to one-to-one sockets, SCTP also offers one-to-many sockets. This works in the same way that a UDP socket does in that an incoming message can be from any sender. Messages reliably arrive from different associations. It is up to the SCTP implementation to choose whether the socket buffer of a one-to-many socket is shared or split amongst associations [99].

SCTP-based MPI middleware has the option to make use of the one-to-many socket style. The advantages of its use are that less sockets are used and also there are less calls because no `select()` is necessary since there is only one possible socket for `recv()`/`send()`. In addition, there is an advantage to having only one single large socket buffer shared

amongst associations rather than multiple buffers which occurs when more than one socket is used.

Using one-to-many sockets in SCTP-based middleware involved changing the logic used by one-to-one implementations. Each MPI process maintains the state of their communications with other processes, keeping track of which messages are outstanding and those which are complete. With one-to-one sockets, there is a socket per process. Using a one-to-many socket, the one-to-one mapping of socket to MPI process no longer existed. The SCTP API does not allow to call `recv()` from a specific association; instead, a message is returned together with the association identifier. Based on this ID and stream, a received message is dispatched towards the correct receive buffer and the state is updated. Similar to MPI middleware's use of multistreaming, use of one-to-many sockets also makes the middleware more message-driven because a message is handled by the middleware as it arrives; its processing only depends on when it arrives as its handling is independent of which association it is on.

Both socket styles were used in our original LAM/MPI work in [53], however the predominant style used was to use the one-to-many style in our later work with LAM/MPI [54]. For the code that we released, one-to-many style versions were completed for both MPICH2 and Open MPI, as will be further detailed in the next sections. For Open MPI, we released code supporting both socket styles because it enabled some specific multirailing comparisons; this is further discussed in Section 3.3.

3.1.5 General Design Overview

SCTP takes over some of the duties middleware traditionally did, for example message framing (since SCTP is message-oriented) as well as some portions of the matching (when mapping tags to streams). Effectively, SCTP thins the middleware, or more specifically, it

thins the portion that does the actual message progression.[2] This general design was used in two specific SCTP-based MPI middleware implementations, which are described next, namely MPICH2 and Open MPI.

3.2 Our MPICH2 Implementation

Argonne National Laboratory releases a popular, open source MPI implementation called MPICH2 [2]. MPICH2 implements the full MPI-2 Standard and the code itself serves as the basis for several commercial MPI implementations. We have contributed code since the 1.0.5 release to enable SCTP for parallel MPI applications in the HPC community, and it continues to be maintained.

Our MPICH2 release has been used by other researchers, particularly those in the SCTP community. Since SCTP is a new transport protocol, there was a lack of applications in order to test a variety of communication patterns and interactions. In our case, we implemented MPI middleware that made use of unique SCTP features so as a result, we could thereby run any MPI program. The MPICH2 release itself comes with an intensive, built-in test suite consisting of hundreds of various MPI programs. Using this test suite together with our SCTP-based MPICH2 code, FreeBSD and Mac OS X kernel developers robustly test their SCTP stacks to help improve their performance; a protocol identifier for SCTP in MPICH2 has been registered with IANA to give guidance to network protocol analyzers for easier handling [44].

In the HPC community, MPICH2 is known for its high performance. There is no runtime notion of modules. The code's design does have different modules but module selection occurs at configure time prior to compilation; for example a specific "channel" like TCP can

[2]With some additions to SCTP and its API, and the right MPI design, the message progression layer could be eliminated and its duties all held responsible by the SCTP transport layer as was shown in my Masters thesis [82].

be chosen as the communication component that sends and receives MPI messages. Static compilation of modules in MPICH2 decreases flexibility but it also decreases overhead and therefore increases performance since module management and dynamic linking are not handled at runtime.

We developed our own SCTP MPICH2 channel called `ch3:sctp`; `ch3` is the name of the MPICH2 code which we incorporated SCTP into. In our implementation, we added the SCTP features mentioned in Section 3.1; our middleware design makes use of the maintained SCTP message boundaries, and multistreaming was mapped onto MPI tags using a per-stream state. Multihoming and one-to-many socket support were also incorporated, however unlike the approach in [54], new challenges were presented by the fact that MPICH2 provides a full SCTP-based MPI-2 implementation; these new challenges are described next.

3.2.1 Dynamic Connection Establishment

Static connection establishment schemes of N MPI processes open $O(N^2)$ connections across the system; many of these connections may be unused in some parallel algorithms. In order to avoid opening sockets to each other process immediately when an MPI application begins, we implemented a lazy connection scheme similar to the algorithm present in TCP-based MPICH2. This results in a savings of resources for applications whose communication pattern is not fully connected, since less resources will be used.

The dynamic connection scheme is more complex than a static system-wide connection algorithm. When an MPI process is launched, its socket is created and bound to a particular port which is returned back to the distributed management ring.[3] When a remote process wishes to establish a connection, the port for the destination process is queried from the management ring, and a new SCTP association is established.

There were several unique aspects to our SCTP-based dynamic connection algorithm.

[3]The management ring consists of a process manager per host interconnected using TCP in a ring topology.

For our algorithm, each MPI process has a single one-to-many socket.[4] This has several implications. Since multiple associations can use a one-to-many socket, the fact that the one-to-many socket API does not use an explicit `connect()`/`accept()` scheme and because `recvmsg()` cannot specify where to receive from means that any `recvmsg()` call can contain data from a new association. In our implementation, we ensure that the first packet read by a new association is a connection packet containing rank and context information. The new association ID is then added to a table that maps to the state information about this particular MPI rank.

For this dynamic connection algorithm, use of multistreaming complicates matters because messages on independent streams can overtake each other. It is therefore insufficient to send only a connection packet on the initial packet once for the new association and maintain valid multistream semantics because a message on a different stream could overtake that connection packet; if the first packet was not a connection packet, the MPI process will not know how to handle the new association. As a result, a connection packet needs to be sent for all uninitiated streams. On the receive side, when a duplicate connection packet is received by the algorithm, it is discarded.

3.2.2 Full Support for New MPI-2 Features

MPICH2 fully supports MPI-2 features including dynamic processes and one-sided remote memory access (RMA) operations. For dynamic processes, having dynamic connection establishment makes examples of this feature (e.g., spawning) less of a special case, since essentially we are handling the connecting of previously unknown processes by default. Once dynamic connections were supported, extending our SCTP channel to support dynamic processes followed naturally.

For the one-sided RMA operations, MPICH2 makes no assumptions about the hardware

[4] In the one-to-many socket case, there are only $O(N)$ sockets used throughout the entire system, whereas for TCP, there are $O(N^2)$ sockets used.

capabilities of the underlying system. For example, on standard Ethernet, accessing remote memory directly without the remote CPU's involvement does not occur. As a result, the RMA operations for the MPICH2 TCP channel provides an algorithm to complete the RMA operation over standard copy-in/copy-out Ethernet hardware. For our standard `ch3:sctp`, we make use of this same algorithm. However, in our recent work [106] which is presented in Section 8.1, we created a variant that we call `ch3:hybrid` where we emulated hardware with Remote Direct Memory Access (RDMA) capabilities, and on that emulated hardware we then made use of SCTP's features in order to more effectively implement the one-sided RMA operations.

3.2.3 Use of Multiple Network Interfaces

Prior to enabling SCTP inside MPICH2, two processes could be configured to connect to one another using for example, shared memory or TCP. However, when a host had two network cards, MPICH2 only made use of one path for the MPI communications.[5] SCTP supports multihoming so one association can make use of multiple paths for additional fault tolerance and performance. Since multiple interface support for MPICH2 did not previously exist, we added a way to specify which subset of network interfaces to selectively bind within an association between two endpoints. Under our mechanism, the interfaces an application wishes to use are defined in a file, and our channel reads this configuration file and calls `sctp_bindx()` to add each interface as desired, creating a subset of interfaces to use. Using standard SCTP, the multiple paths can be used for failover in the event of a failure, and using the CMT extension [48], the bandwidth of all of the available paths can be used simultaneously in MPI.

Overall, our `ch3:sctp` MPICH2 implementation is our most feature-rich and best per-

[5]The communication occurring between two MPI processes directly can use a different path than the process managers (called either "hydra" or "mpd") in MPICH2.

forming[6] SCTP-based MPI implementation. It has served as the basis for several of our own MPI variants as well including the versions executing on top of our userspace SCTP stack, simulated SCTP stack, and simulated RDMA-capable hardware which will be introduced respectively in Chapters 4, 5, and 8.

3.3 Our Open MPI Implementation

A consortium of companies, universities and laboratories came together and from 2005 onward, have collectively released an open source MPI implementation called Open MPI [35]. Their main goal is to provide a quick and easy framework for researchers to insert various runtime modules throughout the MPI implementation. When MPI applications are executed, a user can select which modules to use at runtime. The overhead cost of dynamic linking together with their module architecture is said to be fairly minimal [8] but this will vary on systems that have a high cost for instruction cache misses.

The exact tasks done by the runtime modules vary, but one task is to determine which protocol is used to communicate amongst MPI applications. The lowest layer communication primitive is called the byte transfer layer (BTL). A BTL instance typically corresponds to a single communication device. Some example BTL implementations included in the Open MPI main repository are shared memory, TCP, and Infiniband. We have implemented and contributed the SCTP Open MPI BTL so that it can also be used with this framework for use with this thesis, as well as more generally for the HPC community.

When an MPI program executes, instances of the BTL corresponding to devices are scheduled inside the MPI middleware by the BTL management layer (BML). For example, if an execution has two hosts, each with a TCP/IP over Ethernet network as well as an Infiniband network, applications will create an instance of both the TCP BTL and

[6] Experiments show MPICH2 performs best amongst our SCTP-based MPI implementations in Chapter 7.

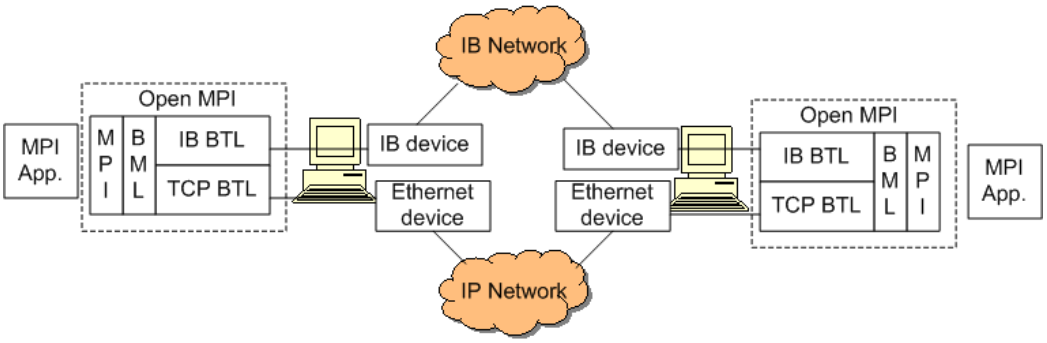

Figure 3.1: MPI application using two networks with Open MPI

the Infiniband BTL; this is shown in Figure 3.1. Inside the MPI middleware, the BML will schedule messages across both BTLs for simultaneous use. This approach is called middleware-based message striping, and it happens in Open MPI whenever there is more than one BTL instance between two communicating processes.

Our SCTP BTL has a unique design. As with MPICH2, we added the common SCTP features mentioned in Section 3.1 like message boundaries and multistreaming. The most unique aspect of the SCTP BTL was that we give the user the choice to perform message striping in the middleware, or instead let the SCTP transport protocol handle all of the message striping by way of multihoming and CMT. In effect, our SCTP BTL is like two BTLs in one. This design allows quantification of the advantages of the two multirailing approaches.

The choice of where to perform multirailing is provided by internally adjusting the multihoming settings, the addresses bound, as well as the socket style. In Open MPI, doing message striping at the middleware layer implies multiple BTL instances. The scheduling occurs amongst BTL instances, typically where each instance binds to an interface on a separate network. Since a BTL exports a socket interface, each BTL requires its own unique socket; this is incompatible with a single one-to-many style socket on a host that can send to and receive from any endpoint because the SCTP one-to-many API does not let the receive

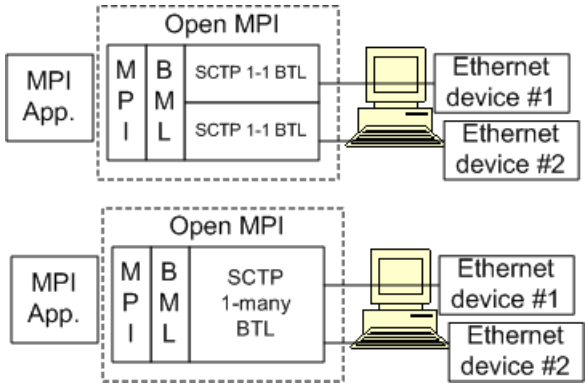

Figure 3.2: Use of SCTP BTLs in Open MPI

call specify the source address to receive from. In order to take advantage of message striping in the middleware, we implemented one version of our SCTP BTL using SCTP's one-to-one style sockets where each of those one-to-one sockets are selectively bound to a single IP address using the `sctp_bindx()` API call. The BML then schedules amongst these BTL instances. This is shown in the upper half of Figure 3.2.

The one-to-one implementation is strictly used for middleware-based message striping (one interface per BTL instance) while the one-to-many design is used for transport layer multi-railing using the CMT SCTP extension [48]; these two designs will be compared in the Chapter 7 which focuses on performance. The one-to-many design is illustrated in the lower half of Figure 3.2. In the one-to-many BTL, all the specified IP interfaces are used by the one socket opened by a single BTL. Only one BTL instance is created, so to the Open MPI middleware-based scheduler, using the one-to-many implementation means that no message striping occurs in the middleware. Our original MPICH2 `ch3:sctp` uses a similar one-to-many design. The CMT extension can then be enabled for multi-railing inside the transport layer using a `sysctl` setting.

3.4 Summary

Overall, the design, implementation, and subsequent release of SCTP-based MPI middleware for MPICH2 has introduced a tool useful for SCTP developers and MPI users alike. On the other hand, our SCTP-based Open MPI BTLs enable the comparison of multihoming in the SCTP transport to data striping done in the MPI middleware because there is a minimal difference in the two implementations otherwise. Our Open MPI design as well as our MPICH2 SCTP design and its derivatives are summarized in Table 3.1 together with a reference to the section where more information can be found.

The initial SCTP-based MPI design used in this chapter was first published with its performance results for MPI-1 in [54] in the Proceedings of Supercomputing 2005, the biggest annual conference in HPC. The design was later adopted during an internship at Argonne National Laboratory to successfully pass the MPICH2 [2] test suite of hundreds of test cases. This demonstrated the compliance of our code so our code was included in the public release of the full MPI-2 release of MPICH2 [2] version 1.0.5; this release served as the foundation for our own MPI variants executing over top of our userspace SCTP stack, simulated SCTP stack, and simulated RDMA-capable hardware which will be introduced respectively in Chapters 4, 5, and 8. Together with the Open MPI implementation which has also been made publicly available, a family of tools has been introduced that is beneficial for use in this thesis as well as for others in the SCTP and HPC communities.

Middleware	**Module Name**	**Description**	**Section Described**
Open MPI	SCTP 1-1 BTL	Uses middleware-based scheduling for multirailing.	3.3
	SCTP 1-to-many BTL	Uses transport-based scheduling for multirailing.	3.3
MPICH2	`ch3:sctp`	Original SCTP-based channel using SCTP socket API.	3.2
	`ch3:copy*`	Userspace SCTP stack MPI with internal copy.	4.2
	`ch3:mbuf*`	Userspace SCTP stack MPI with `mbuf`s internally.	4.2
	MPI-NeTSim	Runs over simulated SCTP stack and networks.	5.1
	`ch3:hybrid*`	Uses simulated SCTP-based RNIC for MPI-2 RMA operations, and standard SCTP for all other MPI calls.	8.1.2

Table 3.1: MPI middleware versions and configuration properties, an asterisk (*) indicating a privately-used name for code not in the official release

Chapter 4

Userspace SCTP Implementation

A userspace SCTP implementation has been developed for use with this thesis. In this thesis, the userspace stack provides an infrastructure for the investigation of transport protocol onloading through a tighter coupling of the transport stack with the MPI application. It also provides a tuned, feature-rich SCTP implementation that is capable of executing on all operating systems including Linux, the most popular commodity operating system in HPC. The Linux kernel SCTP implementation [59] is incomplete, lacking some more recent RFCs and other optimizations. Our userspace stack enables these investigations for this thesis while generally providing an environment for easier protocol development, debugging, and instrumentation compared to kernel development. Our userspace SCTP stack has been contributed to the open source community, where additional benefits have been found for its use outside of HPC, for example, it has been used to deploy SCTP on to the iPhone since this device does not grant developers access to write kernel code.

Recently there are several developments that have made revisiting userspace stacks of interest. First, there is a renewed interest in protocol onload as protocol off-load devices arguably were a point-in-time solution that are less general [85]. There has also been more recent work starting to question whether transport protocols are too large to belong in the kernel. Van Jacobson made the argument that cache use can be improved with a user-level stack [50] on a multicore host because a user-level stack would ensure the transport processing is done on the same CPU as the application; this changes the "end" of the end-to-end principle common in network design from the host to the actual process or thread

on a specific core, as was similarly proposed by Siemon [97]. Another development is that the newer generation of network adapters have started to have more support for network virtualization giving easier data paths to the NIC and providing protection domains on the NIC [47]. In addition, the latest NICs also support an on-board CRC32c checksum computation, useful for iSCSI as well as SCTP, which eliminates a potential performance overhead posed by SCTP's stronger checksum.[1] The combination of these advantages make it an opportune time to re-visit the question of how to make a fully-featured userspace SCTP stack.

Our userspace design required several unique solutions in order to function efficiently in userspace. However, the core parts of our user-level SCTP stack still share the exact code as is compiled inside the FreeBSD kernel, which is the most performant SCTP kernel implementation. As such, we will be able to continue to take advantage of continued improvements and added features to the FreeBSD stack with our userspace design. Although there have been several simple implementations of transport protocols in simulation or in userspace for TCP [28, 57, 86, 98], none of these have been full-featured, performant, device agnostic as well as portable across multiple operating systems, as what we have accomplished simultaneously for SCTP. As a result, our work is also a good test to evaluate the potential benefits for protocol-onload. In Section 4.1, we describe our SCTP stack design then, in Section 4.2, we describe how we have integrated the use of our userspace SCTP stack into the MPI middleware. A performance summary is provided in Section 4.3. A summary is offered in Section 4.4. We know of no previous work that has considered opportunities for tightly coupling a standard transport protocol implementation with MPI middleware. Overall, our userspace stack enables more options for performant execution on Linux as well as the potential for a tighter coupling of the SCTP transport protocol with MPI applications.

[1] Ethernet controllers with CRC32c offload include the Intel 82576 chip for Gigabit Ethernet and the 82599 chip for 10 GigE [46], formerly known as Niantic.

4.1 SCTP Userspace Stack Design

In order to create an SCTP stack that would perform better on Linux, a survey of potential starting points was conducted. The *sctplib* [109] already executes on Linux, however, for our purposes, it lacked the throughput and latency measurements that we wanted to achieve and also lacked the functionality that is present in the full implementation of the SCTP standard as it has lagged behind the RFCs. There are kernel versions of SCTP for all major operating systems but the version of SCTP with the best performance and that is the most feature-rich has been the FreeBSD stack that was originally developed by Randall Stewart, a co-inventor of SCTP. Linux has its own kernel implementation however it has not been optimized nor consistently maintained. The Java SDK supports SCTP but it utilizes the underlying kernel implementation on the host operating system [20]. The code used for the kernel FreeBSD SCTP stack has been used for both a Mac OS X [108] and Windows [23] version of the stack, which made it a good starting point for our work.

A representation of a kernel-based SCTP stack is shown in Figure 4.1-(1). An application uses an SCTP stack in the kernel by way of the Berkeley sockets API. Within the SCTP/IP stack, the transport protocol implementation forms a valid SCTP packet and passes it to the IP layer that then performs routing table lookups for outward bound packets; inward-bound packets are demultiplexed to the appropriate tuple and eventually that application's socket. Within the kernel, SCTP/IP interacts with the NIC controller by way of the device driver.

Under a kernel-based design, the host CPU processes the transport protocol in the operating system kernel, makes intermediate buffer copies, and also performs context switches between userspace and kernel space. As network speeds increase, these overheads cause an increased burden to the host CPU. Networking-related CPU overheads for a kernel-based TCP stack are measured to be 40% for transport protocol processing, 20% for intermediate buffer copies, and 40% for application context switching [43].

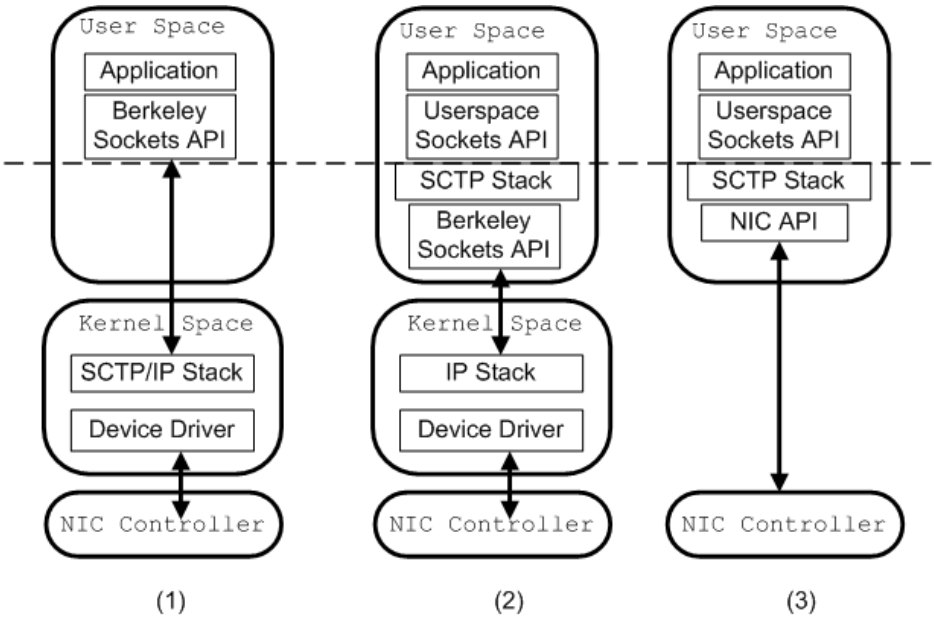

Figure 4.1: SCTP implementation possibilities

Additional copies can be avoided by using zero-copy between networking layers (e.g. TCP and IP) to bypass the kernel [18], as is shown in Figure 4.1-(3). This design avoids unnecessary context switches to/from kernel space because all operations are done in userspace or by the device. Copying is avoided by passing references in-between layers using either a slab allocator between software layers, or to hardware by way of a NIC API's zero-copy read/write functionality.

The major design challenge is to find general ways to reduce these overheads. Over the past years, several companies (e.g. NetEffect[2]) took advantage of the inability of the host OS to perform protocol processing for GigE and 10 GigE by TCP protocol offloading to the NIC card itself to achieve both zero-copy and kernel bypass. Typically, TCP offload devices are all-in-one solutions, although the Microsoft Chimney Architecture [51] has attempted to standardize the integration of TCP offload devices with the Windows operating systems.

[2]NetEffect was acquired by Intel in October 2008.

Nevertheless, transport offload solutions were more of a "point in time" solution [85] and more recently multicore and virtualization technology makes it easier to provide kernel bypass and more generic support on the NIC for protection and abilities such as Large Segment Offload (LSO), which can be used to achieve the performance gains of an offload device on the chip itself.

The basic goal to achieve the best performance was to attempt protocol onloading by moving the protocol stack to userspace, as is illustrated in Figure 4.1-(2). Moving only the transport protocol into userspace is intermediate to the overall final goal of kernel bypass; no one has done full kernel bypass for SCTP but it has been done in a device-specific manner for TCP [98].

There is merit in moving only the transport protocol into userspace, in addition to it being a path towards kernel bypass. Moving only the transport protocol into userspace makes the SCTP stack device-agnostic so it is more portable yet it is a good feature-rich implementation of SCTP at the user-level, as an alternative to the less tuned Linux SCTP stack as well as the *sctplib* userspace stack [109]; this also makes it possible to run SCTP on small, mobile devices such as cellular phones that only allow userspace-level development. Here we describe the design of our userspace stack that we use for protocol onloading, comparing it to how the same code works in the kernel. We first describe the lower-layer protocol (LLP) interactions then the upper-layer protocol (ULP) interactions, both pictured in Figure 4.2. Stack internal implementation issues for our userspace SCTP stack are then shared before we summarize our design.

4.1.1 LLP Interactions

As is shown beneath the SCTP stack in Figure 4.2, the SCTP implementation needs to interact with the layer below it. SCTP was originally specified over IP as its own transport as this was seen to be the architecturally correct solution by the SIGTRAN working

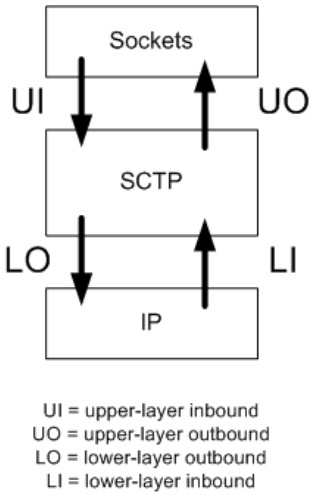

Figure 4.2: SCTP stack interfaces and directions

group [101]. However UDP encapsulation [107] has been specified in case SCTP traffic needs to pass across legacy firewalls or if it needs to run on hosts that do not provide direct access to the IP layer such as the iPhone. Our userspace stack is capable of handling SCTP protocol data units (PDUs) layered over IP or UDP/IP, so all LLP interactions respectively occur using either a single raw IP socket that filters all SCTP traffic or a single UDP socket bound to a known tunneling port.

LLP Outbound – Throughout the stack, LLP outbound (LO) interactions pass SCTP PDUs downward towards the wire. When an application above SCTP sends data destined for some remote application, an LO interaction occurs. Elsewhere, the protocol generates LO interactions when it needs to pass some necessary control information. This happens when, for example, the protocol specification requires a Selective Acknowledgment (SACK) to be sent to acknowledge the receipt of data.[3] All LO interactions in the stack call an

[3]In SCTP, Selective Acknowledgments are the mandatory acknowledgment mechanism whereas in TCP, SACK is an alternative to cumulative acknowledgments. SCTP needs a more expressive acknowledgment scheme because data arrives out-of-order more commonly due to multihoming [101].

IP_OUTPUT macro which in our userspace stack, we implement as a simple `sendmsg()` call used with a raw socket for SCTP/IP or a UDP socket for UDP encapsulation. Therefore, like the kernel SCTP implementation, we presently make use of the kernel IP layer for routing as well as interfacing with the NIC.

LLP Inbound – When packets arrive from the wire and progress up into the SCTP stack in Figure 4.2, an LLP inbound (LI) interaction occurs. LI interactions occur unpredictably, however, it is important that the SCTP PDUs are handled promptly as the internal protocol state of the association is time-sensitive. In the kernel SCTP implementation, SCTP PDUs enter the SCTP stack by way of the `sctp_input()` method which is registered as a callback upon initialization of the network stack. This callback is fired when an SCTP packet arrives to the kernel IP implementation or to the assigned UDP encapsulation port.

In a userspace SCTP implementation, no such callback is registered as the kernel is operating within its own protection domain. Nonetheless, there is still a need to react responsively to LI interactions; in order to provide this, a thread is used to poll for the asynchronous arrival of SCTP PDUs. We use the portable `pthread` library to create a thread that polls with a blocking `recv()` on each lower-layer socket and passes the SCTP PDU into `sctp_input()`. We have one thread that filters SCTP/IP packets by way of the raw socket and in addition, we have another thread that filters UDP-encapsulated SCTP packets on our UDP socket.

The main difference between our userspace stack and the kernel stack is that LI interactions cross the kernel boundary in our userspace version whereas for the kernel, the LLP boundary is internal to the same protection domain inside the kernel. When inside the same protection domain, a callback mechanism can be used for asynchrony, as is the case with the kernel SCTP implementation. On the other hand, for our userspace SCTP implementation, we make use of the kernel IP implementation which is in a different protection domain so we cannot use a callback mechanism for LI interactions to execute userspace code

inside the kernel. Our thread calls `recv()` which uses Berkeley sockets thereby crossing into kernel space for the IP implementation. A wake-up occurs to traverse this kernel-userspace boundary to notify the blocking `recv()` call that an SCTP PDU has arrived and been placed in the lower-layer socket buffer.

4.1.2 ULP Interactions

At the upper layer, much of the FreeBSD kernel code that the userspace stack is based on assumes it is going to use FreeBSD's implementation of the Berkeley sockets API. Within the SCTP stack itself, the structures used to implement sockets in FreeBSD are intertwined throughout the code. Many different socket-related functions and structures are used extensively throughout the SCTP stack. In order for the SCTP stack to operate in userspace as shown in Figure 4.1-(2), these socket structures were exposed to userspace. We implement these socket structures and their related methods that are used by the SCTP stack itself.

In Figure 4.1, ULP interactions to the various SCTP stack implementations are represented by the dashed horizontal line. In the methods we exposed to userspace, these ULP interactions use locks to signal between the application and the transport stack. When an application no longer has to block on either a send or receive, a wake-up occurs via these locks from the stack to the user.

ULP interactions happen between the userspace stack and the application by way of our initial API which uses a `userspace_` prefix to the known Berkeley sockets API to denote the name of the methods we implemented with the semantic equivalent to their Berkeley socket counterparts. This custom socket API is listed in Table 4.1. The disadvantage of using a different function name than the Berkeley socket API is that this requires applications to be ported to use our API directly. However, our intended use is MPI, so only the middleware implementation will have to change to use our API; the MPI programs themselves will retain their portability. Use of our API for within the MPI middleware is discussed in

Function	Description
`userspace_socket`	Returns a userspace socket.
`userspace_bind`	Binds a particular port and address set to a userspace socket.
`userspace_listen`	Enables server-side capabilities of a socket.
`userspace_accept`	Blocks until it returns a new userspace socket on a listening server.
`userspace_connect`	Connects to a remote userspace socket.
`userspace_sctp_sendmsg`	Sends data on a userspace socket.
`userspace_sctp_recvmsg`	Receives data from a userspace socket.
`sctp_setopt`	Allows the setting of socket options on a userspace socket.

Table 4.1: SCTP userspace custom socket API functions

detail in Section 4.2.

4.1.3 SCTP Implementation Internal

Inside the SCTP implementation itself, several other items needed to be implemented for the userspace implementation to operate on all platforms.

Memory Allocation – A transport protocol needs to avoid excessive copying and to quickly allocate/deallocate memory for SCTP PDUs. Inside of the kernel, PDUs are passed between the transport layer and below to the wire without an excessive number of copies; as the PDUs cross these layer boundaries, using an internal structure maintains a reference count and a pointer to the PDU's memory. In the Linux kernel, these memory management structures are called `sk_buff`s [64] whereas in FreeBSD-based systems, they exist within the kernel known as `mbuf`s [104]. These structures are initially allocated in their respective kernel making use of a slab allocator to avoid fragmentation. This is done

through its object caching strategy that is used when allocation and deallocation of memory of the same type/size is happening frequently, as is the case with `mbuf`s within a networking stack. Chunks stay in a per-CPU cache.

`mbuf`s are used throughout the FreeBSD SCTP kernel stack which our userspace SCTP implementation is based, so we re-implemented `mbuf`s and their support functions in user-space. Our stack can be configured to provide `mbuf` allocation using either `malloc()` from the heap or using the user-level libumem slab allocator [75], the latter of which is the default option for better performance in clusters whereas the former is used for smaller devices and for easier debugging.

Timers – As with any transport protocol, there are a number of timers needed to ensure reliable transmission as part of SCTP's state machine. A transport layer needs to keep track of time, deciding when to make responses or queries. An example of this is when establishing a connection. An INIT packet is sent and if the INIT-ACK is not received within a timeout, then the INIT needs to be resent. A more common example is a DATA chunk; if it is not acknowledged by a SACK before a timeout, this DATA chunk needs to be resent.

In our implementation, we provide the ability to schedule and deschedule timed transport interactions using a callout queue that runs in its own thread. This thread has an event loop and it maintains all timed events, firing the appropriate callbacks at their expiration times. If an event is not canceled by another thread before that event timer expires, then it is serviced by the event loop when firing the associated callback at the desired time. This same approach is used in the FreeBSD kernel implementation.

The overhead of our timer implementation is minimal. Timeout values for a transport protocol are typically on the order of tens of milliseconds so we set our timer thread's event loop to only be awaken every 10 ms.[4] This is very coarse for a modern 2 GHz CPU to handle

[4] A timer with a higher resolution could be used if there was a desire to set the timeout values to a smaller numbers.

and therefore comes at a very low cost since it can handle 2.10×10^7 instructions elsewhere in 10 ms. When the thread awakes, it checks to see if it must deliver any expired timers for specific events. However, the majority of the time under standard operating conditions, there are no expired timers as their conditions have been met by packets received by the LLP thread. When conditions of a timeout event are met from any thread in the stack, their events are canceled and removed from the event loop, so even when the timer thread awakes, it typically has very little to do.

Platform Specific – Different operating systems have their own idiosyncrasies. One difference is Linux puts the IP length into network byte order while other platforms do not. Another difference is in FreeBSD and Mac OS X, the length of the socket address structure, `sin_len`, is included in its length calculations while in Linux, it is not.

A final example of a difference across platforms is atomic operations. Because we have several threads operating on the same common data, we needed to add a locking solution. This was accomplished by creating an implementation for the ATOMICS macros used throughout the SCTP implementation. For Mac OS X, we made use of the `OSAtomic` interface where as for other platforms, we made use of the atomics built-in to GCC versions 4.1+ named `__sync_fetch_and_{subtract,add}`.

4.1.4 Userspace Stack Design

We now have a stack capable of running custom-built applications using our userspace SCTP stack. The SCTP portion of our stack runs entirely inside the same process as the application using it. State for application socket mappings or ports does not need to be communicated across the userspace-kernel boundary. Each application has their own instance of the complete stack, so there is no persistent state for default transport configuration settings. Changing most values for an application therefore requires a code change, however for convenience we read commonly used settings such as UDP encapsulation from

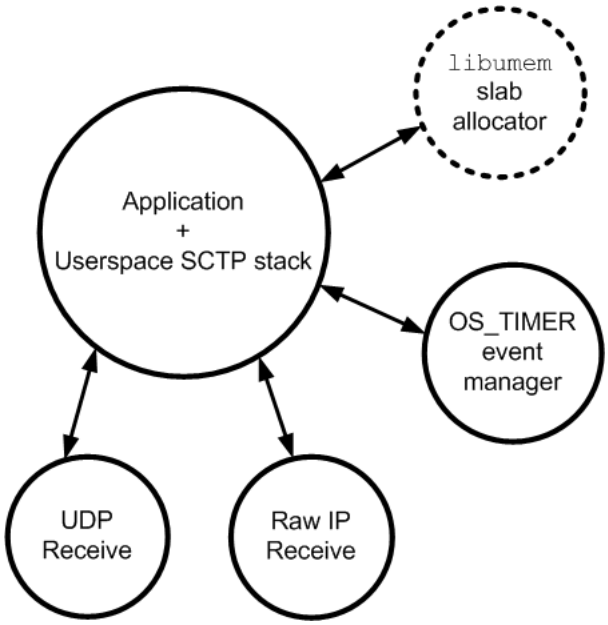

Figure 4.3: Userspace SCTP implementation threads

an environment variable. Future designs could similarly store the values in a configuration file or on a persistent daemon.

Figure 4.3 summarizes the threads used in our userspace stack. There are potentially five threads in total used to implement our userspace SCTP, but potentially only four depending on the choice of memory allocator as libumem internally uses a thread and `malloc()/free()` do not. Under standard operating conditions, the timer event manager thread is inactive as well as is the receive thread that is not being used, e.g., the raw IP receive thread if we our application is running encapsulated over UDP.

Recently, this code has been contributed to the SCTP community by having committed to the common repository for the Windows, Mac OS X, and FreeBSD SCTP stack code. With our changes and use of preprocessor directives, we do not prevent the original code from working on their respective kernel platforms, so long as they are configured directly. Since it is put in this repository, going forward, our userspace stack continues to inherit

future updates to the FreeBSD kernel stack.

4.2 MPI Design Using Our SCTP Userspace Implementation

Our SCTP userspace implementation exports an API but the fact is no real applications use this API. In this section, we incorporate our userspace stack into our custom design of MPI middleware so that we in turn can execute all MPI applications over our userspace stack without modification.

Our new MPI design intertwines our SCTP implementation with the MPI application into the same Unix process amongst a handful of threads, as illustrated in Figure 4.3. A more typical design like our kernel-based SCTP design described next has the MPI portion in userspace and the transport in the kernel. There has been related work where portions of the MPI message processing has been pushed into the kernel; in [61], the expected and unexpected message queues were pushed into a custom kernel whereas in [82], the unexpected queue was pushed into the transport. Compared to these, the userspace MPI designs presented below are completely the opposite, where we are trying to pull everything up into userspace for flexibility, generality, and performance.

4.2.1 Kernel-based SCTP

For reference, our original MPICH2 `ch3:sctp` [54] which was presented fully in Section 3.2 is shown at the top of Figure 4.4. This design illustrates the design of the MPI middleware when making use of a typical kernel-based SCTP implementation as shown in Figure 4.1-(1). For `ch3:sctp`, the `MPI_Send()` passes in a buffer from the application which inside the middleware is pointed to by the `iovec` which is the structure used internally within the middleware for storage as well as being used by the scatter/gather system calls i.e.,

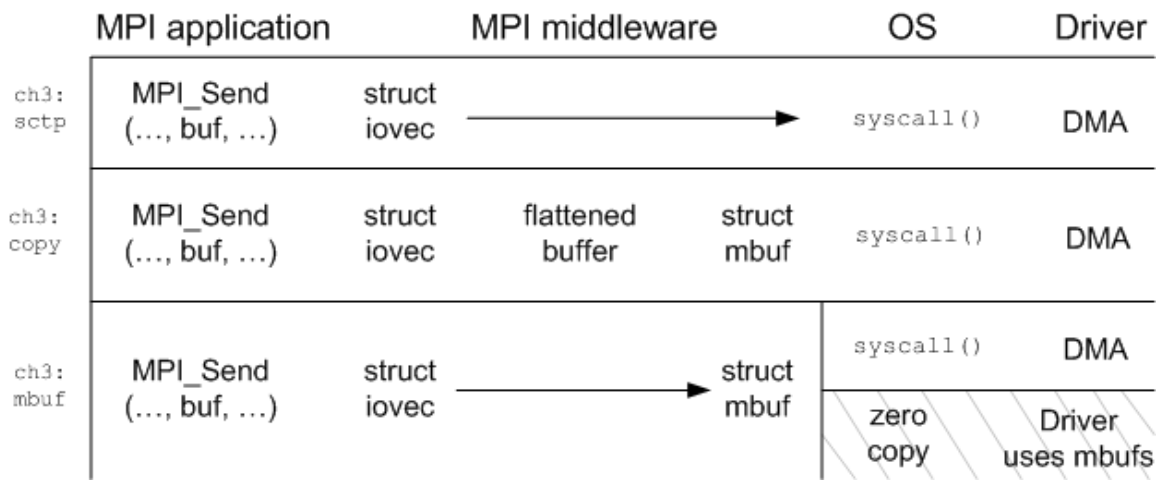

Figure 4.4: Data copies with SCTP-based (`MPI_Send()`)

`writev()/readv()`. Later the `writev()` system call is executed to copy the `iovec` into the kernel, requiring a context switch. Inside the kernel, the device driver eventually copies it again by way of DMA to the wire. In this design, from the user to the wire, there are two copies and one context switch.

4.2.2 Userspace-based SCTP with Copies

We have designed `ch3:copy`, as a variation of MPICH2 `ch3:sctp`. Shown in the middle of Figure 4.4, this design uses our device-agnostic userspace SCTP stack. The userspace stack internally uses `mbuf`s which require a flattened buffer as opposed to a vector for use. The flattening of the buffer requires an extra copy. Inside our `userspace_sctp_sendmsg()` call, the `mbuf` chain is created from the flattened buffer and then internally calls `sendmsg()` on the raw socket used at the lower-layer of our userspace SCTP stack. This system call performs a context switch copying the data to the kernel. Again, inside the kernel, the device driver eventually copies the buffer again by way of DMA to the wire. In this design, from the user to the wire, there are four copies and one context switch. Obviously, the

excessive number of copies is a problem that needs to be solved.

4.2.3 Userspace-based SCTP using `mbuf`s Directly

We reduced the number of copies from the `ch3:copy` design in our next version of MPI called `ch3:mbuf`, shown last in Figure 4.4. For this design, we converted our middleware to use `mbuf`s internally for storage. This is the same structure passed throughout the userspace networking stack. When the middleware uses `mbuf`s, they can be passed without a copy into the userspace SCTP stack by way of our `userspace_mbuf_send()` call which accepts an `mbuf` chain rather than a flat buffer. Avoiding the flattening of the `iovec` saves a copy.

In Figure 4.4, the OS and Driver portions of this implementation show two paths for `ch3:mbuf`. At present, the top path is what is implemented as it is device-agnostic. Specifically, a system call to the raw socket is still made to the kernel requiring a copy by way of the context switch. Again, inside the driver, the information is copied by way of DMA to the device. This path requires three copies and a context switch, eliminating one copy compared to `ch3:copy`. However, this path is still not as good as the kernel version `ch3:sctp` because an extra copy is needed to create the `mbuf` chain for the userspace SCTP stack; otherwise, two of the copies and a context switch are still necessary by the userspace stack for making use of the kernel IP and device driver implementations.

However, the bottom path of `ch3:mbuf` in Figure 4.4 illustrates the potential advantages of using `mbuf`s internally to the MPI middleware. Some devices, particularly those in FreeBSD and Mac OS X, also use `mbuf`s internally. In addition to this, certain devices export a zero-copy API that are passed an `mbuf` chain without the kernel's involvement. For these particular devices, using this same design as is, the `mbuf` chain created in the middleware is passed directly to the device without a system call (i.e., context switch) as well as without any copies of the data from the middleware all the way to the wire. This path requires one copy to create the `mbuf` chain from the MPI application's supplied buffer,

but no context switches.

4.3 Performance Summary

This chapter focuses strictly on the design of the userspace SCTP stack, however, its performance is evaluated in Chapter 7 with the rest of the performance analysis of this thesis. The receive-side and send-side callback optimizations are introduced in Section 7.2.1; the effectiveness of the receive-side callback is measured for a bandwidth benchmark in Section 7.2.2 whereas the benefit of a threshold for when to trigger the send-side callback is shown for the same bandwidth benchmark in Section 7.2.4. With the implementation presented in this chapter above, the bandwidth achieved depends on the size of the message but typically it is half that of the Linux TCP or SCTP kernel implementations, as is shown in Figure 7.6. However, with the callback and threshold optimizations, our userspace stack outperforms the Linux kernel SCTP stack and nearly achieves the bandwidth of the Linux kernel TCP stack when using a single network interface, as is shown in Figure 7.8 on page 101. Section 7.2.5 shows that the latency of our userspace stack is similar to kernel implementations when a slight driver modification is made. Real MPI applications achieve the same performance using our userspace SCTP stack as they do with the TCP kernel implementation, as is shown in Section 7.3.1. Overall, the performance of our userspace SCTP is quite promising and this stack provides different operating systems with the addition transport features that enable further improvements to Ethernet-based MPI systems.

4.4 Summary

Overall, this project produced the necessary infrastructure to investigate this thesis by providing a tight coupling between custom MPI middleware to a complete, full-featured

userspace SCTP stack operating on Linux and other operating systems.[5]

This SCTP userspace stack itself was a major contribution as a tool for use in this thesis and by the SCTP community as a whole. The novel solutions presented here extracted the best kernel implementation of SCTP from FreeBSD and provided the necessary parts in order to make all of its features usable across all network devices and platforms in a performant manner; these achievements had never been met simultaneously by a userspace stack for SCTP [109] nor TCP [28, 57, 86, 98]. For the thesis, the userspace stack enables the investigation of protocol onload for SCTP. This code has been contributed to the SCTP community and has been used to enable use of SCTP on the iPhone. The design presented in this chapter remains to be published as does the performance results of this stack which are shown in Chapter 7.

[5]This coupling will be further demonstrated with a callback performance optimization for the userspace SCTP stack introduced in Section 7.2.1.

Chapter 5

Simulated SCTP Networks with Emulated MPI

In this chapter, we introduce another tool used in this thesis called MPI-NeTSim. MPI-NeTSim is a hybrid, software-in-the-loop simulator that allows MPI programs to execute unchanged in emulation communicating over top of an SCTP stack and network, both simulated at the packet level. MPI-NeTSim enables the easy variation of the network, making it possible to run MPI applications on different networks. In addition, transport settings can also be adjusted. Based on these settings, MPI-NeTSim executes the MPI application and then provides a means to centrally gather complete statistics and traces from throughout the network. In effect, this tool eases the process of understanding the impact of SCTP's new features for MPI.

When testing MPI applications, it is common to vary the number of processes, thus impacting the number of cores and the amount of memory used. By varying the number of processes, users experimentally determine the computational requirements and memory footprint of an application. Different applications or data set sizes can be experimentally investigated in order to understand the computational requirements, but typically the underlying network and transport settings remain constant. As such, it is difficult for users to investigate the communication requirements in a way similar to how they presently determine the computational requirements of scaling the program to more processors. Given our focus on the study of network properties of MPI programs, we need an easy-to-use tool

that enabled the exploration of communication requirements for MPI programs.

Varying network and transport configurations and studying their effects in real networks is difficult. If a transport feature like Nagle's algorithm is to be toggled, it could be done by adjusting the code of the distributed MPI application or by changing the `sysctl` configurations on each end host. Users require special permissions on each end host to change transport settings as well as to obtain network traces and statistics for deeper communication analysis. Traces from middleboxes like routers require additional permissions. These settings, permissions, trace files, and statistics are spread out across the cluster so the sum of them can be cumbersome to manage and gather, although not impossible.

While some configurations are difficult to manage, some are actually impossible to manage. For example, SCTP supports multihoming so to test its fault tolerance on different topologies would require a physical change to some clusters to have two networks instead of only one; adding a second network may be a physical impossibility due to the unavailability of equipment.

MPI-NeTSim allows us to easily obtain data to study the impact of SCTP and network parameters on MPI communications. Our solution executes the MPI processes and passes the data over simulated networks using a simulated SCTP implementation. The transport and the network itself behave according to the settings given to the simulator in a central configuration file, so they are all managed in one place. All processes execute at user level, without the need of special permissions for devices or instrumentation, yet network traces and statistics from end hosts and middleboxes are still available. In addition, topologies can easily be adjusted to model extra links for SCTP multihoming, and latencies and bandwidths can be adjusted without the need of any additional wiring or equipment.

Modeling complex MPI programs is difficult, takes a lot of time to do right, and can be inaccurate because the communication patterns can be intensive and even dynamic therefore models may be an over-simplification compared to the execution of the real program. Instead of an application model, the goal of MPI-NeTSim was to drive the simulation by

executing unmodified MPI applications. The MPI application communicates by way of the simulator and it is the events generated by this communication that act as an input to the internal transport and network model. We will show that this transport and network model have been carefully validated for accuracy in order to ensure that its model captures the desired properties of the real scenario that it is emulating.

There have been previous approaches where simulation was used as a component in the execution of distributed applications. One common approach is to use dummynet [92] to add delay or reduce bandwidth between machines in the system. Because only the underlying system changes, this approach requires no modifications to MPI, and for latency and bandwidth in particular, one can obtain accurate results. While in the past, we have used dummynet to investigate SCTP behavior [54], this approach has been limited due to the lack of availability of dedicated equipment. Configuring dummynet requires superuser privileges and, depending on the type of devices to be simulated, additional machines to emulate the network devices may be required. When dummynet is used, one has to use an appropriate kernel configuration to provide enough memory to store all the packets and to use an appropriate system clock granularity. This requires building a custom kernel. Systems like Emulab [29] configure dummynet for you by providing a graphical front-end, but this requires having access to a fully configured Emulab cluster. Our approach can be done completely at the user-level and does not require any special access or additional machines.

There have been several projects for the simulation of MPI programs. Riesen [91] describes a hybrid approach which combines both emulation and simulation. Similar to MPI-NeTSim, Riesen's system executes the MPI program as before and has the middleware send communication events to a discrete event simulator. The discrete event simulator only processes events, and unlike MPI-NeTSim, is not used as a communication device. The simulator is a custom one and does not simulate at the packet-level of IP transport protocols. The advantage to Riesen's approach was the ability to scale to a larger number

of MPI processes, but that is partially due to not performing a more detailed simulation of the network. Riesen's approach more closely corresponds to a program-driven simulation rather than emulation. Our need was to have a packet level simulator; we chose to use OMNeT++[114].

There have been other projects that simulate MPI programs.Nunez et al. [72] use OMNeT++ to simulate MPI programs where the focus was on storage I/O requirements. They gather trace files from an MPI program and use these traces to perform a trace-driven simulation of the execution. They describe a network simulator with similar goals to our own [112]. The major difference is that they simulate only the network adapters and not the internals of the network. In addition, the simulator does not use standard protocols but assumes a simple transport mechanism. As well, there is work by Prakash and Bagrodia [87] which exemplifies the approach of building a dedicated simulation environment for MPI. Again, this does not model standard transports and is yet another closed, custom system with a fixed number of parameters used to set the properties of the underlying network.

Our MPI-NeTSim tool extends the open-source, discrete event OMNeT++ network simulator [114] and has been committed to the OMNeT++ INET repository at the Münster University of Applied Sciences, so it is available for others to use. Our tool leverages several other research projects based on the popular OMNeT++ simulator. The OMNeT++ INET framework [113] framework containing full, packet level simulation implementations of IP, TCP, and UDP was extended by Rüngeler et al. [93] to include SCTP; it is this SCTP implementation that MPI-NeTSim uses. This original SCTP implementation executes modeled applications, however it was extended by Tüxen et al. [110] so that an endpoint of a client-server modeled inside OMNeT++ could communicate via an external network interface to another endpoint operating on another machine using its own SCTP implementation; it was this external interface implementation that served as a starting point for MPI-NeTSim which is a system where the processes of a real MPI application take advantage of the simulated SCTP stack and network.

In order to enable the use of the simulated transport and network by MPI applications, two major problems had to be solved. The first problem was a matter of functionality; we needed to design an interface that opened up the simulator's SCTP implementation enabling its use by real Unix processes outside of the simulator. Once this interface was defined and added to our MPI implementation, the second major problem was a matter of behavior; we needed to solve how to accurately use the MPI implementation's communication events as input to the OMNeT++ event-driven simulation of the network.

In the design of MPI-NeTSim, we have solved both problems, functionality as well as behavior. In Penoff et al. [84], the functionality problem was solved as well as an initial novel idea for solving the behavior problem using time dilation. In later work in [94], our initial algorithm to solve the behavior problem was vastly improved to be more adaptable and therefore more scalable. The end result is that MPI-NeTSim serves as a useful tool to easily study the outcome of varying topologies and SCTP parameters for MPI programs, as is done in our studies on reliability and performance, in Chapters 6 and 7 respectively.

In the rest of this chapter, we describe the details of MPI-NeTSim. The discussion is broken up to the two major problems that its design had to overcome, namely functionality and behavior. Section 5.1 describes the changes necessary to enable the use of the simulated SCTP implementation by external Unix processes. Section 5.2 describes the two approaches we implemented in order to accurately simulate the transport and network of MPI programs. Finally, Section 5.3 concludes the chapter with a summary.

5.1 Enabling External Use of the Simulated Transport

The first step in the creation of MPI-NeTSim was to enable real applications to make use of the simulated SCTP stack and OMNeT++ network. OMNeT++ was originally created to simulate modeled applications, a transport stack, as well as the underlying network but the interactions themselves were self-contained inside the simulator. As mentioned

above, Tüxen et al. [110] added an external interface module to OMNeT++ to allow remote applications to communicate over a real network with applications modeled in the simulator. We extended the external interface design to instead use interprocess communication inside a custom variant of our MPICH2 MPI implementation that used our socket-like API. To the MPI implementation, the simulator is viewed as simply a communication device, and to the simulator, network events are triggered by a set of external channels each attached to an emulated process. The overall architecture of MPI-NeTSim is shown in Figure 5.1. The system consists of two parts: the MPI application and the simulator.

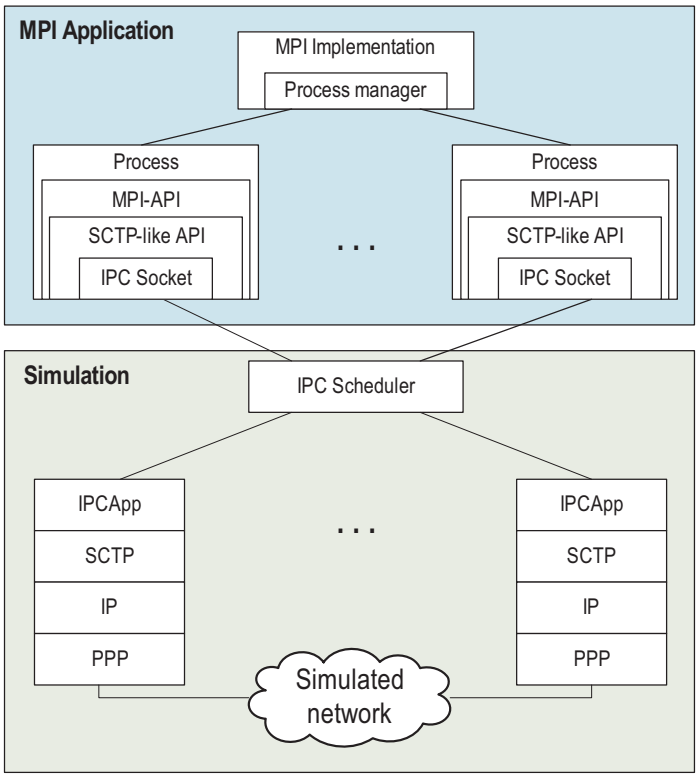

Figure 5.1: Architecture of MPI-NeTSim

In the MPI part of Figure 5.1, the process manager creates and terminates several MPI

processes. The MPI processes communicate using a variant of our MPICH2 implementation that was described in Section 3.2. The original used the SCTP socket API to access the kernel SCTP stack [99]. The module was easy to adapt for use with IPC sockets to communicate with the simulator instead of with the kernel SCTP stack. In order to make use of the simulated SCTP stack, we created an SCTP-like API that interfaced with the OMNeT++ `IPCScheduler`. This is a minimal API that has similar semantics to the SCTP socket API.

One simplification in the design of the SCTP-like API is that we only need to specify our MPI rank in messages, as opposed to IP addresses or association ID. This was done so that all network information about the connection is part of the simulator and not the middleware. All information concerning IP addresses, ports and socket options for a connection are specified in the simulator configuration file.[1] Because the network properties are specified in the configuration file, it is easy to change these properties for a program and to experiment with different networks and vary the transport parameters.

As shown in Figure 5.1, the MPI part communicates with the simulator via IPC sockets, which were chosen because they use the standard Berkeley socket API. The `IPCScheduler` module inside the simulator is the endpoint of the IPC from all the MPI processes. An advantage of using IPC sockets is that the `IPCScheduler` can handle the sockets as normal file descriptors and use `select()`. By using `select()`, the `IPCScheduler` can bind to different kinds of sockets, e.g., IPC and network sockets to handle the call whenever an outside message arrives. This gives the flexibility for allowing our MPI processes locally on the same machine or remotely on another machine; the tool still functions regardless of where the process manager creates the MPI processes.

As shown in Figure 5.1, for every MPI process there is a corresponding `IPCApp` module inside the simulator, which acts as an instance of a host, maintaining its state. As messages

[1] A consequence of this decision is that we do not handle dynamic process creation or dynamic changes to socket options.

from all MPI processes pass through the `IPCScheduler`, they are analyzed and converted into the appropriate network format. In our case, since we are using SCTP, the messages are converted into the simulator's internal representation of an SCTP packet. Control messages, e.g. the closing of a socket, have to be converted into the corresponding SCTP commands. Data messages have to be encapsulated into SCTP `DATA`-chunks, and the source and destination address are adjusted as the messages are routed. Finally, before messages are inserted into the simulation queue, they are time-stamped with the current virtual time, which determines the message's processing time within the simulation. On egress, the `IPCScheduler` collects all messages from the `IPCApps`, finds the corresponding MPI process, transforms the message into the right format and sends it to the middleware of the identified MPI process.

Altogether, messages can now be passed from one MPI process to another MPI process using the simulated SCTP stack and underlying simulated network.

5.2 Solving the Behavior Problem

The previous section described the overall design of the system and the integration of the simulator into the MPI middleware. As a result, all MPI communication is now directed into the simulator which then completes the transport level packet transfer from source to destination. However this does not address how exactly to schedule messages in MPI-NeTSim. In Tüxen's previous external interface work [110], the simulator simply tried to keep up with the external application the best it could. Now that the transport protocol as well as the network are both simulated for distributed MPI applications with more processes, the simulator becomes the bottleneck and the computational delays inside the simulator can alter the communication behavior of the MPI program. In this regard, the notion of time within the simulation is not equal to real-time, and the real-time the simulator is taking is affecting the MPI processes.

As an example, consider Figure 5.2 which shows timelines for two processes A and B. Process A first sends a message to Process B and then Process B sends a message back to Process A. The middle bar is a timeline for the simulator, which receives the message from

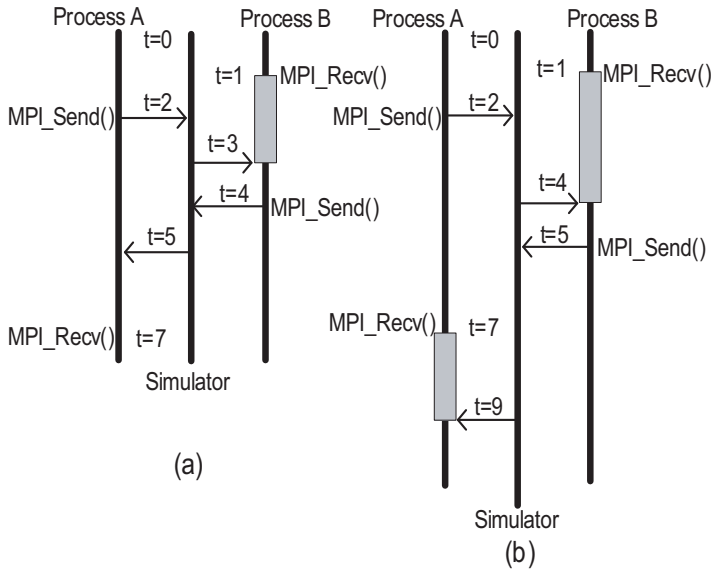

Figure 5.2: The effect of the simulator on program execution time; (a) ideal case where each communication is one unit of real-time, (b) case when the simulator needs more than one time-unit.

the MPI middleware, simulates the transfer of the message, along with all other network traffic, from ingress to egress, and then finally sends the message to the MPI middleware at the destination. We are using blocking MPI communications, and the grey boxes represent process idle time. Figure 5.2(a) shows the idealized execution we are trying to reproduce, where the simulator takes exactly 1 time unit of real-time to complete the transfer of the message. In Figure 5.2(b), when the simulator needs more than one unit of real-time, 2 units at $t = 2$ and 4 units at $t = 5$, then Process A must wait for the message at time $t = 7$. With respect to the MPI middleware, the added delay has changed an unexpected message in

Figure 5.2(a) to an expected/posted message as shown in Figure 5.2(b). We want to ensure consistent behavior where every execution of the program exhibits the same communication behavior and is not dependent on the real-time execution speed of the simulator.

We have implemented two solutions to the clock synchronization problem that introduce additional idle time on the processes to provide a scaled equivalent to Figure 5.2(a). In Penoff et al. [84], a slow-down technique was implemented where a full, system-wide synchronization performs a static time dilation; this is described in Section 5.2.1. Later, we developed a faster, adaptive time dilation technique [94] which utilizes a more local synchronization approach; this is described in Section 5.2.2.

5.2.1 Slow-down Technique

Our first solution implements a *time factor* to uniformly slow down the execution of the MPI processes and the simulator, in order to give the simulator more time to complete its work [84]. Figure 5.3 shows the result of executing the program from Figure 5.2, except now

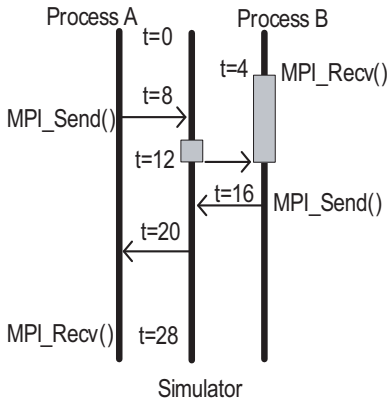

Figure 5.3: The effect of the simulator on program execution time; slow-down solution

processes A and B execute four times slower than before. The introduced idle time makes the MPI program take longer to execute but its execution is now identical in proportion

to Figure 5.2 with respect the message schedule. Prior to $t = 12$, the simulator needs to sleep for two time units to keep to the slower time schedule; for the second communication, the simulation has sufficient time with respect to the slower time schedule. The result is that the behavior of Figure 5.3 is identical to that of Figure 5.2(a) as long as our time factor provides sufficient time for the simulator to keep up with the slower execution of the communicating processes. The resulting network traces from the simulator are time-stamped using simulated time which is identical to the scaled time on the processes.

Given a program, we define *scaled real-time* $SRT_s(t)$ to equal the real-time t divided by the time factor s. Now given two executions of the program with time factors $s_2 \geq s_1 \geq 1$ with executions times t_1 and t_2, then $SRT_{s_1}(t_1) = SRT_{s_2}(t_2)$. In the previous example $s = 4$ and we have that $SRT_4(28) = 7$, the runtime of the original program. To maintain message synchrony we want to also ensure that communication events occur at the same scaled real-time. For all communication events c_i, $SRT_{s_1}(t_1) = SRT_{s_2}(t_2)$, where now t_1 is the real-time c_i occurs in the first execution and t_2 is the real-time the event occurs in the second execution.[2]

In practice, it will almost always be the case that the simulator cannot keep up to the real-time execution of the program and we are unable to measure $SRT_1(t)$ directly. However, for sufficiently large slow-down factor S, for all $s \geq S$, we have $SRT_s(t_1) = SRT_S(t_2)$ where t_1 and t_2 are the executions times of the two runs of the program. As well, in practice, there will be some error in the mechanism and $SRT_s(t_1)$ will not exactly equal $SRT_S(t_2)$. We implemented this slow-down technique throughout MPI-NeTSim and later validated it by showing that for any MPI applications that we executed, some slow-down factor S exists where the runtime of an application will converge and the error with respect to message synchrony becomes bounded [84]. Figure 5.4 shows the convergence of runtimes for a set of applications in the NAS Parallel Benchmark (NPB) suite [69]. The NPBs increase in

[2] Our slow-down mechanism cannot maintain message synchrony for external sources of non-determinism such as the program reading the value of the real-time clock and reacting to it.

size of their data class in this ordering: S, W, A, B, C, D, and E; here, the NPBs from the second data class W are executed with 4 processes on a star topology. The threshold is shown graphically as the leftmost/smallest time factor value at which a given runtime first arrives near to an asymptote; the runtime of each application converges.

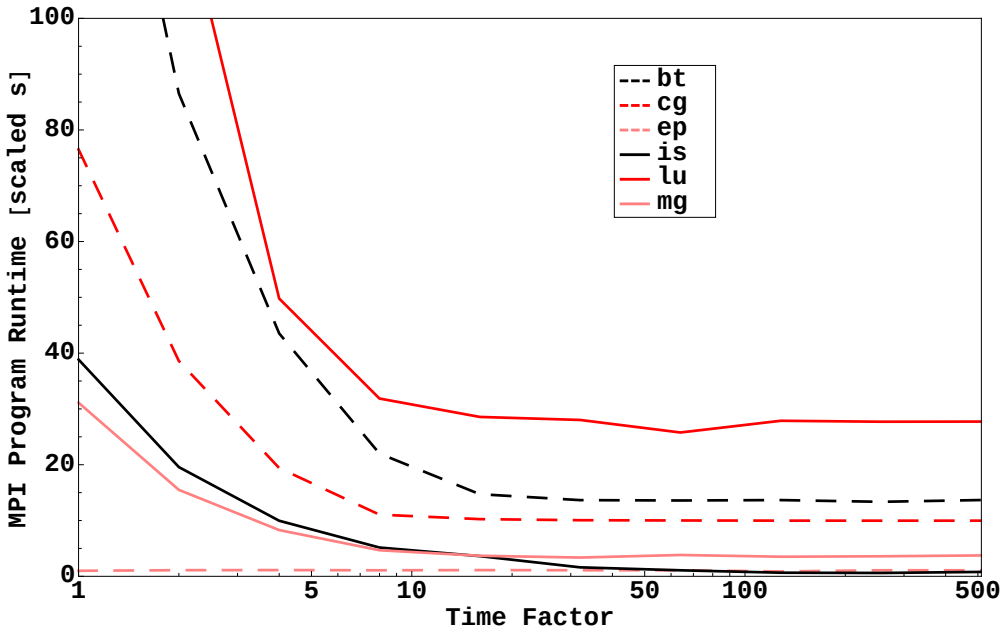

Figure 5.4: Application runtime convergence

The problem with the slow-down technique is that the value of slow-down factor S is determined by the part of the program that has the highest spike in communication and the simulator is the biggest bottleneck. For programs with phases of communication and phases of computation, the slow-down is unnecessary during the computational phases because the simulator is under very little burden then. However, this technique maintains the same time factor throughout the execution of an application. For example, one application that takes a minute on a Gigabit network communicates in high density bursts; in order for the error to converge to its lower bound, a time factor of greater than 500 was required and the

experiment required 8 hours to run. The disadvantage of the slow-down technique is that its methods lack scalability. A more dynamic approach that could adapt depending on the present load was necessary in order to execute larger applications on larger clusters.

5.2.2 Our Adaptable Event Ordering Algorithm

An adaptable event ordering algorithm replaced the slow-down technique in a modified version of the MPI-NeTSim that builds on the previous changes to OMNeT++ and the MPI middleware [94]. The new technique executes several orders of magnitude faster but with the same accuracy; it thereby enables substantially larger scale MPI experiments with more advanced transport and network scenarios.

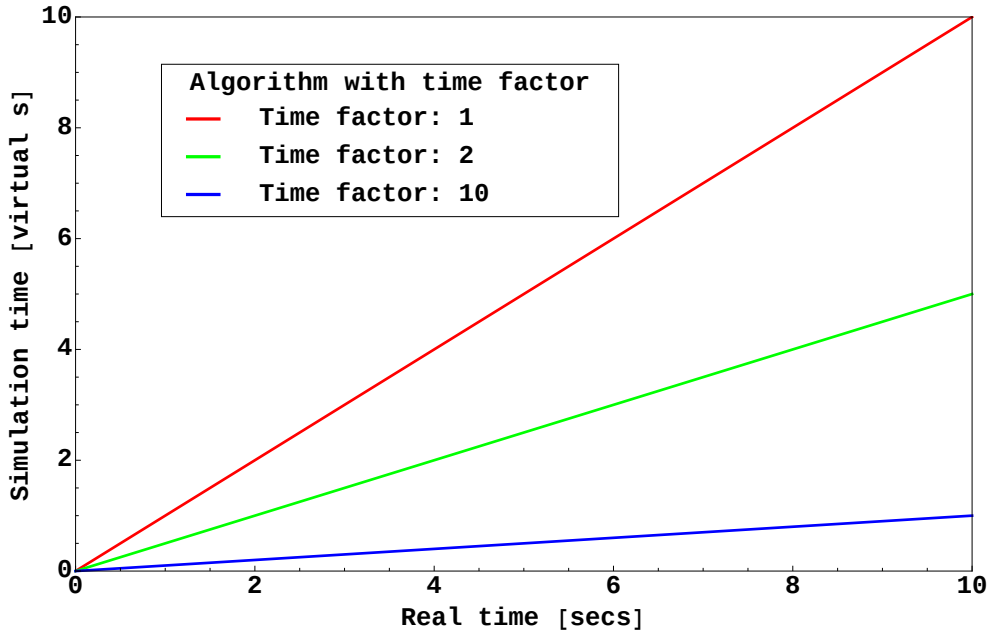

Figure 5.5: Runtime with time factor

The time factor was applied to all events whether they needed it or not as is shown by the

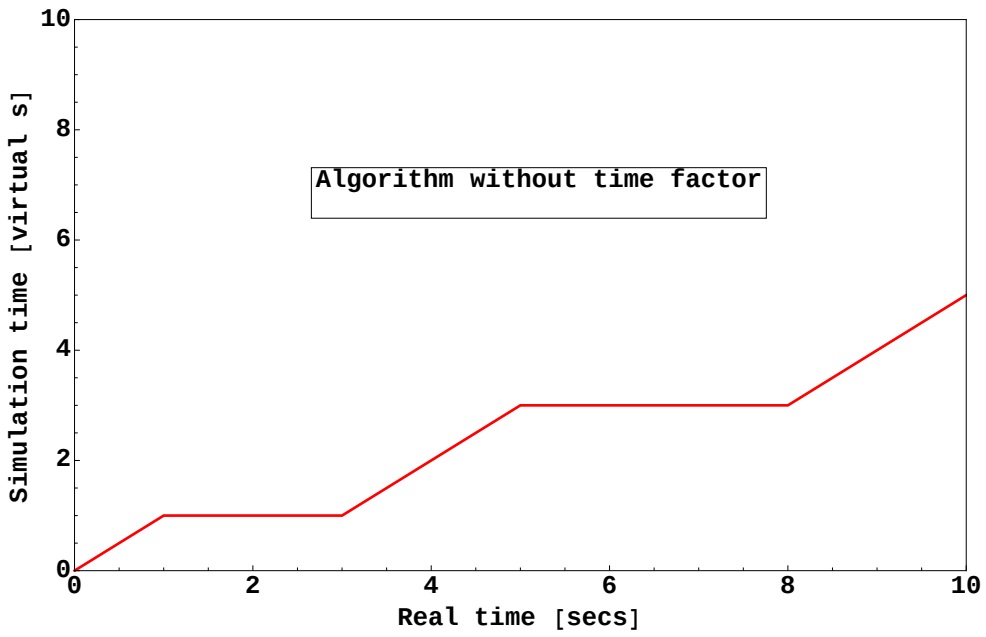

Figure 5.6: Runtime without time factor

constant slopes in Figure 5.5. We wanted our new algorithm to include a time management mechanism that was not universally applied to all events, but adapts according to the real processing times. That meant that when the density of the events (i.e., number of events in a given time unit) was high, the application had to be slowed down - equivalent to a high time factor - whereas at other times a time factor of one could be applied, meaning that applications and the simulation were synchronized. This stepwise linear behavior is shown in Figure 5.6. If the application has to wait, only the real time advances, as is shown by a step, while at the sloped sections both times advance simultaneously. To achieve this event-dependent behavior, the basic `sendToSim()` and `receiveFromSim()` calls for use by the MPI implementation were divided into two parts. A `sendToSim()` call consists of the actual sending of the data and the reception of a notification which mimics the return code. In the `receiveFromSim()` call, the application sends a notification and waits for the data

in a subsequent receive() call. For both calls, the time between the internal system calls is comparable to the 'sleep time' of the former algorithm used to dilate time. By looking at the start and stop times inside the simulator, we calculate a time factor greater than one and enable it by controlling when the notification is sent from the simulator to the process. This locally calculated dynamic time factor should at any given time be less than or equal to the static time factor required in our old algorithm.

The major difference between the previous mechanism and the new mechanism is that the "real-time" clock of our previous system is replaced with a "pseudo-real-time" clock that is updated based on the real-time clocks of processes in the MPI program. When an external process schedules a communication event with the simulator, we update the pseudo-real-time clock of the simulator to the maximum of the real-time clock of the MPI process and the current value of the pseudo-real-time clock. Assuming all MPI communication occurs with the simulator, our local synchronization with the simulator preserves causality since it reduces to a simplified case of Lamport's timestamps [58]. Since the simulator never advances its virtual clock ahead of the pseudo-real-time clock, it is never the case that an MPI process's time is less than that of the pseudo-real-time clock. We assume that all MPI processes used the same clock which is the case for our IPC-based experiments. For larger experiments with MPI processes on different nodes, the clock skew of the different clocks may affect the accuracy of this approach.[3]

Figure 5.7 shows the results comparing the two behavior algorithms for a few different applications, with the problem sizes decreasing from left to right. As Figure 5.7 shows, the new algorithm is significantly faster than our previous algorithm, more notably for large problem sizes, but also for the smallest problem sizes. We have also not included the time required by the previous algorithm to find the minimum time factor needed to converge to the behavior of the application over the course of several executions. Overall, the adaptive

[3]Distributed experiments remain as future work. It should be sufficient to keep the clocks synchronized using NTP (Network Time Protocol).

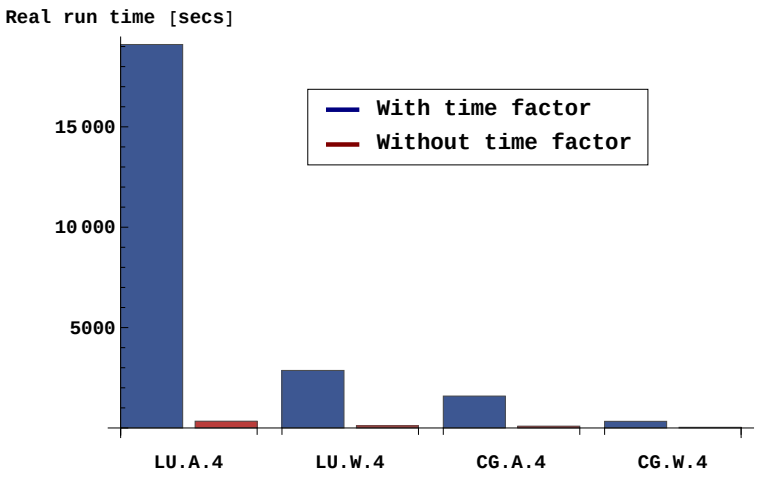

Figure 5.7: Runtime comparison of behavior algorithms

approach showed a significant savings in time.

There have been several related projects that use time dilation techniques together with virtual machines (VMs). Similar to our original slow-down technique in [84], Gupta's DieCast system [42] statically scales back virtual time in order to simulate large scale executions of VMs. In Grau et al. [39], an adaptive approach for VMs modifies the time factor throughout a bursty execution in order to avoid idle time, as is the case with our adaptive event ordering algorithm. The major difference between time dilation approaches and our adaptive algorithm is that our algorithm is simpler because it is not a full system-wide synchronization but instead it leverages local synchronizations with Lamport's time-stamping technique [58]. Another difference is that these VM-based approaches simulate only the network with custom network solutions and the transport executes within the VM, whereas our MPI-NeTSim system simulates the network as well as the transport protocol inside the OMNeT++ simulator [114].

5.3 Summary

Overall, our adaptable algorithm accurately simulates the SCTP transport and network layers by ordering events from real external applications. Without special permissions or needing to physically configure equipment, this tool allows for the easy configuration of the transport and network to execute MPI programs, which in turn enables investigations necessary to support experiments for this thesis.

The major results for this chapter appeared initially in [84] published in the proceedings of the Fifteenth International Conference on Parallel and Distributed Systems (ICPADS'09), with the details of our adaptive algorithm shown in [94] published in the 4th International ICST Conference on Simulation Tools and Techniques (SIMUTools 2011). This work extended the work of Rüngeler et al. [93] and Tüxen et al. [110] to allow for the simulation of network properties of MPI programs. The changes to the OMNeT++ simulator [115] now makes it possible to use OMNeT++ technologies for other similar distributed systems research. These changes are made available as part of the OMNeT++ INET project headed by Varga [113].

Part II

Transport Protocol Enhancements for MPI

Chapter 6

Reliability

Together with its extensions, SCTP enables additional reliability capabilities for MPI implementations. In this chapter, we demonstrate these capabilities using the tools introduced in Part I.

Reliability is very important in MPI. The user does not need to check for transmission errors as complete reliability is the responsibility of the MPI implementation. However, in the event of an error, the MPI Standard defines a set of error codes that can be returned, however the implementation may not be able to recover from these errors and in fact, the Standard has no guarantee that the user can continue to use MPI after an error is reported [62]. In practice in most MPI implementations, the default response when any error is seen anywhere in the complete distributed execution of the MPI application is to abort. Reliability can be added for a message, a network, or an entire node. Node reliability has been investigated in MPI extensions and is complementary to our work. In this thesis, we focus on enhancing message and network reliability in MPI by using SCTP.

SCTP shares many features with TCP for message reliability, however we focus particularly on the features that are present in SCTP but not in TCP. In this chapter, the first covered feature that enables network reliability on Ethernet-based clusters is multihoming; in Section 6.1, we focus on the fault tolerance added by multihoming. In Section 6.2, we focus on how enhancements surrounding SCTP's use of acknowledgments increase message reliability. In Section 6.3 we describe how SCTP's stronger checksum as well as its multistreaming feature also increase message reliability. Section 6.4 concludes this chapter.

6.1 Fault Tolerance and Multihoming

Multiple network cards can be used to increase network reliability. When used by the link layer whose purview is a single network hop, network reliability can be provided for a single hop, for instance, by using channel bonding [88]. Often times though, cluster networks span multiple switched link-layer segments [33], and MPI applications require end-to-end network fault tolerance, not just a single hop. Supporting end-to-end network reliability in the transport protocol through multiple network cards is a new idea in IETF-specified transport protocols [10, 45, 101], and is implemented in SCTP through its multi-homing feature. SCTP has built-in mechanisms for handling multiple paths and network failures, and transparently provides end-to-end network fault tolerance to the application, enhancing the overall reliability. We note that TCP is limited to single-homed connections and resilience to network faults is limited to only detection; when a TCP connection experiences network failure, the connection will eventually fail and report the failure to the application or middleware.

An SCTP sender uses data packets on the primary path and explicit *heartbeat* packets as probes for network connectivity. When the number of packet losses on a path (that result in timeouts and retransmissions) exceeds a user-specified *Path Max Retransmit (PMR)*, the corresponding path is marked as failed, and data transfer *fails over* to an alternate path. The sender continues probing failed paths with heartbeat probes, and switches back to using the original primary path for data transfer if it recovers. Since these probes are at the transport layer, failure anywhere on the end-to-end paths will be detected by SCTP.

In order to show the additional network reliability provided by SCTP multihoming, we conduct two sets of tests. During the execution of these tests, we disconnect network paths in order to demonstrate that instead of aborting as TCP would do, our applications running over SCTP can remain running until completion. In Section 6.1.1, we use the FreeBSD kernel SCTP implementation to execute an SCTP version of the `netperf` bandwidth test

between two multihomed hosts. In Section 6.1.2, we use our MPI-NeTSim tool for executing an application suite across 16 multihomed hosts in order to centralize and therefore simplify the test configuration and gathering of test results in this larger, more complex scenario. Together, these tests demonstrate multihoming's enhanced fault tolerance. The multihoming reliability results are summarized in Section 6.1.3.

6.1.1 Bandwidth Test

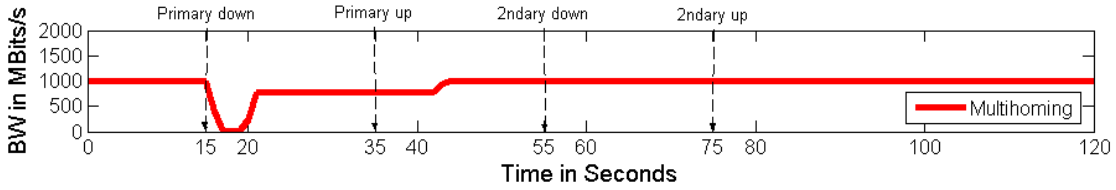

Figure 6.1: Example of failover, two hosts each with two interfaces to create a primary path and secondary path.

Figure 6.1 shows the behavior of SCTP's failover mechanisms for two hosts, each with two Gigabit Ethernet cards connected to separate switches. We execute a bandwidth test and then simulate network failure by using IP firewall rules at the destination to block incoming IP packets to a destination address. We start the transfers, and then (a) at t = 15 seconds, the primary path goes down, (b) at t = 35 seconds, the primary path is back up, (c) at t = 55 seconds, the alternate path goes down, and (d) at t = 75 seconds, the alternate path is back up.

Based on research that recommended values for SCTP failover [16], we used a PMR value of 3, which resulted in a 4.5 second failure detection time.[1] As shown in Figure 6.1, at 15 seconds when the primary path goes down, SCTP takes 4.5 seconds to detect the failure

[1]The SCTP stack was tuned through user controllable `sysctl` settings for cluster environments to have a minimum retransmission timeout (RTO_min) value of 300 milliseconds, leading to a failure detection time of 4.5 seconds.

and to completely switch over to the alternate path. When the path fails, bandwidth does go to zero since all packets on the primary path were lost. Note that once the primary path comes back up, the sender switches back to using the primary path for data transfer, which explains why there is no impact on throughput when the alternate path goes down between 55 and 75 seconds.

In the case of the Concurrent Multipath Transfer (CMT) extension for SCTP [48], the mechanism for determining reachability is different. With CMT, data is distributed over all paths, and not sent only on the primary path. For CMT a congestion window (`cwnd`) for each network path is maintained and data segments are distributed across multiple paths using the corresponding `cwnd`s, which control and reflect the currently allowed sending rate on their corresponding paths. CMT thus distributes load according to the network paths' bandwidths; a network path with higher available bandwidth carries more load. We conducted the same experiment with CMT to test its fault tolerance capabilities.

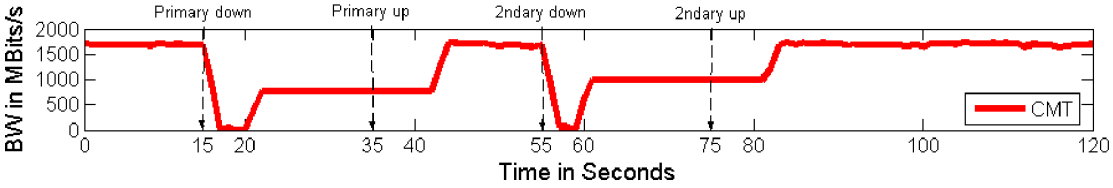

Figure 6.2: Example of failover with CMT, two hosts each with two interfaces where both paths are used for data transfer

As shown in Figure 6.2, CMT shows very similar behavior when the primary path fails. The recovery time is approximately the same as with SCTP, but in the case of CMT, throughput is higher when both paths are up. In the case of core SCTP, the secondary path failure has no effect, but in the case of CMT, recovery of the secondary is similar to that of the primary. This is because all paths are treated alike in CMT; there is no notion of a special "primary path" since data is distributed over *all* paths. When any path

goes down, CMT experiences *receive buffer blocking* [49] that causes a transient dip in its throughput, but on detection of a failed path, CMT recovers to its full potential.

Figure 6.3: Example of failover with CMT-PF, two hosts each with two interfaces with tuning of "possibly-failed" value.

There is the potential for further improvement in network fault tolerance with the use of CMT. Since CMT spreads data over all paths to the receiver, a CMT sender has more information about *all* network paths, and can leverage this information to detect failure sooner. *CMT-PF* introduces a *Possibly-Failed* state at the sender, which reduces failure detection time to one timeout period, which is at least one second, as per current defaults [70, 71].[2] This improvement is significant, as can be seen in Figure 6.3, which shows our network failure experiment repeated with CMT-PF. We can see that CMT, now with the Possibly-Failed state, provides a continuous and smooth transient behavior during failure, transparent to the middleware and application. The throughput never drops to zero as it did in Figures 6.1 and 6.2. CMT-PF is now a part of CMT's default behavior [70].

Fault tolerance is the main motivation for multi-homing as well as for projects such as RI2N/UDP[74]. When using TCP for the transport, network failure due to switches or software (mis)configuration requires the middleware/application to create a new transport connection in order to possibly recover from data lost in the failed connection. Such mechanisms have been implemented in MPI middleware [116], but are unnecessary for SCTP

[2]For cluster environments, failure detection time of one link can be made as low as a few round trips, i.e., to the order of milliseconds. For CMT-PF, failure detection of one link failure does not affect other healthy links.

and CMT based middleware, where multi-homing is built-in and automatic failover and recovery are part of the transport protocol, occurring transparently to the user-level software. However, MPI programs cannot recover from network failure when using standard TCP implementations of MPI.

6.1.2 Real Applications

Unlike the bandwidth test between two hosts that was shown in Section 6.1.1, parallel applications can execute on a greater number of hosts and they tend to have more bursty communication patterns. In order to test the fault tolerance of multihoming at a larger scale on real applications, we tested the fault tolerance of multihoming on the NAS Parallel Benchmarks [69]. These applications demonstrate realistic communication patterns at a larger scale than the bandwidth tests done in the previous section. MPI-NeTSim is used to conduct larger tests of real applications across 16 simulated hosts on simulated networks. These tests could have been conducted on real machines using a kernel stack or our userspace stack, however by conducting them with MPI-NeTSim, the settings as well as the results are all centrally managed by the simulator, easing the testing process but maintaining testing accuracy.

We investigated multi-homing for a network where every host has two interfaces and each interface is connected to two separate subnets via a router or switch. As a result there are paths between every pair of hosts in the system. In our network, we have set the routing tables on hosts so that all processes use the same subnet as the primary path. We reserve the second subset for failover in case of link errors or link failure.[3] The MPI processes each act as a separate host machine and on initiation, bind to both interfaces so that inside the SCTP transport protocol there are two paths between every pair of MPI processes. Because multi-homing is performed by the transport protocol, link failures can occur completely

[3]The SCTP implementation inside OMNeT++ did not have the CMT extension implemented at the time of this writing.

transparently to MPI and thus enhance the fault tolerance of the MPI program. It is important to note that using a TCP-based MPI without fault tolerance implemented at the user level[4] would result in complete failure under either of these conditions, but SCTP provides additional reliability at the transport level, which is the lowest end-to-end layer; MPI-NeTSim allows us to test its effectiveness. We describe the details of the parameters in SCTP and using MPI-NeTSim, experimentally investigate the effect of one or more link failures on the execution of an MPI program.

Failover in SCTP occurs according to a mechanism made of several determinants. For each Ethernet frame sent by SCTP a timer is started, which is set to expire after a retransmission timeout of RTO seconds. The RTO timeout value is based on the current round trip time measurement, with a minimum value of RTO_{min} and a maximum value of RTO_{max}. If the timer expires before an acknowledgment for the data is received, then the data is retransmitted on the alternate path and the RTO is doubled. Consecutive timeouts continue to double the RTO until $RTX_{max}^{(P)}$ rounds for path P after which SCTP fails-over to a secondary path. When the links are idle, SCTP uses a heartbeat mechanism to detect link failure. There is a user-configurable parameter, $HB_{Interval}$, that controls the heartbeat interval. Heartbeats are sent out every $HB_{Interval} + RTO$ seconds and need to be acknowledged to verify path integrity.

Formula 6.1 calculates $T_{failure}^{(P)}$, an upper bound on the maximum amount of time it takes to failover from the primary path to a secondary one.

$$\begin{aligned} T_{failure}^{(P)} &= 0.5 \cdot HB_{Interval} \\ &+ \sum_{i=0}^{RTX_{max}^{(P)}} (RTO_{max} + HB_{Interval}) \end{aligned} \tag{6.1}$$

To investigate the impact of path failures in an MPI program, we created the network

[4]This is a characteristic of most TCP-based MPI implementations.

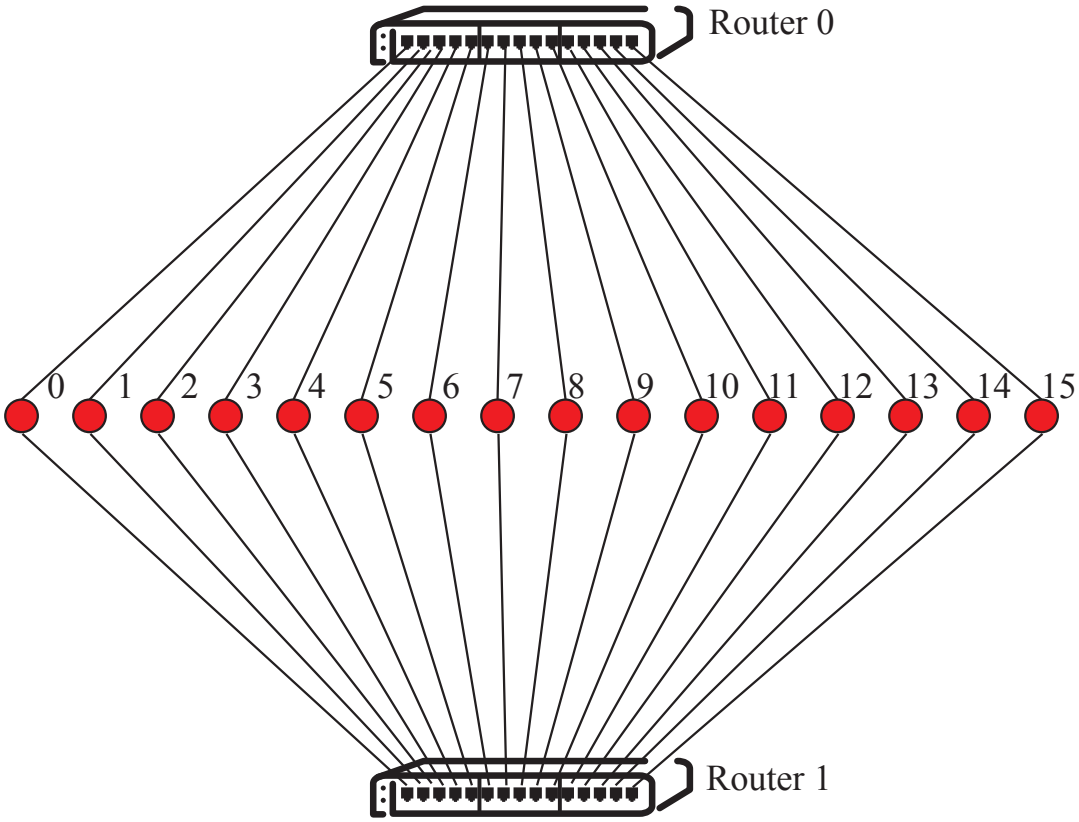

Figure 6.4: Topology of a dualhomed star

shown in Figure 6.4.[5] All links are configured with a bandwidth of 1 Gbps and a latency of $1\mu s$. Table 6.1 compares the default SCTP settings of parameters affecting failover time typically used for wide-area networks (WANs) with long round trip times, versus the settings we used, which were optimized for MPI in the network shown in Figure 6.4. There is a substantial difference in these settings that will affect failover time, according to Equation 6.1. As heartbeats are only sent if the heartbeat interval timer expires and no acknowledgments have been sent for the duration of $HB_{Interval}/2$, Equation 6.1 is the upper bound for the time between the failure of the path, its detection and failover to a

[5]The routers can also be substituted by switches.

Parameters	**Default**(WAN)	**MPI** (LAN)
RTO_{Min}	1 sec	20 msecs
RTO_{Max}	60 secs	100 msecs
$HB_{Interval}$	30 secs	50 msecs
$RTX_{Max}^{(P)}$	5	2

Table 6.1: User configurable parameters for failover, default versus experimental settings for MPI in Figure 6.4.

second path. For our configuration, this upper bound T_{Max} would result to

$$T_{Max} = 0.5 \cdot 50ms + 3 \cdot (50ms + 100ms) = 475ms \tag{6.2}$$

On busy paths the failure can be detected earlier, because the heartbeats are not needed to probe the link. Therefore,

$$T_{Min} = 20ms + 40ms + 80ms = 140ms \tag{6.3}$$

For our tests, we used the 16 process versions of the five NAS benchmark tests BT, CG, IS, LU, and MG with data size B. We tested two failure scenarios. First, we failed the link going from Host 0 to Router 0 in Figure 6.4; for the second case, all links leading to Router 0 were failed, simulating a complete router failure. Link failure was simulated by setting the probability of packet loss on the link to one.

The recovery times, i.e., the times between the link failure and the last connection switching to the new path, are shown in Table 6.2. The times vary according to communication patterns and thus, the number of busy and idle links involved. IS has the highest number of connections (120), which means that with a router failure 240 path failures have to be detected. In other benchmark tests, for instance CG, some nodes are not connected at all.

	BT	CG	IS	LU	MG
Host 0	144 ms	205 ms	292 ms	340 ms	300 ms
Router 0	336 ms	365 ms	397 ms	396 ms	365 ms

Table 6.2: Recovery times for various NAS benchmarks

From one application to the next, the total runtimes do not necessarily increase with these network failures by the same amount. For MG, for example, failing only the single path takes 300 ms to recover however the total runtime was only 296 ms longer than a run without any link failures. For the same MG test, when the router outage occurs, it takes 365 ms to recover but in that case, the application itself is only 335 ms longer than a run of MG without any link failures. In general, the NAS MPI applications are able to recover from various path and switch failures in a reasonable timespan.

6.1.3 Multihoming Summary

We have shown that multihoming can help add reliability to MPI programs. The failover mechanism was described and demonstrated by a bandwidth test between two hosts as well as with real applications across 16 nodes. By using SCTP-based MPI middleware, MPI applications withstand link and router failures that would cause the application to abort with the typical design of TCP-based MPI middleware.

6.2 Acknowledgment Enhancements

In order to implement basic reliability, an acknowledgment scheme is a part of a protocol that informs the sender of the state of the receiver so that the sender can determine if data needs to be resent. In this section, some topics based on SCTP and acknowledgments are addressed. First in Section 6.2.1, we briefly describe how SCTP's acknowledgment scheme provides message reliability more efficiency. Finally in Section 6.2.2, concerns over the

computational costs of SCTP's acknowledgment scheme are addressed.

6.2.1 Increased Efficiency

SCTP implements an efficient acknowledgment scheme that allows the receiver to describe the exact gaps of out-of-order data to the sender. Data can arrive out-of-order because of loss, intermediate route changes, or if data is arriving on multiple paths. When acknowledging data, TCP uses a cumulative acknowledgment scheme by default that informs the sender of the lowest sequence number of the data it has not yet received. Data that has arrived to the receiver that has a higher sequence number than the lowest sequence number not yet received is therefore not reported to the sender. The Selective Acknowledgment (SACK) extension [60] to TCP increases the efficiency of the acknowledgments because the exact gaps can be described. The number of gaps that can be expressed are limited to the space in the TCP Options part of the TCP header. SCTP uses SACK as its default acknowledgment scheme and it has more space to express a larger number of gaps, therefore it can implement reliability with out-of-order data arrival more efficiently because the exact state of the receiver can be more easily expressed with fewer acknowledgments. The exact state can result in more precise retransmissions as the exact gaps can be resent instead of everything after the lowest received sequence number, as is the case with TCP cumulative acknowledgments.

6.2.2 Minimal Computational Costs

Kant and Jani [56] discuss several concerns about the computation costs of SACK for SCTP, specifically in the Linux kernel implementation. The SCTP standard says that by default, one SACK is sent for every other packet [100]. The Linux kernel implementation [59] sends a SACK when the receiver's window changes, so for the tests conducted by Kant and Jani [56] in Linux, it was observed that a SACK was sent per data packet as result of these rules.

They misinterpreted this behavior to be true of the SCTP standard itself and not merely the Linux kernel stack where their tests were conducted.

Another misunderstanding in Kant and Jani [56] was that the ability to efficiently express an arbitrarily large number of gaps by SACK in SCTP was irrelevant because a SACK is sent for every other packet, so there can only be at most 2 gaps. This is simply not true. The number of gaps can grow if for example, every other packet was lost. Another cause of gaps can be out-of-order arrival happening when packets are transferred on different paths due to multihoming or intermediate route changes.

Our userspace SCTP stack, which is derived from the FreeBSD implementation, does not have the implementation-specific SACK performance bugs that the Linux SCTP stack does. The FreeBSD stack is much more optimized in general, and this is again true in the case of SACK.

In our userspace stack, SACK is also sent less frequently. As the standard requires by default, a SACK is sent for every other packet, however the frequency can be lessened by way of a `sysctl` setting. The other source of SACKs in the Linux stack as seen in Kant and Jani [56] was that a SACK was sent for each change in the receiver window which is more than an MTU. Our userspace stack sends a SACK for each change in the receiver window which is more than 25% of the maximum receiver window. When a SACK is processed, typically the standard case is there is no loss; here, the computational costs are the same for SACK as for cumulative acknowledgments because an express path of highly optimized code is executed. However, when there is loss, the ability to express more gaps in the transmission results in a more efficient communication of state and retransmissions.

6.3 Stronger Checksum and Multistreaming

SCTP has two additional features that increase message reliability: the CRC32c checksum and multistreaming.

6.3.1 Stronger Checksum

SCTP uses CRC32c as its checksum algorithm [102]. CRC32c used by the iSCSI storage standard as well. TCP uses an additive 16-bit checksum which has been shown to accept 1 in 1×10^7 packets as valid despite being corrupted [103]. The CRC32c strengthens the message reliability by providing more protection by way of a stronger algorithm. The CRC32c has been proposed to add to TCP as well, but has not yet been accepted at the time of this thesis [13].

There have been concerns about the computational costs of CRC32c [56]. The CRC32c offload is becoming more common on NICs, e.g., those using Intel's 82599 Niantic controllers. When this is available, then the host is not burdened by the extra cost of the CRC32c algorithm. However, when the host does have to bear the burden of calculating the stronger checksum, the additional reliability can cost extra compute cycles. For example, when conducting a bandwidth test using our userspace SCTP stack, disabling the CRC32c calculation resulted in a 1-3% higher bandwidth, depending on the message size. The computational cost by the host may be justified for the additional message reliability, but checksum offload is becoming more common so the benefit of the algorithm will be available without the computational cost by the host CPU.

6.3.2 Multistreaming

When applications are written in a latency tolerant manner, SCTP multistreaming can also enhance message reliability. Our SCTP-based middleware maps different MPI tags to SCTP streams. An MPI program can be latency tolerant if it can adapt to an incoming message regardless of its source or tag. This can be done by pre-posting several MPI requests using non-blocking communication and calling `MPI_Wait()` over all of them, or through the use of wildcards.

In Kamal, Penoff, and Wagner [54], we created a bulk processor farm in order to study

the benefits of multistreaming to avoid head-of-line blocking. This MPI application had a manager and any number of workers. The workers request work from the manager and the manager assigns 10 tasks for the worker to do, each with a different tag. The application is tolerant such that it executes whichever task comes first. The same application was executed using our SCTP-based MPI middleware which maps tags to streams as well as a variation of this where the mapping is always to the same stream, as would be the case in TCP since it is fully ordered.

To focus on the potential benefits of multistreaming for MPI, the variants of the bulk processor farm were executed using a single network path under normal conditions and then using loss rates of 1% and 2% using dummynet. Under this test, loss leads to out-of-order arrival however out-of-order arrival could occur under multipath scenarios as well; multistreaming can alleviate any out-of-order arrival scenarios. Using a single path, when there was no loss then the average execution times of the farm were equal, however, for the loss scenarios, the reduction in average run-times when using multiple streams compared to a single stream is about 25%. This shows that head-of-line blocking can have a substantial effect on a latency tolerant program like the bulk processor farm, and that SCTP multistreaming together with SACK can help alleviate the problems caused by head-of-line blocking and thereby increase message reliability for MPI programs.

6.4 Reliability Summary

In this Chapter, we showed the additional network reliability added by the multihoming feature of SCTP to a simple bandwidth test as well as to the larger, more complex NAS parallel benchmark suite of various MPI applications. Network failures in TCP-based MPI implementations would fail[6] completely whereas our tests completed and with minimal

[6]Standardization efforts are underway in the IETF attempting to enable multipath capabilities in TCP [45].

delay.

The message reliability that SCTP adds was also demonstrated. Selective acknowledgments provides an efficient way to express gaps caused by out-of-order arrival. The CRC32c checksum in SCTP provides a higher level of data integrity compared to TCP's 16-bit checksum. Our SCTP-based MPI's mapping of MPI tags to SCTP streams within SCTP's multistreaming feature together with SACK decreases the effects of head-of-line blocking in MPI applications, also increasing the message reliability.

Overall, SCTP increases the network and message reliability for MPI implementations.

Chapter 7

Performance

Applications are executed in parallel in order to achieve better performance. For MPI implementations executing in Ethernet-based parallel clusters, we show that SCTP-based MPI middleware matches the performance of TCP and when using multiple networks simultaneously, surpasses TCP. This is demonstrated using the tools introduced in Part I.

In this chapter, we first look in Section 7.1 at the use of the Concurrent Multipath Transfer (CMT) extension to SCTP, comparing it to data striping implemented in the application layer; we show how it can enhance the performance of an MPI implementation. After this, we take an extensive look at the performance of our userspace SCTP implementation in Section 7.2. Next, in Section 7.3 we show some optimizations for the NAS MPI parallel benchmarks using both the userspace stack as well as MPI-NeTSim. Finally, we conclude in Section 7.4.

7.1 Data Striping: Transport versus Application Layer

The original SCTP RFC 2960 supports multihoming for the purposes of fault tolerance, however CMT extends SCTP's multihoming in order to provide the striping of data across all available paths simultaneously at the transport level. CMT was originally tested only in simulation [48] however, in our work [80, 81], we have improved an initial kernel implementation so that it can execute real MPI applications over real 1 Gbps networks on FreeBSD; these results are shared in this section where we show how CMT enables additional per-

formance for MPI compared to data striping performed in the MPI middleware by Open MPI.

Experiments were performed to evaluate the potential performance benefits of CMT and compare our approach to the alternative approach of message striping in the middleware. In Section 7.1.1, we describe in detail the test setup used in our evaluations of SCTP and CMT. In Section 7.1.2 we measure the bandwidth of TCP and SCTP with and without CMT using `iperf` to benchmark the performance of the protocols outside of MPI. Finally, in Section 7.1.3, we use an MPI microbenchmark to evaluate the potential of bandwidth of CMT for MPI programs and compare it to Open MPI message striping. Generally in this section, latency tests are not reported because their results showed no appreciable differences between the two approaches in contrast to our bandwidth results. As a result, the focus of our performance evaluation throughout this section is on bandwidth.

7.1.1 Test Setup

We performed our evaluations on a small cluster of 3.2 GHz Pentium-4 processors that are part of an IBM eServer x306 cluster. Each compute node has three Gigabit Ethernet interfaces. There are two private interfaces on separate VLANs connected to a common Baystack 5510 48T switch, and the third (public) interface was connected to a separate yet identical Baystack switch. The two private interfaces are both on-board NICs, however, as will be discussed in Section 7.1.2, one port displays much better performance. Generally both on-board NICs were faster than the public interface which was attached to the PCI bus, so these private interfaces were used in our testing. All SCTP tests were based on the SCTP stack for FreeBSD.[1] CMT was enabled or disabled within the SCTP stack using the `net.inet.sctp.cmt_on_off sysctl` setting.

Configuring our SCTP-based MPI implementations to specify which interfaces to use is

[1] A patched version of the SCTP stack for FreeBSD 6.2 (`http://www.sctp.org`) was used. This same SCTP stack has since become part of the FreeBSD 7 release and beyond.

easy. In Open MPI, command-line arguments are used to specify the interfaces and the data striping technique. In MPICH2 `ch3:sctp`, the interfaces can be specified by setting the environment variable `MPICH_SCTP_MULTIHOME_FILE` to point to a configuration file listing the networks to use. Single path runs occur in MPICH2 when this environment variable is unset. The long message rendezvous protocol threshold for MPI was set to 64 KBytes on MPICH2, matching `ch3:sctp` to Open MPI's default.

7.1.2 Raw Bandwidth and CPU Utilization Measurements

In this section, we investigate the raw bandwidth achievable on our test network and the CPU costs associated with the transport protocol processing.

In order to determine our network's baseline bandwidth values, independent of MPI, we selected an SCTP-capable version of `iperf`, a standard tool used for bandwidth measurements. There were some modifications required to the `iperf` code before it could be used. Firstly, in order to use CMT, we needed control over which interfaces to bind to, and therefore modified `iperf` to selectively bind to the interfaces listed in a configuration file. Secondly, inspection of `iperf` code revealed that it sends messages of size equal to the maximum segment size (MSS) of the path. We noticed that this resulted in a large number of system calls being made and excessive consumption of CPU cycles. We modified the `iperf` code to send messages of 64 KBytes instead, to reduce the number of system calls and increase performance. This modification also allows for a direct comparison with MPI bandwidth results discussed later, since we employ the same technique for sending large messages, by breaking it up into fragments of 64 KBytes, in our SCTP-based MPI implementations.

We experimented with maximum transmission units (MTU) sizes of 1500 bytes and 9000 bytes (jumbo frames). The use of jumbo frames is important because it significantly reduces the number of interrupts caused by packet arrivals at the end systems. Interrupt

coalescing, along with other optimizations, are available on high performance network cards, but our intent was to examine the benefit for the lower cost Gigabit cards. Jumbo frames are supported on most cards as well as most inexpensive switches, although compatibility can sometimes be an issue and result in additional latency.

Single Link Bandwidth Tests

As mentioned in Section 7.1.1, our test hardware has two on-board private network interface cards, where one network interface was serviced more frequently than the other. Prior to enabling CMT, we conducted `iperf` runs to measure the maximum bandwidth achievable on a single link between hosts. The same numbers were obtained for both SCTP and TCP.

For an MTU of 1500 bytes we obtained 934 Mbps (116.75 MBps) on one network and 577 Mbps (72.125 MBps) on the other. With MTU of 9000 bytes, these bandwidth numbers increased slightly to 990 Mbps (123.75 MBps) and 775 Mbps (96.875 MBps) respectively. These were assumed to be the maximum of our setup since `iostat` showed idle CPU cycles when these numbers were obtained, indicating that the `iperf` runs were I/O bound and not CPU bound. The increased bandwidth obtained by using a larger MTU size was due to increased packet efficiency.

Two Link Bandwidth Tests

In order to evaluate the effectiveness of CMT at taking advantage of more than one network connection, we conducted experiments with two paths between the test nodes. Ideally, with CMT turned on, we expected to see the sum of the single link numbers quoted above, i.e., 1511 Mbps (188.875 MBps) with MTU 1500 bytes and 1765 Mbps (220.625 MBps) with MTU 9000 bytes. Our bandwidth tests achieved 1070 Mbps (133.75 MBps) with MTU 1500 bytes and 1680 Mbps (210 MBps) for MTU 9000 bytes.

For any transport protocol, there are trade-offs between bandwidth and the associated

CPU overhead due to protocol processing, interrupts, and system calls.[2] CPU overhead is important since it reduces the CPU cycles available to the application. We used `iostat` together with `iperf` to investigate the CPU overhead of CMT processing and the resultant penalty paid in terms of bandwidth utilization.

CPU utilization for TCP and SCTP with an MTU of 1500 bytes was similar. However, with CMT, the CPU was a limiting factor in our ability to fully utilize the bandwidth of both Gigabit interfaces. For both the MTU sizes, the CPU was the bottleneck (0% idle) on the sender side. For an MTU of 9000 bytes, the CPU was still saturated, yet CMT was able to achieve almost 95% of the bandwidth of the two links, because more data could be processed with each interrupt.

In summary, CMT is able to aggregate bandwidth, a process that can be computationally expensive.

7.1.3 Microbenchmark Performance

In this section, we use MPI microbenchmarks to compare multi-railing in the transport layer (through CMT) with that in the middleware (through Open MPI message striping). For our evaluation, we used the MPI microbenchmarks released by the MVAPICH research group at the Ohio State University [73].

Prior to comparing the two multi-railing approaches, an important test was to investigate any variation introduced in the results due to the use of two SCTP Open MPI implementations (one-to-one versus one-to-many). We experimented with a single link between hosts and kept all other parameters constant, except the type of BTL module used. Figure 7.1 shows that the difference between the SCTP implementations is minimal. This test validates the experiments discussed below and properly attributes the performance

[2] As was mentioned in Section4.1, a saturated CPU using a bandwidth microbenchmark was shown by Hausauer [43] for a kernel-based TCP stack to use 40% CPU for transport protocol processing, 20% for intermediate buffer copies, and 40% for application context switching.

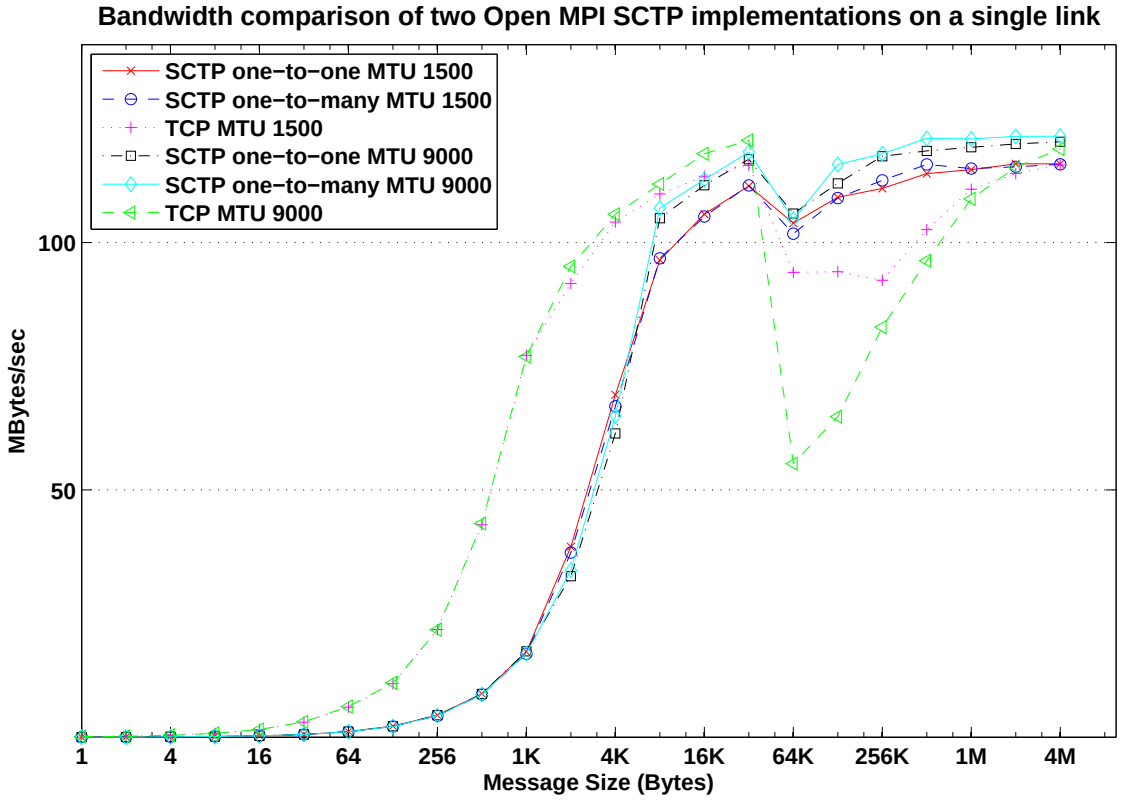

Figure 7.1: OSU Bandwidth Test comparing the two Open MPI SCTP implementations using a single link between hosts. TCP performance is also shown.

differences to the multi-railing approach used, rather than the SCTP socket style in the implementations. For reference, TCP performance is also shown. For the largest message size of 4 MB, all implementations converge to show nearly equivalent maximal performance on a single link.[3] TCP yields higher bandwidth for short messages than SCTP and the difference can be attributed to the fact that SCTP is message-based. No message aggre-

[3]Figures 7.1 and 7.4 show a large dip for TCP at the short to long message boundary (64KBytes). This large difference in the dip between SCTP and TCP only shows up on FreeBSD systems, but not on Linux. This anomalous behavior of TCP BTL on FreeBSD has been reported to the Open MPI developers.

gation occurs inside our SCTP-based MPI middleware and for each `sendmsg()` call, each message also gets an SCTP chunk header in addition to the overall packet header; this is one source of additional overhead in SCTP that is relatively larger for smaller message sizes. In contrast, TCP has only the packet header and no per-message header since it is not message-based.

Another test was conducted to determine whether MPI, through CMT, is able to exploit the available bandwidth of multiple interfaces. Figure 7.2 shows the performance of the system using our Open MPI SCTP one-to-many implementation with CMT turned on or off, for MTU 1500 and 9000. At 1500 MTU, the CPU is saturated and using CMT does not add any benefit with regards to bandwidth. As mentioned in Section 7.1.2, the maximum bandwidth achievable on our test network is about 220 MBytes per second with an MTU size of 9000 bytes. As Figure 7.2 shows, CMT with jumbo frames achieves 90.8% of the bandwidth of two Gigabit links for large messages. This overhead can be attributed to the cost of the MPI middleware since the base was achieved using only `iperf`. It clearly shows the ability of CMT to exploit the bandwidth available on the two paths in MPI.

We only begin to obtain an advantage with CMT after messages of size 8 KBytes. It is well-known that for small message sizes, it is not possible to fill the pipe and hence utilization for short messages, as shown in Figure 7.2 is low. CMT does not get an opportunity to saturate both links since demand is lower than the capacity of a single link. The dip of all the values at the 64 KBytes message size occurs because it is the point at which MPI switches from the eager short message protocol to the rendezvous long message protocol.

The third test was to compare CMT's multi-railing at the transport protocol layer, with Open MPI's multi-BTL multi-railing at the middleware layer. Testing under multi-homed conditions was done in two ways. For the one-to-one version, we ran the OSU tests with two SCTP BTLs, identical to how a dual BTL run over TCP works. For the one-to-many version, we used a single SCTP BTL but enabled CMT in the FreeBSD kernel and used `sctp_bindx()` to bind the second interface to our BTL.

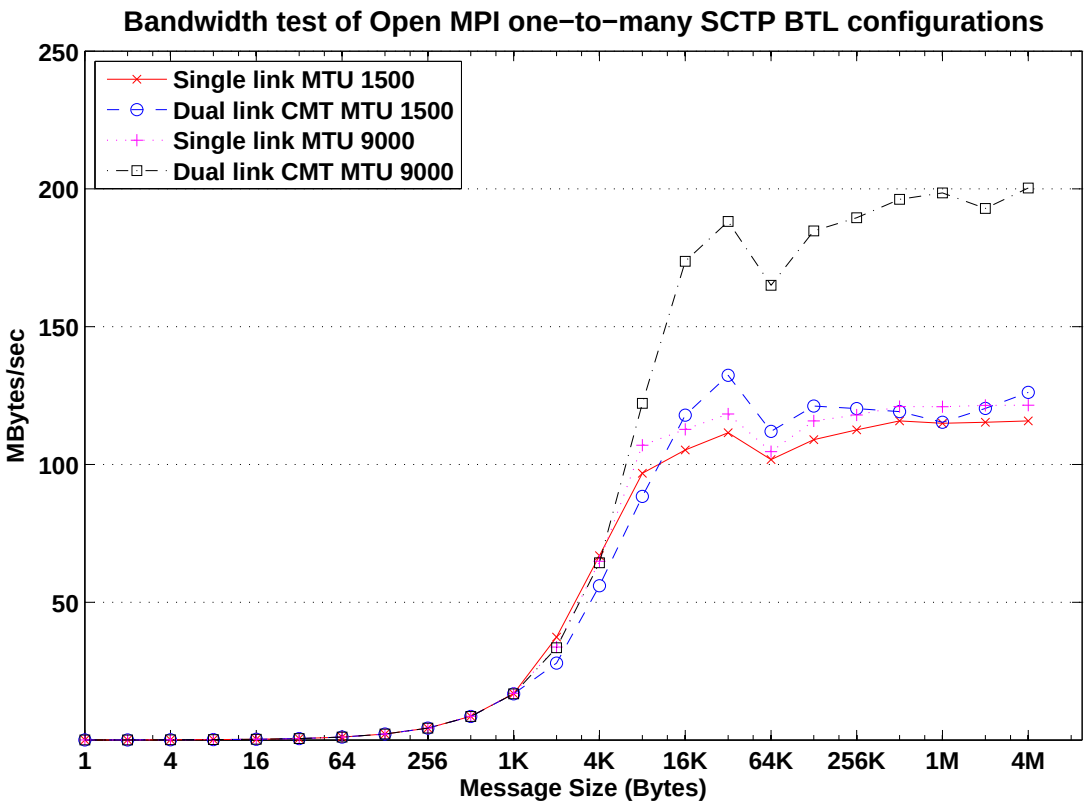

Figure 7.2: OSU Bandwidth Test comparing SCTP-based one-many Open MPI implementation with CMT on versus CMT off, for MTU 1500 and 9000.

In the two BTL case (both TCP and one-to-one SCTP), each socket has its own buffer and hence double the amount of buffer space compared to a single one-to-many BTL with two links, where one socket buffer is shared between both links. To ensure the same amount of socket buffer space for both scenarios, we doubled the size of the single buffer in the one-to-many case to match the two socket buffers used in the two BTL case.

Figure 7.3 shows the performance of the system using Open MPI with two BTLs versus using CMT. For jumbo frames, CMT can achieve 90.8% of the two Gigabit links versus

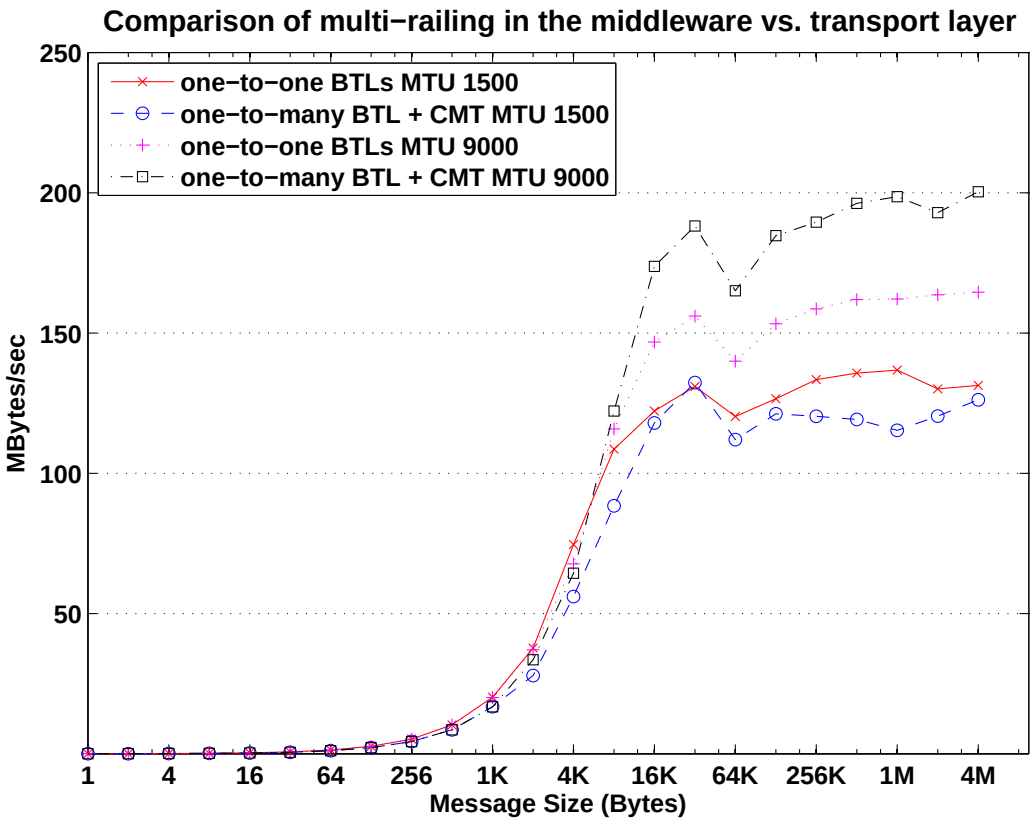

Figure 7.3: OSU Bandwidth Test comparing SCTP-based one-to-one Open MPI implementation with two BTLs versus one-to-many with CMT.

only around 74.5% using two BTLs. This shows a clear performance advantage of multi-railing at the transport layer versus middleware layer. CMT can dynamically schedule data chunks into the first available congestion window, thereby better utilizing the bandwidth. Open MPI, on the other hand, assigns messages to whichever socket buffer has space, without regard to which BTL below has available congestion window space, and may thus not be able to saturate the congestion windows of the BTLs. As the figure shows, there is no significant advantage to either approach until messages are larger than 8 KBytes.

Interestingly, middleware-based message striping does better for a 1500 MTU size. As previously explained, there is additional CPU protocol processing overhead with CMT that affects the bandwidth when the CPU becomes the bottleneck.

In the final set of tests with Open MPI, we compare the different SCTP implementations with the TCP implementation in Open MPI. The TCP implementation does multi-railing in the middleware and provides a reference point against which our SCTP implementations can be compared.

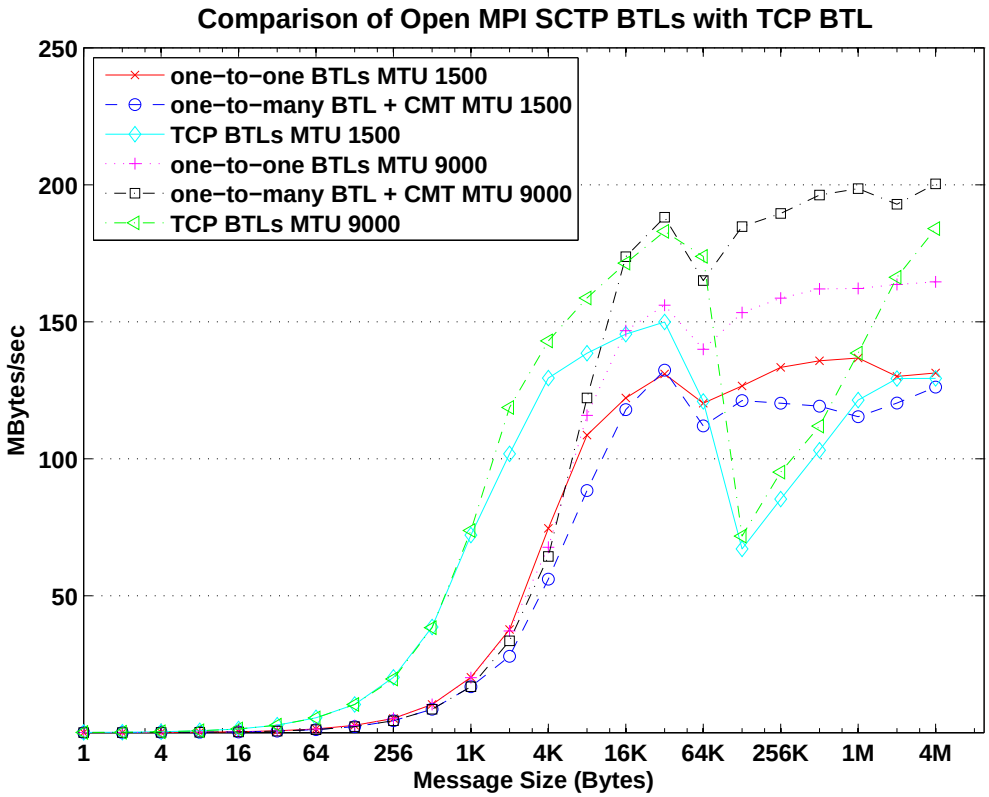

Figure 7.4: OSU Bandwidth Test comparing SCTP implementations with the TCP implementation using two links between the hosts.

As shown in Figure 7.4, the SCTP one-to-many BTL outperforms TCP in the MTU 9000 case and is comparable to TCP in the MTU 1500 case, especially for large messages. TCP performs better in its two BTL run than our one-to-one two BTL case. There is an interesting difference between SCTP and TCP at the location of the dip caused by the switch from the short to long message protocol at the 64 KBytes boundary. Both TCP and SCTP recover from the decrease in bandwidth but SCTP is able to recover much faster than TCP. As well, the dip experienced by SCTP is less severe than that experienced by TCP.[3]

Given that CPU saturation-limited performance in the case of `iperf` (Section 7.1.2), we attempted to discover the extent to which Open MPI's progress engine may add to the compute time for scheduling between BTLs and be responsible for saturating the CPU time in the multiple BTL cases. The MPI middleware was instrumented to report how many times we entered the progress engine during each run of the OSU Bandwidth benchmark. The numbers obtained fluctuated between sender and receiver. At times they were similar in magnitude, while other times they were off by an entire order. Given the inconclusive nature of this test, no definitive conclusion was made as to the exact cause of CPU saturation.

To further investigate CPU saturation, we compared the progress engine behaviors of Open MPI middleware to MPICH2, given that the progress engine design is a major difference between the two implementations. After running the bandwidth tests using `ch3:sctp`, we noticed that MPICH2 makes better use of CMT by obtaining the maximum bandwidth while demonstrating CPU idle times of roughly 10%. For completeness, we also provide Figure 7.5 to demonstrate CMT's capabilities to increase the bandwidth in MPICH2. This find helps support our theory that our Open MPI BTL is not CPU bound but is unable to achieve its maximum possible throughput due to the middleware's use of the CPU.

The range of experiments cover all the different implementations and give relative indications of benefits of each implementation. It is important to recognize that although

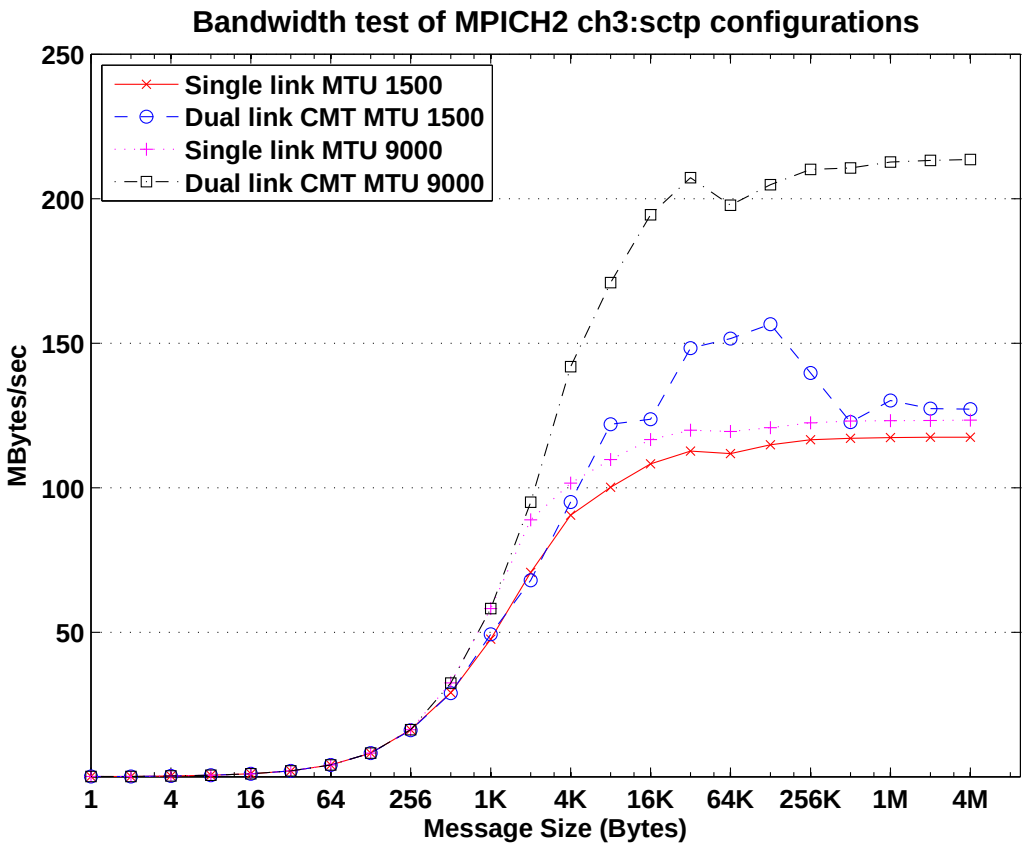

Figure 7.5: OSU Bandwidth Test using MPICH2 ch3:sctp comparing various configurations.

we have attempted to do a fair comparison by keeping all settings identical between the implementations and performing extensive experiments, some variation is to be expected. For example, we have not compared the performance between MPICH2 and Open MPI because their SCTP modules have a completely independent code base. These results are also dependent on the scheduling of socket events and it is possible that the one-to-one or one-to-many implementations, may change because of future improvements to Open MPI with regards to event processing.

7.2 Userspace Stack Microbenchmarks

We developed our own device-agnostic, userspace SCTP implementation that we introduced in Chapter 4; its unique design is able to surpass the Linux kernel SCTP implementation in a single Gigabit NIC configuration and come close to the Linux kernel TCP. Here, we show its initial performance results using a single-homed bandwidth test. First, we introduce in Section 7.2.1 our callback API, an optimization that we designed for our userspace stack to increase performance. After this in Section 7.2.2, we compare the results of the ported socket API to our callback API. Next in Section 7.2.3, we compare our userspace stack's performance running on FreeBSD to Linux. After that in Section 7.2.4, we show the benefit that adding a threshold can have in order to activate our send-side callback. Finally, in Section 7.2.5, we show the improved performance we can get when our userspace stack is no longer device-agnostic by showing that we can modify the device driver to disable interrupt coalescing in order to achieve within 2μs of one-way latency compared to the Linux kernel TCP.

7.2.1 Callback API

In the kernel implementations, ULP interactions cross the kernel-userspace boundary, necessitating a wake-up operation to alert blocking operations that they are complete. For compatibility with the original kernel code, we implemented a ULP wake-up operation for the userspace stack, despite the fact that ULP interactions do not cross protection domains since both are in userspace. As a result, our initial userspace stack has an extra wake-up operation because in the userspace stack, there is a wake-up for the ULP interactions above the stack to the application as well beneath the stack to the socket being used for LLP interactions. Profiling confirmed that our initial implementation of ULP sockets spent an

excessive amount of time acquiring locks.[4]

To avoid this extra wake-up within the socket implementation at the ULP boundary of our userspace stack, we designed an alternative API whose purpose was to bypass locking altogether since the application and transport stack were already in the same protection domain, namely userspace. We implemented a callback mechanism where all operations are non-blocking, so either the send or receive will complete immediately or the user can register a function that is fired as soon as the socket call is no longer blocking. For a receive, this callback is fired when the DATA chunk arrives. For a send, this callback is fired when sent data is acknowledged by the corresponding SACK packet, and space opens up in the send socket buffer beyond some specified threshold; this send-side threshold avoids wasting compute cycles by firing the send-side callback unnecessarily when a send is not going to succeed because a message is larger than the free space in the send socket buffer at the time. The threshold gives the transport protocol a hint to the application's next desired use, so it is therefore an example of a tighter integration of application and transport protocol; its benefits are shown in Section 7.2.4.

Recall from Figure 4.3 in the design description that there are receive threads, one for SCTP/IP packets and another for SCTP/UDP/IP packets. When one of these threads receives a packet, the packet is processed within that receive thread; in addition to doing the necessary processing required by SCTP, the registered callback function from the application is also executed in that same thread. This occurs without requiring a signal to the application as is the case when using costly locks contained within the socket structures. The callback mechanism is an efficient technique for using the userspace stack, compared to sockets. This approach is similar to APIs provided by Infiniband and other Ethernots [7]. It does not avoid locks altogether within the stack but it lessens their use at the ULP boundary; the LLP wake-ups are still necessary beneath our stack because we use the OS

[4]The exact results varied depending on the operating system of the test, as will be presented in Section 7.2.3.

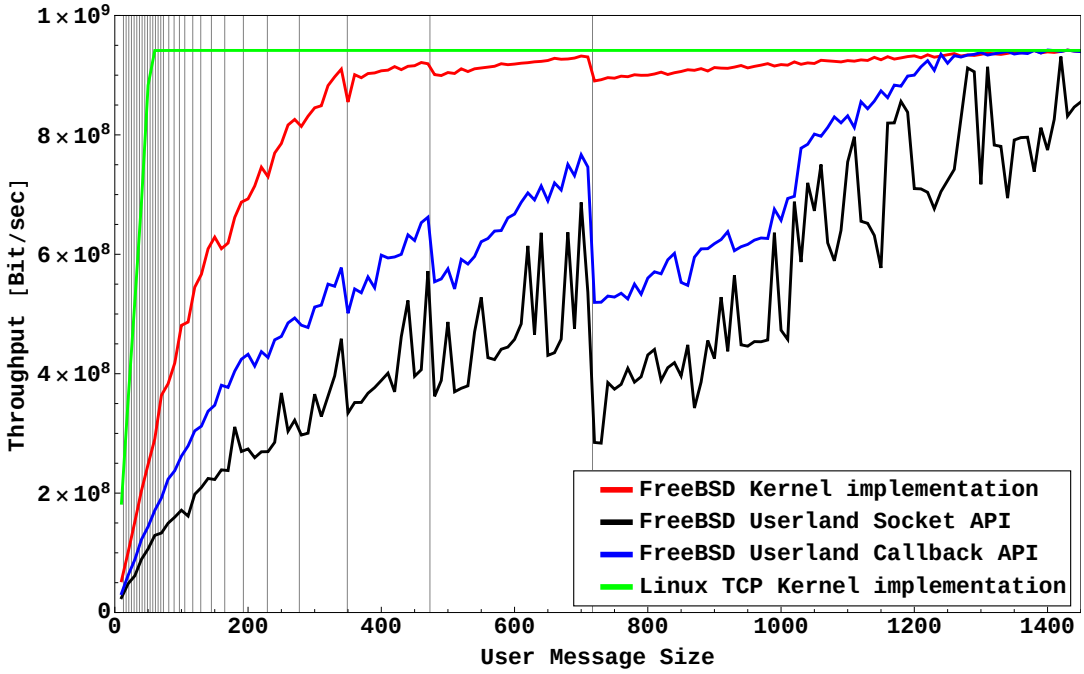

Figure 7.6: Socket API versus callback API for userspace SCTP on FreeBSD

supplied implementation of Berkeley raw sockets and UDP sockets.[5]

7.2.2 Socket API versus Callback API

Our original userspace SCTP results running on FreeBSD are shown in Figure 7.6, with data points every 10 bytes. The faint vertical lines are the packing boundaries for multiple DATA chunks in one packet of MTU 1500, so naturally for SCTP which is message-based, there is a drop just after these boundaries when a packet cannot be efficiently packed.

The top two lines shown are the Linux kernel TCP curve as well as the FreeBSD kernel SCTP results to a bandwidth test. We repeat these two lines in Figures 7.6, 7.7, and 7.8 for

[5] A callback mechanism like ours could avoid the costs of wake-ups in a kernel-based sockets implementation as well however, giving the application hooks to execute their code in the kernel can be unsafe as their execution would block the kernel from doing other more important tasks.

reference. Although not shown in this figure, the FreeBSD kernel had the best throughput of all the SCTP implementations that we tried, matching the theoretical best for message sizes of 340 bytes and higher. Linux kernel TCP and FreeBSD kernel SCTP are equal for larger messages but as expected for bandwidth results, TCP can do better for smaller messages because it does not delimit the message boundaries in its stream of bytes. TCP's header is 20 bytes for all message sizes, while on the other hand, SCTP uses a 16 byte common header and then has a 12 byte header per DATA chunk. Small messages are CPU-bound for SCTP to do this extra processing therefore their throughput is less than TCP; the communication pipe can be filled more easily for SCTP as the message size grows and the CPU is no longer the bottleneck.

The lowest line in Figure 7.6 shows our bandwidth results from our initial API which is similar to the Berkeley sockets API, described in Section 4. As one can see, this original userspace design, which uses the full userspace port of the FreeBSD socket structures, cannot achieve the bandwidth levels that the kernel implementations can.

Figure 7.6 shows that the decrease in locking in the callback mechanism results in higher bandwidth benchmark results.[6] This figure shows that the lowest bandwidth results are of the original port that made use of the socket structures whereas the curve using our new callback mechanism that avoids the locks performed approximately 35% better than the socket API port. However, both of these userspace approaches when measured on FreeBSD are still worse than the kernel implementations. Further optimizations were necessary, as we continue to describe below.

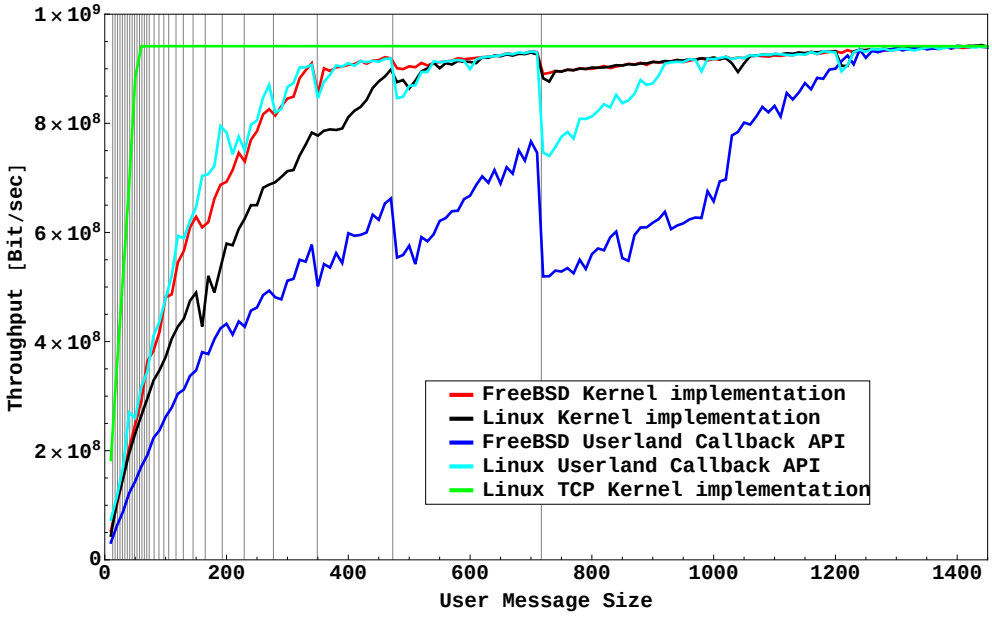

Figure 7.7: FreeBSD versus Linux for userspace SCTP with callback API

7.2.3 Linux versus FreeBSD

In Figure 7.6, we had conducted our tests in FreeBSD. Using the same hardware, we tested our bandwidth results for Linux. These results are shown in Figure 7.7. Figure 7.7 shows several facts. To begin, the FreeBSD kernel SCTP implementation is better than the Linux kernel SCTP implementation, however both of these are better than the FreeBSD userspace results making use of the callback API reported previously. Figure 7.7 shows the userspace stack with the callback mechanism on Linux generally outperforms the kernel SCTP implementation in Linux, particularly for messages < 475 bytes. These results indicate that we could now achieve the desired throughput for an SCTP stack in

[6]The semantics of our callback mechanism are different than those provided by the socket API, however our bandwidth application could easily adapt from the socket API to the new semantics of our callback mechanism.

Linux by using our portable userspace SCTP implementation.

In general, the results for our userspace stack were better in Linux than in FreeBSD. We decided to profile why this is the case. Using the `valgrind callgrind` tool which works with both FreeBSD and Linux, we found that the most time consuming functions are `pthread_mutex_unlock` and `pthread_mutex_lock`, where `pthread_mutex_unlock` is called more often and needs more time. However, we saw that with Linux the impact of these functions was not as great (5% versus 17%), so it was logical to conclude that the implementations of `pthreads` are different on Linux and FreeBSD. It also indicated to us that by using Linux on the same hardware, we could obtain higher bandwidths with our userspace implementation as a result of the improvements to the `pthreads` implementation or using an alternative to `pthreads` such as coroutines that demonstrates consistent performance portability across platforms.

7.2.4 Callback Threshold

We also implemented a send-side callback, however, our bandwidth results still were not as high as a kernel implementation. To investigate, we performed bandwidth interoperability tests between our userspace stack and the FreeBSD kernel implementation, doing (1) kernel-send/userspace-receive followed by (2) userspace-send/kernel-receive. The results were uneven, indicating a bottleneck. The bandwidth for (1) was higher than the bandwidth for (2). The send-side implementation of our userspace stack was therefore the bottleneck. Not only is the send-side problematic because it has to deal with SACK processing, but the send-side callback was firing too frequently since our original send-side callback scheme invoked the callback with each SACK.

In light of excessive firing of the send-side callback, we modified our `register_send_cb` call so that the user can now choose to provide a threshold such that the send-side callback was only called when a user-specified amount of free socket buffer space is present. The

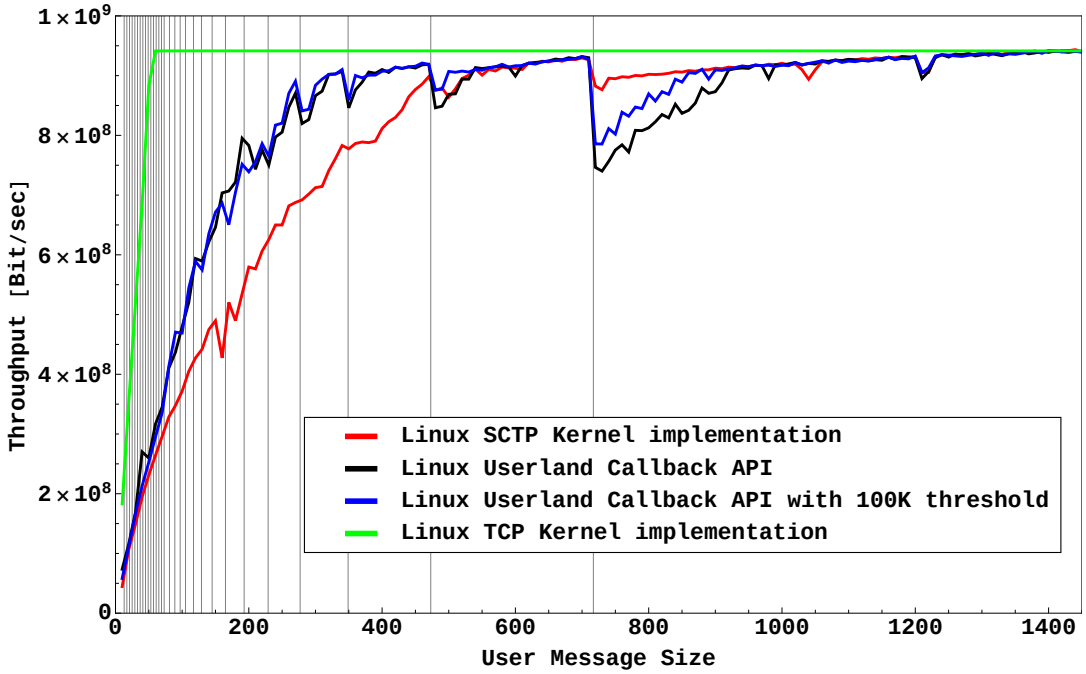

Figure 7.8: Send-side callback with each SACK versus at a 100KB threshold

OS and Transport	Time (μs)
Kernel Linux TCP	21.48
Userspace FreeBSD SCTP CB w/o AIM	23.24
Userspace FreeBSD SCTP CB w/ AIM	25.2
Userspace Linux SCTP CB	30.5
Kernel Linux UDP	12.5
Kernel FreeBSD UDP w/o AIM	12.26
Kernel FreeBSD UDP w/ AIM	12

Table 7.1: One-way latency for 30 byte payload

idea was, that if the send-side callback is fired less often, we could improve the throughput. The threshold is compared to the free space in the socket send buffer. Typically this is the size of the next message that will be sent. We compare our new approach with the original in Figure 7.8 and as is shown there, making use of a threshold increased the bandwidth for some message sizes up to 5%.

7.2.5 Driver Effects on Latency

Our initial latency tests showed that the latency of a kernel implementation was much faster than our userspace SCTP stack. We suspected that this was because the NIC's device driver coalesces interrupts by way of its automatic interrupt modification (AIM) scheme. Coalescing would increase message latency for small messages whereas for larger messages, bandwidth would be the limiting factor. In the default driver, AIM was enabled and there was no simple mechanism for disabling it other than recompiling the driver.

As a solution to this, a driver modification would be necessary. So far, all of our modifications have been either within the SCTP stack itself or adjusting the upper-layer boundary to the stack. Michael Tüxen provided a simple patch for the device driver to be

able to adjust when AIM occurs. As shown in Table 7.1 for 30-byte messages, this slight modification to the FreeBSD driver can decrease the one-way latency of the userspace SCTP stack to be on par with that of the Linux kernel's TCP implementation. Without modifying the lower-layer of the userspace stack, we have the optimal combination of portability with high performance, as was shown in Figure 4.1-(2). Future work on integrating the lower-layer of the stack to a particular device as is shown in Figure 4.1-(3), will make our userspace stack less portable but result in further decreases in latency than have been initially demonstrated here.

7.2.6 Userspace Stack Conclusions

We have provided a tuned userspace implementation of SCTP which enables SCTP's additional features on most major platforms, including Linux. For small messages less than 480 bytes, our userspace stack outperforms the Linux kernel SCTP implementation. Our results show that having the transport protocol and the application in the same protection domain can increase performance, as suggested by Jacobson and Felderman [50]. By putting the transport stack into userspace and providing a callback API, our MPI implementation unifies the control loops and makes the implementation more event-driven. Our optimizations outperform the Linux kernel implementation of SCTP even without device-specific modifications. Making use of our callback mechanism, our SCTP userspace stack was able to perform on par with the Linux kernel TCP stack for large messages in bandwidth microbenchmarks. A simple driver adjustment decreased the latency measurements of our userspace SCTP stack for small messages by disabling interrupt coalescing. Future device-specific optimizations such as utilizing zero-copy to obtain full kernel bypass could yield further performance gains for the userspace stack.

7.3 NAS Parallel Benchmarks

While the previous section looked at the performance of bandwidth and latency microbenchmarks, real applications behave differently than microbenchmarks. The NAS Parallel Benchmark (NPB) suite is a standard set of parallel applications written in MPI that each perform a common algorithm. Each parallel application demonstrates a different communication pattern. Overall, they are meant to typify the applications that can be run across a parallel cluster.

The goal of this section is to investigate the performance of the NPBs over SCTP. We make use of our MPI implementation that runs over our userspace SCTP stack in order to demonstrate the initial NPB results in Section 7.3.1, and compare them to TCP. In our analysis of the NPBs, we observe the benefit of an SCTP acknowledgment optimization called I-Bit which allows the application to inform SCTP of the application protocol and thereby control the timing of the acknowledgments; in Section 7.3.2, we conclude with a larger scale investigation of the I-Bit using MPI-NeTSim.

7.3.1 Userspace Stack

Here, we investigate the NPB performance of our two userspace SCTP-based MPI middleware designs, and we compare them to kernel-based TCP all using MPICH2. Throughout all tests, `ch3:copy` and `ch3:mbuf` yielded identical results, so we report only `ch:copy` when portable hardware is being used as the full benefits of the `ch3:mbuf` design will not be had until it is tailored to specific hardware and the same `mbuf` is zero-copied all the way from the MPI middleware down to the wire.

We ran a suite of various applications across 4 and 8 nodes using `ch3:copy` and `ch3:sock` to respectively compare MPI programs using our userspace SCTP stack and the TCP kernel stack. The communication patterns of these applications differed, and they were all executed over four different sizes of data.

At first, the TCP results were always equal or several times faster than the use of our SCTP stack. This was most simply shown for the embarrassingly parallel (EP) test performed on a small data size. EP has one small phase of communication at the beginning, a proportionately much larger computation phase in the middle, and then a final communication phase. In the `ch3:sock` for data size "S", this consistently took 0.22 seconds for all runs. However, for `ch3:copy`, this varied between 0.22, 0.42, or 0.62 seconds. This was a step-wise variance as in there was never a number between these values. Further investigation showed that the default value for SACK frequency was to SACK every other DATA chunk, however this application sent only one message and the delayed SACK timeout timer of 0.2 seconds was firing. This was avoided in TCP by disabling Nagle's algorithm in `ch3:sock`. In SCTP, there is Nagle's algorithm for the clustering of sending data but also there is the I-Bit [95, 111] which can be used for each SCTP send in order to SACK immediately instead of clustering the sending of acknowledgments. The communication pattern in MPI is often a query followed by a response, meaning that the sender has to wait for acknowledgments before it can go on. As the receiver always waits for two packets by default in the standard before it sends an acknowledgment, an immediate answer is beneficial. For our tests, once the I-Bit was set, all of the applications in the NAS Parallel Benchmark suite ran over `ch3:copy` matched those ran over `ch3:sock`, for all applications, number of nodes, and data set sizes.[7] Next, we use MPI-NeTSim to investigate the use of I-Bit and Nagle further.

7.3.2 Using MPI-NeTSim for I-Bit and Nagle

For latency sensitive MPI applications, saving wall clock time is important. When communication and computation cannot be overlapped, transport protocol timeouts lessen the performance by increasing the runtime of the application. One such timeout is the delayed

[7] The results are identical so no graph or chart is shown.

acknowledgment timer. In the previous section, we investigated this using our userspace SCTP stack on a modest sized cluster. Next, we use MPI-NeTSim to do a larger scale investigation of the effects of the I-Bit and Nagle on the NPBs.

The I-Bit is sent by the sender to tell the receiver not to delay the acknowledgment of that message when processed. In this sense, it is similar to disabling Nagle's algorithm when packing outbound data only it is instructing the receiver to avoid any delay in the sending of the acknowledgment.

There are several possible scenarios where the I-Bit can be applied beneficially such as error prone links, short-term connections, and query-response protocols [95]. In our experiments, we show that the I-Bit reduces the negative effect that the Nagle algorithm [67] has on performance for MPI programs. Sending small messages in packets of their own increases the network load because of the transmission of unnecessary header bytes. Nagle's algorithm which is enabled by default prevents the sending of packets that could be filled with more messages if there is data that has been sent but not yet acknowledged. An exception to this rule is the bundling of data with an acknowledgment that has to be sent.

The MPI message flow is characterized by an exchange of small messages followed by the actual data that can easily exceed 64 KBytes. Thus, the small messages may only be sent if all data sent so far has been acknowledged or if an acknowledgment has to be sent. When transmitting large data sets, which have to be fragmented, the Nagle algorithm does not present an obstacle. Although it is possible to disable the application of the Nagle algorithm, it is not desirable.

The I-Bit can be applied at the transport and the application layer. If SCTP is configured to apply the I-Bit at the transport layer, it is set to the last chunk allowed to be sent in one `cwnd` flight, and for large messages, to the penultimate fragment. At the application level, the user can decide which messages should be acknowledged immediately by setting the I-Bit on a per message basis. In our simulation, both variants are implemented.

We decided that the I-Bit should be applied whenever a message was too small to fill a complete packet. To show the impact of this flag, we set up a two layer tree for 16 hosts with a bandwidth of 100 Mbps and a latency of $1\mu s$ on all links.

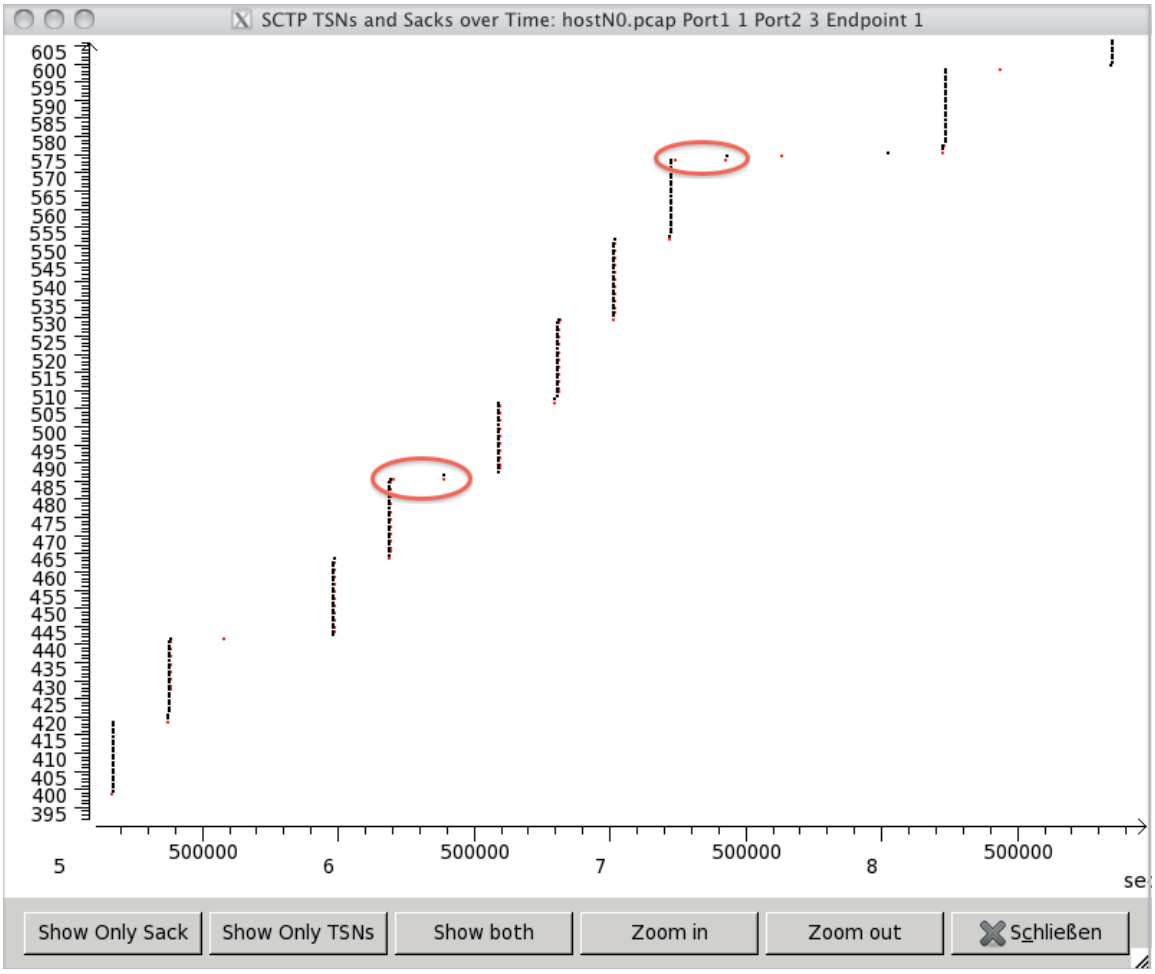

Figure 7.9: Message flow, when Nagle is enabled and the use of the I-Bit is disabled

The effects of the Nagle algorithm are shown in Figure 7.9. It is a screen shot of the graphical representation of the message flow of an SCTP connection in the Wireshark packet analyzer [118]. This trace was written on the side of the host sending the data, as

output from MPI-NeTSim. The values on the y-axis are the sequence numbers of the user data messages, on the x-axis the time in seconds is plotted. The black dots represent the messages containing user data, and the red dots represent the corresponding acknowledgments. The circled areas point out two of the cases when the sender has to wait for an acknowledgment before it is allowed to send the next data message which was withheld by the Nagle algorithm.

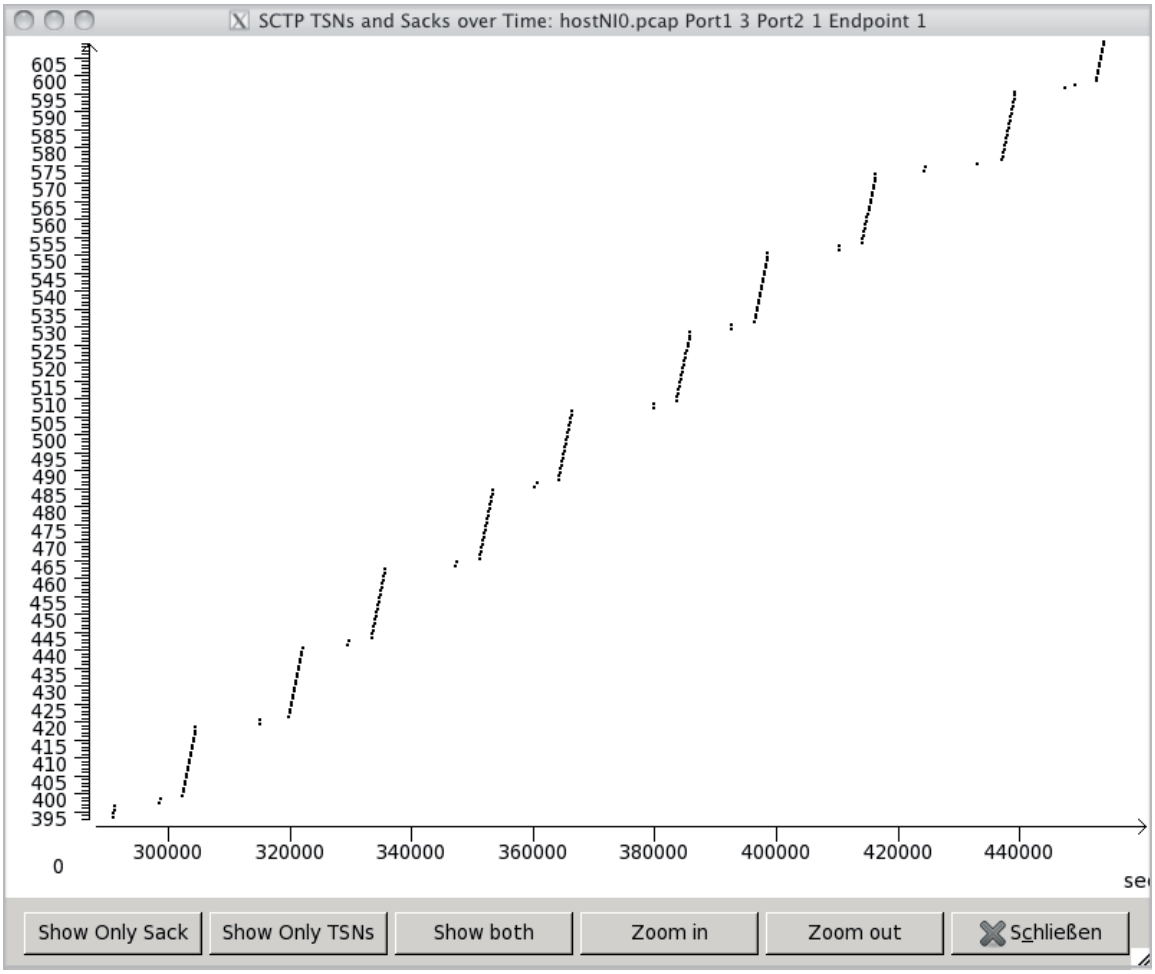

Figure 7.10: Message flow, when Nagle is enabled and the use of the I-Bit is enabled

Nagle	**I-Bit**	**Run time**
-	-	5.69 s
-	+	5.69 s
+	-	115.37 s
+	+	5.69 s

Table 7.2: Run time for CG.A.16 in the presence or absence of the I-Bit and the Nagle algorithm

As a comparison, Figure 7.10 shows the same scenario, but using the I-Bit. The y-axis encompasses the same range of user messages as Figure 7.9, whereas the x-axis of Figure 7.9 represents 20-fold the time range of Figure 7.10. As a result, Figure 7.10 is more detailed yet the acknowledgments are not to be seen, because they are hidden by the data messages.

Table 7.2 shows the run times of various configurations for the Conjugate Gradient (CG) application of the NAS Parallel Benchmark. CG uses the conjugate gradient method to estimate the smallest eigenvalue of a matrix. Class A of CG was executed across 16 machines using MPI-NeTSim for all combinations of the application of the I-Bit and the Nagle algorithm. The difference between enabling the Nagle algorithm with and without setting the I-Bit at the same time is striking. As an optimization, the I-Bit can completely eliminate the negative side effects of the Nagle algorithm; in this sense, our userspace results were confirmed with these from the simulation, and on a larger scale.

7.4 Performance Summary

In this chapter, we showed that SCTP and the CMT extension better utilizes the additional bandwidth provided by a second NIC than application-level data striping; CMT operates at a lower level and provides a better opportunity to balance traffic across the links.

Our userspace stack and its optimizations were then presented and it was shown that

comparable performance can be shown in our userspace SCTP implementation on Linux to its kernel TCP implementation for large message bandwidth, latency, and the NPBs; for small message bandwidth, he userspace SCTP stack outperformed the Linux kernel SCTP implementation. The effectiveness of the I-Bit feature was noticed when executing the NPBs over the userspace stack and was then verified in MPI-NeTSim on a larger scale. In summary, these results conclude that these transport features can increase the performance of MPI middleware.

Chapter 8

Simplicity of Design

This thesis contends that SCTP and its extensions enhance the design simplicity of MPI implementations running on Ethernet-based cluster systems. This is shown in this chapter but central to this is understanding exactly what is meant by "design simplicity". Typically, transport protocols implement general features in a complete way so that their applications can utilize them and not have to reimplement them. For example, TCP implements in-order reliability on a single path, so its applications do not need to reimplement checks for in-order arrival since the underlying transport transparently ensures this feature. Applications are simplified if one of their features can be implemented in the transport protocol in a general and complete manner. In this sense, use of SCTP compared to TCP simplifies MPI implementations because more protocol features are available for use with SCTP by the MPI implementation; if these features are utilized by the MPI implementation, then they do not have to be reimplemented in the middleware. Overall, this is what is meant by "design simplicity".

Unlike Chapters 6 and 7, design simplicity is illustrated through observations, designs and features rather than quantitative results. Two of our MPI-related projects are described and show how SCTP simplified their particular designs in comparison to their TCP counterparts. The first project is our design of a simulated Ethernet-based RDMA-capable stack; how its use of SCTP simplified its design compared to TCP is described in Section 8.1. Secondly, in Section 8.2 we relate our use of the CMT SCTP extension[49] to similar approaches to multirailing implemented in middleware, and show how CMT simplifies the

design and eases the configuration for the user. Finally, we summarize in Section 8.3.

8.1 SCTP Simplifies IP-based RDMA

To cope with higher transmission rates, some network fabrics enable RDMA. Traditionally, network interface cards read incoming data to their own local buffers, then the data travels through the front-side bus to the CPU registers before it then travels across this front-side bus again to be placed in main memory. RDMA-enabled network interfaces avoid unnecessary data movement by interacting with main memory itself. This means that once a connection is established between a pair of endpoints, memory can be adjusted on the remote end without the involvement of the remote CPU, freeing up cycles and avoiding unnecessary traffic on the front-side bus. Performance can increase since the OS can be bypassed costing fewer context switches and also avoiding some copying. An example of an RDMA-enabled network fabric is Infiniband. For Ethernet, the Internet Wide Area RDMA Protocol (iWARP) [89] and more recently RDMA over Converged Ethernet (RoCE) [22] have been proposed for enabling RDMA. We look at how SCTP could be beneficial for iWARP by first describing some background of the various iWARP layers in Section 8.1.1 and then demonstrating in Section 8.1.2 the design benefits of using SCTP compared to TCP for iWARP.

8.1.1 Internet Wide Area RDMA Protocol

iWARP is a stack consisting of specifications providing RDMA support for the existing IP infrastructure to overcome the performance deficiency of traditional Ethernet-based network devices. RDMA-capable Network Interface Card (RNIC) Ethernet hardware is necessary in order to take advantage of the performance benefits of RDMA, although software implementations exist for iWARP's initial roll-out [25, 26]. iWARP RNICs complement other hardware offloading schemes that have either portions of the stack implemented in hard-

ware like the computation of segmentation and checksums, or full protocol offload devices such as the Chelsio S310E family [19] which provides TCP offload in addition to TCP-based RNIC support.

The iWARP stack is illustrated in Figure 8.1. At the bottom, the standard specifies using either SCTP or TCP. In order to understand this decision, first the layers are introduced from the top down.

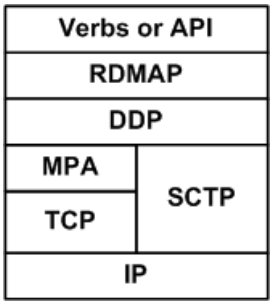

Figure 8.1: iWARP stack layout

- **Verbs API**: The verbs or API layer exposes the RDMA operations to the user application [52]. Furthermore, it also provides direct access to the RNIC. The verbs layer communicates directly with the RDMAP layer (described next). The iWARP standard does not provide any exact verbs specification as only the necessary functionalities of this layer are described in the standard. As a result, this layer is flexible in terms of API design. However, without any standardization, every vendor can potentially implement their own API resulting in application incompatibility. OFED [77] has attempted to remedy this by defining a general API appropriate for RNICs supporting iWARP, Infiniband, and RoCE.

- **Remote Direct Memory Access Protocol (RDMAP)**: The RDMAP layer provides both traditional send/receive and read/write services to the user applica-

tions [5, 90]; send/receive requires the applications of both sides involved in the communication while read/write requires only one side to be involved. RDMAP itself is a fairly thin layer, but it depends on the lower layer protocol to provide the corresponding service. In this case, the lower layer is the Direct Data Placement layer.

- **Direct Data Placement (DDP)**: The DDP layer is responsible for placing the data in the proper place after being received [96]. It supports two models of message transfer: Tagged and Untagged. Tagged message transfers require the use of a steering tag (within the DDP header) to allow the receiver to place the incoming data directly to a user buffer that is registered with the specified steering tag. In the case of untagged message transfers, they provide the traditional Copy-in/out model by specifying both the message and buffer queue number. Both models require a buffer to be pre-posted in order to ensure correctness. Moreover, each DDP segment is message-based and each fragment must have a corresponding header attached. As a result, each DDP segment preserves the self-describing property of the protocol and can be placed at the appropriate location without any information from other DDP segments. In addition, DDP requires reliable delivery from its lower layer protocol[1]; either TCP or SCTP can be used beneath the DDP, but TCP cannot be used directly.

8.1.2 SCTP versus TCP for iWARP

DDP is an end-to-end protocol but the intermediate nodes do not necessarily support DDP. It is possible that intermediate switches/routers can splice the packets; this is known as the "Middle Box Fragmentation" problem. In the worst case, fragmentation could cause the self-describing property of an incomplete DDP segment to fail. To overcome this challenge

[1] An extension to iWARP has been proposed to enable unreliable support over UDP [37, 38]. The author's claim that existing datagram-based applications can also utilize the benefits of RDMA. A connectionless, unreliable iWARP standard can increase the scalability of applications since less overhead is required.

for TCP, the Marker PDU Aligned (MPA) layer was introduced to provide message framing capability for TCP/IP, serving as an extension to TCP/IP in order to maintain backward compatibility [24]. The MPA's function is to insert markers at specific places in order for the receiver to have a deterministic way to find the markers and find the right header for the DDP segment. Each MPA segment is known as the Framing Protocol Data Unit (FPDU). In addition, the MPA layer adds a more robust CRC32c checksum to each FPDU in order to preserve data integrity. During connection initialization, iWARP endpoints are required to exchange information about the specific location of MPA markers. It has been shown that the MPA layer is fairly complex as the placement of the marker is fairly tricky and this problem becomes worse in the case of supporting iWARP with multiple network devices in which out-of-order communication can occur [6]. Moreover, along with the added checksum calculation, the MPA layer can add a considerable overhead to the overall performance and even result in TCP conformance problems [85].

Alternatively, SCTP is a candidate to replace MPA-TCP. The SCTP stack provides reliable message transfer, stronger CRC32c checksum and message framing support natively.[2] Multistreaming and unordered message delivery can be used to implement DDP segment independence. IETF RFC 5043 [12] specified how SCTP would be used by the DDP; we were the first to implement this in Tsai et al. [106] and research SCTP's potential for use with iWARP.

Given the increased control and overall number of features available, SCTP is able to adapt to new technologies easier than TCP. As reported by Tsai et al. [106], we found that SCTP simplifies an iWARP implementation because its features can be used directly to match the requirements of the upper layer to reliably deliver self-contained and self-describing DDP segments. Message framing in SCTP overcame the problems of middlebox fragmentation and marker placement present in the MPA layer for TCP because SCTP

[2]Messages are delimited by default but users can avoid message framing per `send()` and send all of their data in one stream like TCP using the `SCTP_EXPLICIT_EOR` socket option[99].

messages are not fragmented by default.[3] Data chunks in SCTP can be assigned a type using the Payload Protocol Identifier (PPID) field in order to distinguish between DDP segments and session control data [12].

We demonstrated these design benefits by simulating SCTP-based iWARP network cards and then implementing the MPI-2 Remote Memory Access (RMA) functions using this simulated device in Tsai et al [106]. This showed that using SCTP-based MPI middleware for MPI programs that use RMA simplifies the underlying iWARP layers by eliminating the complex MPA layer necessary when TCP is used. Although TCP is the most common choice for iWARP, it requires the problematic MPA layer in order to adapt. By using SCTP as the transport layer protocol for iWARP, the difficulties are eliminated without an extra layer and SCTP features for multi-streaming and multi-homing can be exploited to respectively enhance the scalability and fault tolerance capabilities of the implementation of MPI RMA functions [11].

8.2 SCTP Simplifies Multirailing

In SCTP, an association between two endpoints can be aware of multiple IP addresses. The base RFC 4960 SCTP protocol uses information about multiple paths between endpoints strictly for the purposes of fault tolerance whereas the CMT extension utilizes all paths simultaneously in order to maximize bandwidth. The reliability of multihoming and CMT for MPI was investigated in Section 6.1 whereas in Section 7.1, its performance was studied compared to Open MPI's approach of multirailing in the middleware. In this section, we instead focus on how the multirailing feature of SCTP compares to middleware-based multirailing and other similar approaches. We show how use of SCTP and CMT simplifies the configuration of multirailing for MPI programs compared to other approaches.

[3] Partial message fragments are not delivered to SCTP applications using the default settings, however this can be adjusted using the `SCTP_FRAGMENT_INTERLEAVE` socket option if need be [99].

MuniSocket[65, 66], RI2N/UDP [74] (both of which use UDP), and SCTP all provide an abstraction of an association where the end-point of a connection consists of one or more IP addresses. For example, if two hosts each had two NICs on two separate networks, all these approaches can selectively bind the two connections on one host to the second host. The actual route taken depends on the IP layer where the network layer's routing function chooses a source address to the specified destination address based on routing information. For all these systems, routes must be set appropriately and switches configured, in order for there to be independent paths between the endpoints.

In RI2N/UDP and Open MPI[4], the characteristics of the links are determined statically by user-provided configuration information. MuniSocket advocates its additional flexibility over these static approaches by implementing a network-discovery mechanism for dynamically determining these parameters at start-up time. The drawback of these approaches is that they are unable to adapt dynamically to the changes in the condition of the network as an application runs. SCTP and in particular CMT are constantly monitoring the network, using both heartbeats and round trip time and adapts to any changes in the network conditions.

In the case of Open MPI, one of the most useful applications of message striping is its use with other network fabrics such as InfiniBand where there are often multiple independent lanes. In addition, not specific to MPI, there is also the work on Madeleine II [4] that has investigated techniques for improving communication for heterogeneous collections of Ethernot type networks [7]. Our SCTP-based modules for Open MPI do not preclude the use of having multiple BTLs using different protocols or network fabrics. For example, one could have an InfiniBand BTL running together with our SCTP one-to-many BTL with CMT turned on. Open MPI would do message scheduling between the BTLs. The SCTP BTL combines together all of the Ethernet interfaces into a single association, exposing a

[4]Open MPI and its middleware-based multirailing were described in Section 3.3.

single socket interface to the progress engine. The transport layer is the lowest end-to-end layer for the Ethernet-based paths, so since multirailing of those paths can be completely done transparently in the transport protocol using CMT, it seems appropriate to simplify the MPI middleware to place the multirailing functionality there instead of in the MPI middleware for the paths sharing the common Ethernet fabric; the Infiniband BTL can still be used for that network fabric and the middleware can schedule between the SCTP BTL and the Infiniband BTL. In this regard, CMT and our SCTP module for Open MPI complement the multiple-BTL support for heterogeneity.

There is one other possible technique available for using multiple interfaces, which is channel bonding [88]. Channel bonding is supported by several network card and switch manufacturers and allows several ports on the card to appear as a single port. Channel bonding is completely transparent to the application software and is a hardware solution to multi-railing. The major advantage of channel bonding is that it is able to schedule Ethernet frames inside the NIC to take advantage of the bandwidth and latency. The disadvantage is that link aggregation and channel bonding has to be configured along the entire path, across switches, to get full advantage of it, and this is often complicated if different operating systems are used since this solution is non-standard. Another disadvantage of this approach is the lack of fault tolerance for the switch itself. Once the full path is configured and multiple switches are involved, this technique only allows link binding between two switches directly connected by the multiple links; it does not however recover from a failure in the switch itself [74], as SCTP is able to do.

Another manner in which SCTP gives an application more control compared to other transport protocols is its control of retransmission timers and retransmission attempts. As was described in Section 6.1, a failover is triggered in SCTP multihoming when some number of failures occurs. The user can specify the time a packet has to be delivered before failure is determined; the number of failures until path failover occurs can also be set by the application in order to match its needs. Telephony applications have rigid timing

requirements and it was this lack of control in the case of TCP retransmission timers that inspired the design of SCTP to be more controllable [101]. SCTP provides applications a set of solutions as possibilities and allows them to adjust the features to match their exact requirements. We were able to specify the timers and maximum number of retransmissions that was appropriate for MPI in an Ethernet-based cluster environment.

In conclusion, SCTP with CMT is almost completely transparent to the MPI application once SCTP-based middleware is used; CMT requires almost no effort to configure to take advantage of its features as it is enabled by way of a simple `sysctl` setting. As with all cluster systems, the routing tables and switches will need to be appropriately configured to ensure independent routes, but network properties do not need to be statically configured as they do in other approaches, and scheduling dynamically adapts to emergent network conditions. Interestingly, the mptcp IETF working group has recently formed and is working to support this feature in TCP [45].

8.3 Simplicity Summary

Overall, the items discussed in this chapter show that SCTP enhances the design simplicity of MPI. This was illustrated with two specific examples:

- First, we showed how for MPI-2 RMA operations, SCTP can adapt to enabling Ethernet-based RDMA for iWARP more easily than TCP; SCTP simplifies the iWARP stack by avoiding the cumbersome MPA layer necessary for TCP. The work was published in the Proceedings of the 2008 IEEE International Parallel and Distributed Processing Symposium 2008 [106].

- The second example showed that the CMT-based design for multirailing enabled more capabilities compared to other approaches and it also simplified the configuration since it occurred dynamically and is transparent to the user. This work appeared originally

in the Proceedings of the 2007 Euro PVM/MPI users' group meeting [81] with an expanded version appearing in the March 2010 issue of the Journal of Parallel and Distributed Computing [80]. The reliability of CMT was shown in Section 6.1 whereas the performance of CMT was shown in Section 7.1.

Chapter 9

Conclusion

The major claim in this thesis is that by adding features to the transport protocol, additional capabilities are enabled for MPI implementations using Ethernet-based commodity clusters. In order to support this claim, SCTP was used as a vehicle to provide these features after which two distinct phases were necessary: (1) the development of SCTP-based software for parallel MPI applications in the cluster environment to utilize the new transport features, and (2) using the tools to investigate the additional reliability, performance, and design simplicity that these new transport features provide to MPI implementations in an Ethernet-based cluster. These phases were presented in this thesis in two respective parts, Part I for the tools and Part II for the capabilities.

Part I of this thesis describes the SCTP-based tools created and contributed to the community. These tools show how to use the new transport features. The central tool is our SCTP-based MPI middleware design because it enabled the use of SCTP for all MPI programs on a variety of platforms. This design was developed and contributed to MPICH2 and Open MPI, two major open-source releases of MPI. Variations of this MPI design were created (a) with the development of our userspace SCTP stack to investigate protocol onload, (b) with the simulated SCTP stack to easily toggle communication parameters, and (c) with our SCTP-based MPI using RDMA. Overall, the tools developed in this thesis formed an infrastructure to conduct the experiments necessary for this thesis, and in addition, they contributed to both the SCTP and HPC communities since they were released with mature open source projects.

We then investigated the various SCTP capabilities in Part II that are added to MPI implementations regarding reliability, performance, and design simplicity. With regards to the reliability of TCP, experiments showed that use of SCTP provided additional fault tolerance with multihoming, increased acknowledgment efficiency, as well as resilience to errors with its stronger checksum and multistreaming feature. SCTP could provide equal MPI application performance compared to TCP on Linux when using our optimized userspace SCTP stack. Further experiments showed the benefits of SCTP-specific features to better utilize system bandwidth; these features include transport-based data striping as well as the use of the I-Bit. The simplicity of design that SCTP provides by giving the user more flexibility and control over its features was also shown. Altogether, these results support the major claim in this thesis, that the addition of the appropriate transport features enables further capabilities compared to existing transport protocols for MPI implementations over Ethernet.

There are several future directions the ideas in this thesis could be taken. The next generation of Ethernet, called Converged Enhanced Ethernet, will increase the features available in the link layer potentially rendering some features present in today's transport protocols as unnecessary [22]. The flexibility of SCTP features and its design that has extensibility, may be play a key part to adapting to the network fabrics of the future [32].

There is also the upcoming wide-spread deployment of IPv6. A talk in January 2011 by Vinton Cert at linux.conf.au [17] discussed the imminent depletion of the IPv4 address space and the challenges of moving to IPv6. Being designed substantially later than TCP, SCTP's design was able to provision better for the advent of IPv6. A recent IETF proposal called Happy Eyeballs [117] has suggested the use of SCTP to aid in the shift from IPv4 to IPv6. Understanding the benefits of SCTP for IPv6 is grounds for future work.

Another possibility for future work is to extend the device-agnostic design of the userspace SCTP stack presented in Chapter 4. The stack presented there uses the kernel IP implementation so that it can make use of device drivers provided by the kernel. The potential

for future work is to adapt the userspace SCTP stack to instead use a specific network controller with zero-copy, like the Intel 82599 chip [46]. This could provide additional performance benefits by doing less copies as well as bypassing the kernel thereby avoiding context switches. Integrating such a userspace stack more tightly with MPI middleware could improve performance.

The additional features of SCTP increase an application's reliability but also its memory footprint because additional state must be kept for multihoming, multistreaming, and SACK; Ono and Schulzrinne have reported that for Linux, an SCTP association uses five times the memory as a TCP connection but by disabling SACK, can be reduced to twice the size of TCP [76]. Closing this gap in memory usage between TCP and SCTP remains as future work in both the Linux and FreeBSD kernel implementations. Part of the memory increase is due to maintaining the state associated with the improved reliability features of SCTP. Depending on the characteristics of the network fabric, it may be possible to reduce SCTP's memory usage by selectively turning off some of the features. However, in less reliable networks over commodity parts, there is a benefit in maintaining the extra state.

A final possible direction for future work is to tune our SCTP-based middleware to use the latest generation of MPICH2. For full fault tolerance, this would require enabling SCTP to be used with the latest hydra process manager. For the communications amongst MPI processes, a new subsystem called Nemesis [14] is now used. In Nemesis, for processes on the same host, an efficient transportation means is used to get better performance for multicore architectures. Processes that communicate over the network each have their own sockets and connections. SCTP one-to-many sockets and multistreaming could be used to funnel communications from the processes of one host to the processes of the other, saving resources and keeping the sliding window large.

Bibliography

[1] Amazon. Elastic compute cloud (ec2). Available at `http://aws.amazon.com/ec2/`, accessed on 4/4/2011.

[2] Argonne National Laboratory. MPICH2 homepage. Available from `http://www-unix.mcs.anl.gov/mpi/mpich/`, accessed on 4/4/2011.

[3] Rob T. Aulwes, David J. Daniel, Nehal N. Desai, Richard L. Graham, L. Dean Risinger, Mark A. Taylor, Timothy S. Woodall, and Mitchel W. Sukalski. Architecture of LA-MPI, a network-fault-tolerant MPI. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, Sante Fe, New Mexico, April 2004.

[4] Olivier Aumage, Elisabeth Brunet, Raymond Namyst, and Nathalie Furmento. NewMadeleine: A Fast Communication Scheduling Engine for High Performance Networks. In *Communication Architecture for Clusters Workshop (CAC 2007), workshop held in conjunction with IPDPS 2007*, March 2007.

[5] S. Bailey and T. Talpey. The Architecture of Direct Data Placement (DDP) and Remote Direct Memory Access (RDMA) on Internet Protocols. Available from `ftp://ftp.rfc-editor.org/in-notes/rfc4296.txt`, accessed on 4/4/2011, December 2005.

[6] P. Balaji, W. Feng, S. Bhagvat, D. K. Panda, R. Thakur, and W. Gropp. Analyzing the Impact of Supporting Out-of-Order Communication on In-order Performance

with iWARP. In *In the IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, November 2007.

[7] Pavan Balaji, Wu chun Feng, and Dhabaleswar K. Panda. Bridging the Ethernet-Ethernot Performance Gap. *IEEE Micro*, 26:24–40, 2006.

[8] 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.

[9] Victor R. Basili, Jeffrey C. Carver, Daniela Cruzes, Lorin M. Hochstein, Jeffrey K. Hollingsworth, Forrest Shull, and Marvin V. Zelkowitz. Understanding the high-performance-computing community: A software engineer's perspective. *IEEE Softw.*, 25(4):29–36, 2008.

[10] M. Becke, T. Dreibholz, J. Iyengar, P. Natarajan, and M. Tuexen. Load Sharing for the Stream Control Transmission Protocol (SCTP). Available at `http://tools.ietf.org/html/draft-tuexen-tsvwg-sctp-multipath-01`, accessed on 4/4/2011, Dec 2010.

[11] C. Bestler. Multi-stream MPA. *In the Proceedings of IEEE International Cluster Computing*, Sept. 2005.

[12] C. Bestler and R. Stewart. Stream Control Transmission Protocol (SCTP) Direct Data Placement (DDP) Adaptation. IETF RFC 5043, October 2007.

[13] A. Biswas. Support for Stronger Error Detection Codes in TCP for Jumbo Frames, draft-ietf-tcpm-anumita-tcp-stronger-checksum-00 (work in progress). *IETF*, May 2010.

[14] Darius Buntinas, Guillaume Mercier, and William Gropp. Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2

using the Nemesis communication subsystem. *Parallel Computing*, 33(9):634 – 644, 2007. Selected Papers from EuroPVM/MPI 2006.

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

[16] A. Caro. *End-to-End Fault Tolerance Using Transport Layer Multihoming*. PhD thesis, Computer Science Dept., University of Delaware, 2005.

[17] Vinton G. Cerf. linux.conf.au talk. Available at `http://www.isoc-ny.org/p2/?p=1713`, accessed on 4/4/2011, January 2011.

[18] Jeff Chase, Andrew Gallatin, and Ken Yocum. End-System Optimizations for High-Speed TCP. *IEEE Communications Magazine*, 39:68–74, 2000.

[19] Chelsio Communications. S310E Product Brief. Available from `http://www.chelsio.com/assetlibrary/products/S310EProductBrief090721.pdf`, accessed on 4/4/2011.

[20] Chris Hegarty. SCTP in Java. Available at `http://openjdk.java.net/projects/sctp/`, accessed on 4/4/2011.

[21] Giuseppe Ciaccio and Giovanni Chiola. GAMMA and MPI/GAMMA on Gigabit Ethernet. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 129–136, London, UK, 2000. Springer-Verlag.

[22] David Cohen, Thomas Talpey, Arkady Kanevsky, Uri Cummings, Michael Krause, Renato Recio, Diego Crupnicoff, Lloyd Dickman, and Paul Grun. Remote direct memory access over the converged enhanced ethernet fabric: Evaluating the options. *Symposium on High-Performance Interconnects*, 0:123–130, 2009.

[23] Bruce Cran. SctpDrv: an SCTP driver for Microsoft Windows. 2010. Available at `http://www.bluestop.org/SctpDrv/`, accessed on 4/4/2011.

[24] P. Culley, U. Elzur, R. Recio, S. Bailey, and J. Carrier. Marker PDU Aligned Framing for TCP Specification. Available from `http://tools.ietf.org/id/draft-ietf-rddp-mpa-08.txt`, accessed on 4/4/2011, October 2006.

[25] D. Dalessandro, A. Devulapalli, and P. Wyckoff. Design and Implementation of the iWarp Protocol in Software. In *Parallel and Distributed Computing and Systems (PDCS), 2005*, November 2005.

[26] D. Dalessandro, A. Devulapalli, and P. Wyckoff. iWarp Protocol Kernel Space Software Implementation. In *IPDPS 2006*, April 2006.

[27] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design and Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[28] David Ely, Stefan Savage, and David Wetherall. Alpine: a user-level infrastructure for network protocol development. In *USITS'01: Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems*, pages 15–15, Berkeley, CA, USA, 2001. USENIX Association.

[29] EmuLab. Network Emulation Testbed. Available at `http://www.emulab.net/`, accessed on 4/4/2011.

[30] Adiga et al. An Overview of the BlueGene/L Supercomputer. *Supercomputing 2002*, November 2002.

[31] Bryan Ford. Structured streams: a new transport abstraction. *SIGCOMM Comput. Commun. Rev.*, 37(4):361–372, 2007.

[32] Bryan Ford and Janardhan Iyengar. Breaking up the transport logjam. In *Proc. of workshop on Hot Topics in Networks (HotNets-VII)*, 2008.

[33] I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, G. K. Thiruvathukal, and S. Tuecke. A Wide-Area Implementation of the Message Passing Interface. *Parallel Computing*, 24(12):1735–1749, 1998.

[34] William J. Frawley, Gregory Piatetsky-Shapiro, and Christopher J. Matheus. Knowledge discovery in databases: An overview. *AI Magazine*, 13(3):57–70, 1992.

[35] 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.

[36] B. Goglin. Design and implementation of Open-MX: High-performance message passing over generic Ethernet hardware. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–7, April 2008.

[37] Ryan E. Grant, Mohammad J. Rashti, Pavan Balaji, and Ahmad Afsahi. iWARP Redefined: Scalable Connectionless Communication over High-Speed Ethernet. *In the Proceedings of the 17th International Conference on High Performance Computing (HiPC 2010)*, Dec 2010.

[38] Ryan E. Grant, Mohammad J. Rashti, Pavan Balaji, and Ahmad Afsahi. RDMA Capable iWARP over Datagrams. *25th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011)*, May 2011.

[39] Andreas Grau et al. Efficient and scalable network emulation using adaptive virtual time. *IEEE Conf. on Computer Communications and Networks*, pages 1–6, 2009.

[40] William D. Gropp. Learning from the success of MPI. *Lecture Notes in Computer Science: High Performance Computing (HiPC 2001)*, 2228, 2001.

[41] Higher Speed Study Group. IEEE Group for 100/40 Gigabit Ethernet. Available from `http://www.ieee802.org/3/hssg/`, accessed on 4/4/2011.

[42] Diwaker Gupta et al. Diecast: Testing distributed systems with an accurate scale model. In *In Proc. of NSDI*, pages 407–421, 2008.

[43] Brian Hausauer. iWARP: Reducing Ethernet Overhead in Data Center Designs. Available at `http://www.eetimes.com/design/communications-design/4009333/iWARP-Reducing-Ethernet-Overhead-in-Data-Center-Designs`, accessed on 4/4/2011, November 2004.

[44] IANA. Stream Control Transmission Protocol (SCTP) Parameters - [RFC4960]. Available from `http://www.iana.org/assignments/sctp-parameters`, accessed on 4/4/2011.

[45] IETF. Multipath TCP (mptcp) Charter, 2010. Available at `http://datatracker.ietf.org/wg/mptcp/charter/`, accessed on 4/4/2011.

[46] Intel Corporation. Intel 82599 10 GigE Controller. Available at `http://download.intel.com/design/network/prodbrf/321731.pdf`, accessed on 4/4/2011.

[47] Intel Corporation. PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. Available from `http://download.intel.com/design/network/applnots/321211.pdf`, accessed on 4/4/2011, Dec 2008.

[48] Janardhan Iyengar, Paul Amer, and Randall Stewart. Concurrent Multipath Transfer Using SCTP Multihoming Over Independent End-to-End Paths. *IEEE/ACM Transactions on Networking*, 14(5):951–964, October 2006.

[49] Janardhan Iyengar, Paul Amer, and Randall Stewart. Performance Implications of Receive Buffer Blocking in Concurrent Multipath Transfer. *Computer Communications*, 30(4):818–829, February 2007.

[50] Van Jacobson and Bob Felderman. Network channels. Available from `http://lwn.net/Articles/169961/`, accessed on 4/4/2011, January 2006.

[51] Saqib Jang. Microsoft Chimney: The Answer to TOE Explosion? Available from `http://margallacomm.com/downloads/TOE_Chimney.pdf`, accessed on 4/4/2011, Mar 2008.

[52] Jim Pinkerton Jeff Hilland, Paul Culley and Renato Recio. RDMA Protocol Verbs Specification. Available from `http://www.rdmaconsortium.org/home/draft-hilland-iwarp-verbs-v1.0-RDMAC.pdf`, accessed on 4/4/2011, April 2003.

[53] 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.

[54] 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.

[55] K. Kant. Towards a virtualized data center transport protocol. *Computer Communications Workshops, 2008. INFOCOM. IEEE Conference on*, pages 1–6, April 2008.

[56] K. Kant and N. Jani. SCTP performance in data center environments. In *Proceedings of SPECTS*, July 2005.

[57] Antti Kantee. Environmental Independence: BSD Kernel TCP/IP in Userspace. In *Proceedings of AsiaBSDCon 2009*, 2009.

[58] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

[59] Linux Kernel Stream Control Transmission Protocol (lksctp) project. Available at `http://lksctp.sourceforge.net/`, accessed on 4/4/2011.

[60] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. Available from `http://www.ietf.org/rfc/rfc2018.txt`, accessed on 4/4/2011, October 1996.

[61] 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.

[62] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`, accessed on 4/4/2011, June 1995.

[63] Message Passing Interface Forum. MPI 3.0 Working Group. `http://meetings.mpi-forum.org/MPI_3.0_main_page.php`, accessed on 4/4/2011, November 2010.

[64] David S. Miller. How SKBs work. 2011. Available at `http://vger.kernel.org/~davem/skb.html`, accessed on 4/4/2011.

[65] Nader Mohamed. Self-configuring communication middleware model for multiple network interfaces. In *COMPSAC (1)*, pages 115–120, 2005.

[66] Nader Mohamed, Jameela Al-Jaroodi, Hong Jiang, and David R. Swanson. High-performance message striping over reliable transport protocols. *The Journal of Supercomputing*, 38(3):261–278, 2006.

[67] J. Nagle. Congestion Control in IP/TCP Internetworks. *RFC 896*, January 1984.

[68] John Naisbitt and Patricia Aburdene. *Megatrends : ten new directions transforming our lives*. Warner Books, New York :, 1982.

[69] NASA Ames Research Center. Numerical aerodynamic simulation (NAS) parallel benchmark (NPB) benchmarks. Available from `http://www.nas.nasa.gov/Software/NPB/`, accessed on 4/4/2011.

[70] Preethi Natarajan, Nasif Ekiz, Paul D. Amer, Janardhan Iyengar, and Randall Stewart. Concurrent multipath transfer using SCTP multihoming: Introducing the potentially-failed destination state. In *Networking*, pages 727–734, 2008.

[71] Preethi Natarajan, Janardhan Iyengar, Paul D. Amer, and Randall Stewart. Concurrent multipath transfer using transport layer multihoming: Performance under network failures. In *MILCOM*, Washington, DC, USA, October 2006.

[72] A. Nunez, J. Fernandez, J.D. Garcia, and J. Carretero. New techniques for simulating high performance mpi applications on large storage networks. pages 444–452, 29 2008-Oct. 1 2008.

[73] Ohio State University. OSU MPI Benchmarks, 2010. Available from `http://mvapich.cse.ohio-state.edu/benchmarks`, accessed on 4/4/2011.

[74] Takayuki Okamoto, Shin'ichi Miura, Taisuke Boku, Mitsuhisa Sato, and Daisuke Takahashi. RI2N/UDP: High bandwidth and fault-tolerant network for PC-cluster

based on multi-link Ethernet. In *21st International Parallel and Distributed Processing Symposium (IPDPS'07)*, Long Beach, California, March 2007. Workshop on Communication Architecture for Clusters.

[75] Omni-TI-Labs. Libumem project website. 2010. Available at `https://labs.omniti.com/trac/portableumem/`, accessed on 4/4/2011.

[76] Kumiko Ono and Henning Schulzrinne. The Impact of SCTP on SIP Server Scalability and Performance. In *GLOBECOM*, pages 1421–1425. IEEE, 2008.

[77] OpenFabric Alliance. Available from `http://www.openfabrics.org/`, accessed on 4/4/2011.

[78] Keshab K. Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation*. John Wiley and Sons, 1999.

[79] Penguin Computing and Scyld Software. The Beowulf Cluster Site. Available at `http://www.beowulf.org/`, accessed on 4/4/2011.

[80] Brad Penoff, Humaira Kamal, Alan Wagner, Mike Tsai, Karol Mroz, and Janardhan Iyengar. Employing transport layer multi-railing in cluster networks. *J. Parallel Distrib. Comput.*, 70(3):259–269, 2010.

[81] Brad Penoff, Mike Tsai, Janardhan Iyengar, and Alan Wagner. Using CMT in SCTP-based MPI to exploit multiple interfaces in cluster nodes. In *Proceedings, 14th European PVM/MPI Users' Group Meeting*, Paris, France, September 2007.

[82] 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.

[83] Brad Penoff and Alan Wagner. High performance computing using commodity hardware and software. In Victor Leung, Eduardo Parente Ribeiro, Alan Wagner, and Janardhan Iyengar, editors, *Multihomed Communications for Heterogeneous Networks: Concepts and Applications of SCTP Multihoming*. Wiley, 2011.

[84] Brad Penoff, Alan Wagner, Irene Rüngeler, and Michael Tüxen. MPI-NeTSim: A network simulation module for MPI. In *The Fifteenth International Conference on Parallel and Distributed Systems (ICPADS'09)*, 2009.

[85] S. Pope and D. Riddoch. End of the Road for TCP Offload. Technical report, Solarflare, April 2007.

[86] Prashant Pradhan, Srikanth Kandula, Wen Xu, Anees Shaikh, and Erich Nahum. Daytona: A User-Level TCP Stack. Available from `http://nms.csail.mit.edu/~kandula/data/daytona.pdf`, accessed on 4/4/2011.

[87] S. Prakash and R.L. Bagrodia. MPI-SIM: using parallel simulation to evaluate MPI programs. *Winter Simulation Conference*, 1:467–474, 1998.

[88] Kyle Rankin. Bond, Ethernet Bond. *Linux J.*, 2011, January 2011.

[89] RDMA Consortium. Available at `http://www.rdmaconsortium.org/`, accessed on 4/4/2011.

[90] R. Recio, P. Culley, D. Garcia, J. Hilland, and B. Metzler. An RDMA protocol specification. Available from `http://tools.ietf.org/id/draft-ietf-rddp-mpa-08.txt`, accessed on 4/4/2011, April 2005.

[91] R. Riesen. A hybrid MPI simulator. In *IEEE Cluster Computing*, pages 1–9, Sept. 2006.

[92] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, 1997.

[93] I. Rüngeler, M. Tüxen, and E.P. Rathgeb. Integration of SCTP in the OMNeT++ simulation environment. In *Proc. of OMNeT++'08*. ICST, Brussels, Belgium, 2008.

[94] Irene Rüngeler, Brad Penoff, Michael Tüxen, and Alan Wagner. A New Fast Algorithm for Connecting the INET Simulation Framework to Applications in Real-time. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques (SIMUTools2011)*, 2011.

[95] Irene Rüngeler, Michael Tüxen, and Erwin P. Rathgeb. Improving the acknowledgment handling of SCTP. In *4th International Conference on Digital Society (ICDS 2010)*, February 2010.

[96] Hemal Shah, James Pinkerton, Renato Recio, and Paul Culley. Direct data placement over reliable transports. Available from `http://tools.ietf.org/html/rfc5041`, accessed on 4/4/2011, October 2007.

[97] Dan L. Siemon. The IP Per Process Model: Bringing End-to-end Network Connectivity to Applications. Master's thesis, University of Western Ontario, London, Ontario, Canada, 2007.

[98] Solarflare Communications. OpenOnload Application Accelerator. Available at `http://www.solarflare.com`, accessed on 4/4/2011.

[99] R. Stewart, K. Poon, M. Tuexen, V. Yasevich, and P. Lei. Sockets API Extensions for SCTP. Available from `http://tools.ietf.org/html/draft-ietf-tsvwg-sctpsocket-27`, accessed on 4/4/2011, Mar 2011.

[100] 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 Proto-

col (SCTP). Available from `http://www.ietf.org/rfc/rfc2960.txt`, accessed on 4/4/2011, October 2000.

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

[102] J. Stone, R. Stewart, and D. Otis. Stream Control Transmission Protocol (SCTP) Checksum Change. Available from `http://www.ietf.org/rfc/rfc3309.txt`, accessed on 4/4/2011, September 2002.

[103] Jonathan Stone and Craig Partridge. When the CRC and TCP checksum disagree. In *SIGCOMM*, pages 309–319, 2000.

[104] Yar Tikhiy. FreeBSD Manual Page: mbuf. 2008. Available at `http://www.freebsd.org/docs/man.html`, accessed on 4/4/2011.

[105] Top500. Top 500 Supercomputing Sites. Available from `http://www.top500.org`, accessed on 4/4/2011.

[106] Mike Tsai, Brad Penoff, and Alan Wagner. A hybrid MPI design using SCTP and iWARP. In *Communication Architecture for Clusters: Proceedings of the 2008 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2008.

[107] M. Tuexen and R. Stewart. UDP Encapsulation of SCTP Packets. Available at `http://tools.ietf.org/html/draft-tuexen-sctp-udp-encaps-06`, accessed on 4/4/2011, Jan 2011.

[108] Michael Tüxen. An SCTP network kernel extension for Mac OS X. 2010. Available at `http://sctp.fh-muenster.de/sctp-nke.html`, accessed on 4/4/2011.

[109] Michael Tüxen and Thomas Dreibholz. The sctplib Userspace SCTP Implementation. 2009. Available at `http://www.sctp.de/sctp-download.html`, accessed on 4/4/2011.

[110] Michael Tüxen, Irene Rüngeler, and Erwin P. Rathgeb. Interface connecting the INET simulation framework with the real world. In *Simutools '08: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, pages 1–6, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[111] Michael Tüxen, Irene Rüngeler, and Randall Stewart. SACK-IMMEDIATELY extension for the Stream Control Transmission Protocol, draft-tuexen-tsvwg-sctp-sack-immediately-05 . *IETF*, January 2011.

[112] Mustafa Uysal, Anurag Acharya, Robert Bennett, and Joel Saltz. A customizable simulator for workstation networks. In *Proc. of the IPPS*, pages 249–254, 1996.

[113] A. Varga et al. INET Framework. 2010. Available at `http://github.com/inet-framework/inet-doc`, accessed on 4/4/2011.

[114] A. Varga et al. OMNeT++ 4.1 Docs. 2010. Available at `http://www.omnetpp.org/documentation`, accessed on 4/4/2011.

[115] Andras Varga. The OMNeT++ Discrete Event Simulation System. In *In the Proceedings of the European Simulation Multiconference*, 2001.

[116] Abhinav Vishnu, Prachi Gupta, Amith R. Mamidala, and Dhabaleswar K. Panda. Scalable systems software - a software based approach for providing network fault tolerance in clusters with uDAPL interface: MPI level design and performance evaluation. In *Proceedings of Supercomputing 2006*, page 85.

[117] D. Wing, A. Yourtchenko, and P. Natarajan. Happy Eyeballs: Trending Towards Success (IPv6 and SCTP). Available at `http://tools.ietf.org/id/draft-wing-http-new-tech-01.html`, accessed on 4/4/2011, Aug 2010.

[118] Wireshark protocol analyzer.
Available at `http://www.wireshark.org`, accessed on 4/4/2011.

[119] J. Yoakum and L. Ong. An introduction to the Stream Control Transmission Protocol (SCTP). Available from `http://www.ietf.org/rfc/rfc3286.txt`, accessed on 4/4/2011, May 2002.