Hybrid Design of MPI over SCTP

by

Mike Yao Chen Tsai

B.Sc., The University of British Columbia, 2005

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Master of Science
in
The Faculty of Graduate Studies
(Computer Science)

The University Of British Columbia
September, 2007
© Mike Yao Chen Tsai 2007
Abstract

Message Passing Interface (MPI) is a popular message passing interface for writing parallel applications. It has been designed to run over many different types of network interconnects ranging from commodity Ethernet to more specialized hardwares including: shared memory, and Remote Direct Memory Access (RDMA) devices such as InfiniBand and the recently standardized Internet Wide Area RDMA Protocol (iWARP). The API itself provides both the point-to-point and remote memory access (RMA) operations to the application. However, it is often implemented based on one kind of underlying network device, namely entirely RDMA or point-to-point. As a result, it is often not possible to provide a direct mapping from the software semantics to the underlying hardware. In this work, we propose a hybrid approach in designing MPI in which network device to use can depend on its functional requirement. This allows the MPI API to exploit the potential performance benefits of the underlying hardware more directly. Another highlight of this work is the design of the MPI middleware to be IP based in order to provide support for both cluster and wide area network environment; this can be achieved via the use of a commodity transport layer protocol, namely Stream Control Transmission Protocol (SCTP). We will demonstrate how SCTP can be used to support MPI with different kinds of network devices and to provide multirailing support from the transport layer.
Contents

Abstract ................................................................. ii

Contents ............................................................... iii

List of Tables ........................................................ vi

List of Figures ........................................................ vii

Acknowledgements ..................................................... ix

1 Introduction .......................................................... 1
  1.1 Motivation ....................................................... 1
  1.2 Contribution .................................................... 4
  1.3 Thesis outline ................................................... 5

2 Background .......................................................... 6
  2.1 Message Passing Interface ...................................... 6
  2.2 Stream Control Transmission Protocol ......................... 9
  2.3 Internet Wide Area RDMA Protocol .......................... 11

3 Design and Implementation ........................................ 16
  3.1 Overview ........................................................ 16
  3.2 MPICH2-SCTP .................................................. 16
    3.2.1 Dynamic Connection and Initialization .................. 17
## Contents

3.2.2 Message de-multiplexing and SCTP events .......................... 19
3.2.3 Progress Engine .................................................. 21
3.2.4 Multihoming ...................................................... 22
3.2.5 Process Spawning .................................................. 23
3.2.6 Limitations ......................................................... 24

3.3 iWARP Implementation ................................................. 28
  3.3.1 Direct Data Placement (DDP) over SCTP ......................... 29
  3.3.2 DDP header ..................................................... 30
  3.3.3 DDP to SCTP semantic mapping ................................ 30
  3.3.4 Asynchronous support ........................................... 31
  3.3.5 User level API, Verbs ............................................ 33
  3.3.6 Limitations ..................................................... 35

3.4 MPICH2-Hybrid ....................................................... 36
  3.4.1 Memory Management ............................................. 36
  3.4.2 Connection Management .......................................... 37
  3.4.3 Progress Engine Extension ...................................... 41
  3.4.4 MPI-2 RMA semantic mapping ................................... 42
  3.4.5 Limitations ..................................................... 48

4 Evaluation ............................................................ 50
  4.1 Experiment setup .................................................. 50
  4.2 MPI_Put/Get performance .......................................... 51
  4.3 CMT Related Experiments ......................................... 54
  4.4 General Benchmarks ................................................ 58
    4.4.1 Microbenchmarks .............................................. 59
    4.4.2 Real applications ............................................. 60
  4.5 Summary .......................................................... 63
## Contents

5  Related Work .................................................. 64
   5.1 Hybrid Design Related .................................. 64
   5.2 CMT related .................................................. 66
   5.3 RDMA related .................................................. 67

6  Future work and Conclusion ................................. 71

Bibliography ....................................................... 73
List of Tables

3.1 OSC-iWARP Verbs (Function calls extended from OSC-iWARP are underlined) ........................................ 34

3.2 MPI-2 RMA function prototypes .................................................. 43
List of Figures

1.1 Hybrid MPI design over SCTP ......................... 3
2.1 MPI middleware structured view ..................... 7
2.2 iWARP stack layout ................................. 12
3.1 MPICH2 Implementation Structure .................. 17
3.2 Object/Component view of ch3:sctp .................. 18
3.3 MPI and SCTP semantic mapping ..................... 20
3.4 Progress Engine design ............................. 22
3.5 MPLANY_TAG issue ................................. 27
3.6 iWARP semantics and modules ...................... 33
3.7 Component view of ch3:hybrid ....................... 37
3.8 Static connect all routine .......................... 40
3.9 Dynamic connection setup for QPs ................... 41
3.10 MPL_Put via RDMA_Write during fence call ......... 45
3.11 MPL_Get via RDMA_Write during fence call ......... 46
3.12 MPL_Get via RDMA_Read during fence call ........... 47
4.1 Cluster Configuration ............................... 51
4.2 MPL_Put/MPL_Get latency test ..................... 52
4.3 MPI-1/2 Synthetic test ............................. 54
4.4 Iperf SCTP test .................................... 56
List of Figures

4.5 OSU Bandwidth test ........................................... 57
4.6 OSU Latency test (Less is better) ............................ 60
4.7 OSU Bi-directional test (More is better) .................. 61
4.8 NAS Parallel MPI Benchmarks (Less is better) .......... 62
5.1 Layer view of SDP ............................................... 68
5.2 TCP Latency test with NetPIPE (Less is better) ........ 69
I would like to express sincere gratitude to my supervisor, Dr. Alan Wagner. His encouragement, insight, and faith in my abilities are immensely appreciated. Without his guidance and ideas, I would not have completed this program.

I would like to dedicate this thesis to my family and Jiemin for their unquestioning love and support.

I am also grateful to Dr. Norm Hutchinson for being the second reader and providing his insight in making this thesis more complete. Many thanks to Randall Stewart, one of the core designers of SCTP, for technical support of the stack.

Last but not the least, I would like to thank Dennis Cheong for his support and the gang in the Distributed Systems Group for all the fun times: Brad Penoff, Kan Cai, Camilo Rostoker, Brendan Cully, Gang Peng, Le Cuong and other buddies.

Mike Yao Chen Tsai

The University of British Columbia

September 2007
Chapter 1

Introduction

1.1 Motivation

Network components are critical to any distributed systems in both performance and reliability. Traditional network fabrics such as Ethernet are widely available in every computing environment, however, these devices have been criticized for the lack of performance scaling as network speed increases. As a result, various technologies have been proposed such as TCP offload engines (TOE) and RDMA (Remote Direct Memory Access) to enable transport protocol offloading and zero-copy respectively. Although RDMA based devices such as InfiniBand can provide low latency and high bandwidth, it does not support the traditional IP framework and, in comparison to Ethernet, it is still relatively expensive. There are many different network interconnects available, each with its own pros and cons. I believe that the application should be able to decide on the appropriate network devices to support with respect to its functional requirements and hence increase flexibility.

Currently, there are many network technologies that can provide low latency and high bandwidth. These include InfiniBand [8], Myrinet [1], and Quadrics [3]. The per port cost of these devices is often fairly expensive and it depends on the applications whether such high-end devices are necessary. The argument against RDMA capable devices has been the application semantic mapping [21]. RDMA requires memory to be pinned in order for the receive operation to take
place and pin and unpin are relatively expensive operations as it requires the host operating system to intervene. As a result, applications that seek gains in performance need to be modified and redesigned in order to exploit the benefits of RDMA devices.

Ethernet, on the other hand, is relatively cheap and common in both wide area networks and major cluster environments. Although its performance is not as competitive as others, there is a large base of legacy applications that rely on the traditional socket interface and the existing Ethernet infrastructure that has been deployed for many years. However, Ethernet has been highly criticized for its high latency and it is not suitable for applications that require low latency support in a cluster environment. Some of the latency cost can be minimized by designing the software in a task farm structure that allows communication to overlap with computation in order to hide some of the latency; this is shown to be particularly useful in a wide area network in which network conditions can fluctuate [28]. There are also technologies that can help improve the performance of traditional Ethernet devices such as SolarFlare's Ethernet Accelerator [5]. This is an interesting approach as it supports the Berkeley socket interface and provides the performance improvement transparently over the IP network without any modification to legacy applications. Hence, we believe that Ethernet still has its place in the future as there has been proposals in building 40/100 GbE (Gigabit Ethernet) devices [7]. With its large deployment base over other network technologies, it may remain the dominant network interconnect for years to come.

MPI [18] is a popular message passing interface for writing parallel applications and it supports a variety of network interconnects ranging from shared memory to Ethernet to RDMA capable devices. In addition to the hardware sup-

\footnote{In the latest rankings of the Top 500 supercomputers, 211 of the systems use GbE as the internal network [50].}
port, MPI specification [18] has two notions for communication: point-to-point (MPI-1) and remote memory access (MPI-2) operations. The MPI-2 specification was later defined to clarify some of the MPI-1 issues and to propose extensions of the MPI programming model substantially beyond the strict message-passing model represented by MPI-1. While we are focusing on the communication aspect of MPI, MPI-2 introduced remote memory access (one-sided) operations and these specifications are meant to be hardware independent. In most middleware implementations of MPI-1/2, the MPI-1/2 communication functions are usually implemented entirely on top of point-to-point or RDMA hardware regardless of the specification. In this work, we are proposing a hybrid design of MPI based on its functional requirements. We believe by providing a more direct mapping from the MPI primitives to the underlying network device, this can increase the flexibility of middleware and allow the application to better exploit the hardware. In the end, all of the necessary changes are transparent to the end user.

![Hybrid MPI design over SCTP](image)

**Figure 1.1: Hybrid MPI design over SCTP**

Our prototype implementation of MPI middleware supports both socket based calls for point-to-point and RDMA based semantics via iWARP (Inter-
net Wide Area RDMA Protocol), which is a set of protocol specifications for RDMA over IP. Both semantics are using Stream Control Transmission Protocol (SCTP) as the underlying transport layer protocol to provide reliable message transferring and the use of commodity Ethernet devices for both local and wide area networks as shown in Figure 1.1.

SCTP was considered because it is message-based and it simplifies the implementation of point-to-point as well as providing a solid foundation for iWARP. Namely, it avoids the message framing problem that occurs when iWARP is layered over TCP. Furthermore, our previous work has shown that SCTP is able to adapt to fluctuating network conditions (varying latency and loss) better than TCP [28] and we will be demonstrating how various configurations of SCTP can be supported in our implementation to exploit performance improvement via multirailing in a cluster environment. Due to lack of hardware support, the iWARP stack is implemented in software [16, 38] and the hardware device itself is simulated. As a result, performance improvement of the simulated RDMA device is not the main focus of this work. Moreover, we will describe how a software based iWARP stack can be used with certain Ethernet accelerators to provide low latency and can also be used to communicate with native iWARP based hardware.

1.2 Contribution

The main contribution of this work is the exploration of issues that arise in a hybrid design to support MPI-1 and MPI-2 where a functional decomposition is implemented such that the point-to-point and one-sided primitives use the best available device. We take advantage of SCTP as it provides a common layer for both point-to-point and iWARP. The RDMA device is based on Ohio Supercomputing Center's (OSC) software implementation of iWARP. The OSC's iWARP
software stack [16] only supports TCP/IP and we extend this implementation to use SCTP as there is currently no such implementation that we are aware of. Previous work by our group had constructed an MPICH2 implementation of SCTP and this was the basis for the hybrid design extension.

1.3 Thesis outline

The remainder of the thesis is organized as follows. In Chapter 2, we provide some background information about the higher level components that are being addressed by this work, namely MPI, SCTP and iWARP. In Chapter 3, we discuss the design of each of the main components in our hybrid design and their interactions. In Chapter 4, we evaluate the design and compare the hybrid extension to its parent design, the SCTP channel. In Chapter 5, we present some of the related work to this project. Lastly, we discuss possible future extensions to our design and finally conclude in Chapter 6.

\footnote{MPICH2 is a popular open source implementation of MPI Middleware and SCTP is implemented within MPICH2 as a communication module. It is also part of the official release of the MPICH2: http://www-unix.mcs.anl.gov/mpi/mpich/and has been thoroughly tested.}
Chapter 2

Background

In this chapter, we provide some background information about the main technologies that were used in the design and implementation of our system.

2.1 Message Passing Interface

Message Passing Interface (MPI) is a popular API for writing parallel applications for distributed systems. Its popularity is based on its transparent support on a variety of network devices and languages binding for C/C++ and Fortran. It is primarily used in writing scientific computing code that can run on a variety of different clusters ranging from small to large supercomputers. MPI typically consists of a middleware library, various helper commands to compile and run MPI-specific programs and a runtime daemon to create and teardown the MPI universe. From a component point of view, MPI consists of a process scheduler, memory manager, job scheduler and communication modules (Figure 2.1). Often MPI middleware is modularized to glue more flexibilities in the choice of hardware and software.

The MPI daemon acts as an administrator that runs in the background and is responsible for managing the runtime environment. Each endpoint has an MPI daemon attached to it and the daemon itself communicates with others to coordinate its activities, such as resource discovery. In addition, the daemon also serves as a database to store the information about each MPI process and
Chapter 2. Background

this information is kept globally consistent across the MPI universe.

The MPI library comprises of both MPI-1 and MPI-2 specifications. The MPI-1 library contained over 200 API calls\(^1\) and a variety of communication primitives including the basic send/receive operations both synchronously and asynchronously. MPI allows asynchronous communication via nonblocking primitives to overlap computation with communication, hence helping to improve the latency tolerance of the application. Aside from point-to-point communication, MPI also provides mechanisms for collective communication for a group of MPI processes such as broadcast, reduce, gather, and all-to-all. The MPI-2 specification was later introduced primarily to clarify issues in MPI-1 and to provide three extensions in the areas of remote memory access (RMA) operations, dynamic process spawning, and parallel I/O.

From the MPI specification, each MPI process is assigned to a globally unique rank (from 0 to \(N-1\), where \(N\) is the size of the MPI universe) and the rank is used to address an MPI peer with respect to the participants of the parallel application. A typical MPI message has an envelope attached that con-

\(^1\)MPI-1 specification: http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html
tains the metadata of the message. The metadata includes the type of message, source and target rank, and a tag value. In addition, for point-to-point operations, each message is also designated by an opaque value that serves an MPI level context ID. An MPI level context ID is used to separate collections of MPI messages. In particular, the context ID is used to differentiate MPI primitives from each other. For instance, MPI collective messages should not be mixed with simple point-to-point communication. By including a context value, MPI middleware can ensure that MPI messages from different primitives are matched correctly. Message matching is performed at the TRC level (Tag, Rank, and Context) and not every operation requires the use of a tag (e.g., collectives and one-sided operations). TRC information is used in both sending and receiving.

In addition, MPI allows the use of wildcards in posting a receive call to match from any source (MPI.ANY.SOURCE) and any tag (MPI.ANY.TAG). MPI also provides the abstraction for process group and communicators that allows the user application to parallelize their problem domain in a more descriptive way. Overall, the MPI specification exports a comprehensive set of abstractions to the users and by using a common specification, this can improve the portability of MPI programs across different MPI middlewares.

Internally, MPI defines the notion of a short/long and expected/unexpected messages. Short messages are transmitted immediately while long messages (larger than a threshold value) require the use of a rendezvous protocol to prevent the sending side from overflowing the receiver's buffer. An expected message is one for which the receiver is waiting ("posted") and an unexpected message is an MPI message that was not able to be matched and thus will be buffered.

Recall that MPI provides asynchronous transfer of messages and in order to ensure a nonblocking send or receive request is completed, user applications need
Chapter 2. Background

to query the MPI middleware. Moreover, with the introduction of MPI-2's dynamic process support, MPI processes can be created and join the MPI universe dynamically. As a result, within the MPI middleware, messages and endpoints are defined as MPI level abstractions that need to be progressed and maintained by the Message Progression Layer, or a progress engine. In general, the progress engine is responsible for managing the life-cycle of a message and the connection states of MPI peers. The life-cycle of a message includes: performing message matching for incoming messages, handling unexpected messages, and transmitting pending send requests (short and long messages).

The MPI specification does not mandate any design decisions for the middleware other than the implementation requirements for developers. As a result, there are many popular open source MPI middleware implementations available: LAM/MPI [19], MPICH2 [52], Open MPI [20]. Most MPI implementations support TCP/IP since Ethernet is widely used and available in every cluster. Aside from TCP/IP, there is also support for shared memory, Myrinet, InfiniBand and iWARP. In general, every large parallel machine supports the MPI library.

With the strong growth and availability of multi-core processors, we believe that parallel computing will become more critical than ever and MPI will continue to evolve through research and provide the tools that can ease the development of parallel applications.

2.2 Stream Control Transmission Protocol

Stream Control Transmission Protocol (SCTP) is a new reliable transport layer protocol that has recently been standardized by the IETF [48]. It was originally designed for the telephony community. Unlike TCP, it is message oriented and is similar to UDP but without message truncation. Aside from the TCP-like features such as congestion control, it provides additional features: multistreaming,
multihoming, security enhancements and improved extensibility. SCTP maps a set of IP and Port pairs to an association. Within an association, multiple streams can be supported. A stream is an uni-directional flow of data and data ordering is only preserved within a given stream. This is analogous to opening multiple TCP connections except streams share common flow and congestion control. Moreover, by assigning data packets to different streams, this can overcome the potential head-of-line problem found in TCP [47].

Multihoming is a major feature of SCTP that is not in TCP, and provides additional fault tolerance for user applications. Multihoming allows an application to bind to multiple network interfaces and switch from one to another upon network failures. For maximum fault tolerance, independent links are preferred so that paths do not share a common bottleneck. There is a recent project from the University of Delaware to extend the multihoming feature to support Concurrent Multipath Transfer (CMT), which inherits the fault tolerance aspect of multihoming and provides additional bandwidth by striping data across all available links at the transport layer [24].

SCTP was introduced with backward compatibility in mind. As a result, both UDP and TCP style sockets are supported through the use of the Berkeley socket API. With respect to UDP-style sockets, SCTP is able to provide a reliable message oriented service. In the underlying design, UDP and TCP style sockets are compatible with each other as everything is mapped to association IDs. In addition to the Berkeley Socket API, SCTP introduced another set of APIs to support its new features. For instance, there is a new sctp_bindx call that allows a user to selectively bind to specific network interfaces. Furthermore, there are multiple extended send and receive calls that allow the user to specify or retrieve more SCTP related options or information.

TCP was found to have various security loopholes, for example; during the
connection establishment and handshake process where SYN-flood attacks can be made. As a result, SCTP incorporates a cookie mechanism during initialization with a four-way handshake and a 32-bit sequence number to prevent such denial of service attack. Moreover, SCTP uses a stronger checksum, CRC32c, to ensure end to end reliability and data integrity. The stronger checksum requires more computing cycles and it is often not available for offload to present-day hardware.

SCTP is also more resilient to packet loss as it uses Selective Acknowledgment (SACK). This is available in TCP as well, but with a smaller number of SACK blocks. This feature is useful where loss is likely such as Wide Area and Satellite networks. With regards to extensibility, SCTP defines a notion of chunk within each SCTP datagram and it is possible for user to define their own chunks. For instance, there are proposals for SCTP to support partial reliability of the messages.

Overall, SCTP is gradually gaining popularity as many application layer protocols have been ported such as FTP [30], HTTP [44] to support the new transport layer protocol. In this work, we are focusing on supporting SCTP within MPI and iWARP with proper mappings to exploit the benefits of the protocol.

2.3 Internet Wide Area RDMA Protocol

Internet Wide Area RDMA Protocol (iWARP) is a set of protocol specifications to provide RDMA support for wide area networks and it was designed to operate on top of the existing IP infrastructure to overcome the performance deficiency of traditional Ethernet based network devices. It is an update from the IETF of the RDMA Consortium’s RDMA over TCP standard [4]. In order to improve the performance, both RDMA and OS-bypass are necessary in order
to move data without the CPU or the operating system being involved. Most RDMA-capable network adapters also handle protocol offloading (TOE) and this allows the NIC to do the network processing locally. Hence, the host CPU can continue application processing concurrently while the network adapters handle the majority of the network traffic, thus improving the overall efficiency.

![iWARP stack layout](image)

**Figure 2.2: iWARP stack layout**

iWARP is combination of the following layers (Figure 2.2):

- **Verbs API** The verbs or API layer exposes the RDMA operations to the user application [26]. Furthermore, it also provides direct access to the RDMA capable Network Interface Card (RNIC) hardware. The verbs layer communicates directly with the RDMAP layer. iWARP does not provide any verbs specification. 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.

- **Remote Direct Memory Access Protocol (RDMAP):** The RDMAP layer provides both traditional send/receive and read/write services to the user applications [10, 42]. RDMAP itself is a fairly thin layer in which it depends on the lower layer protocol to provide the corresponding service. In this case, the lower layer is the Direct Data Placement layer.
Chapter 2. Background

• **Direct Data Placement (DDP):** DDP layer is responsible for placing the data in the proper place after being received [23]. 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 upcoming 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 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 a reliable delivery from the lower layer protocol.

• **Marker PDU Aligned (MPA)-TCP/SCTP:** Since DDP is an end-to-end protocol, the intermediate nodes do not necessarily support DDP. As a result, 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 to fail. To overcome this challenge, Marker PDU Aligned or MPA was introduced to provide message framing capability for TCP/IP serving as an extension to TCP/IP in order to maintain backward compatibility [40]. Its 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 segment. Each MPA segment is known as the Framing Protocol Data Unit, FPDU. In addition, the MPA layer adds a more robust checksum, CRC32, to each FPDU, to preserve data integrity. During initialization,
active/passive 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 [39]. Moreover, along with the added checksum calculation, the MPA layer can add a considerable overhead to the overall performance.

Alternatively, SCTP is a candidate to replace MPA-TCP as the SCTP stack provides reliable message transfer, stronger checksum and message framing support natively. With SCTP, it is possible to schedule messages across multiple streams and out-of-order if necessary. Recall that SCTP uses SACK to inform data gaps during transmission and this is particularly useful in the case of multitrailing in which out-of-order arrival of messages is inevitable. In this work, we are using SCTP as the lower layer protocol for DDP to exploit some of the possible benefits of this newly standardized transport layer protocol.

Since iWARP is based on a layered design, there are various options to provide iWARP support. For instance, one can implement all or part of the software iWARP stack that runs in user space or in the host operating system. Although this approach may not provide any performance improvement, it serves as a good rollout strategy as there are only a few iWARP capable devices available on the market. In addition, the software approach allows iWARP to be hardware independent as it will run on any Ethernet capable device and it can be easily patched should the specification change. One other interesting approach is to run the software iWARP on top of an Ethernet Accelerator; this will enable low latency and high bandwidth support. Ultimately, from the performance perspective, iWARP was designed to run on RDMA capable devices.
(RNIC)$^2$ with protocol offloading in order to provide true zero-copy and OS-bypass. Nonetheless, all these implementation options allow more flexibility in future deployment and support.

\[\text{NetEffect 10Gbps iWARP Ethernet Channel Adapter: } \text{http://www.neteffect.com/products.html}\]
Chapter 3

Design and Implementation

3.1 Overview

In this chapter, we introduce the design and implementation of various components in our prototype and describe how they interact with each other.

3.2 MPICH2-SCTP

We have implemented SCTP as a Channel-3 (CH3) device in MPICH2, which is a direct extension from our previous work with LAM/MPI [29]. MPICH2 was developed at Argonne National Laboratory and it is one of the most popular MPI implementations. MPICH2 is aimed at supporting both MPI-1 and MPI-2 features and implementing a new modular design to provide more performance, portability and flexibility.

Conceptually, a CH3 layer implements only a subset of functions from a higher level abstract module called an Abstract Device Interface 3 (ADI3, third generation design) (Figure 3.1). Consequently, a channel implements the CH3 interface and there are existing channels for most communication architectures including TCP, and shared memory (SHMEM). Since the CH3 interface only exposes a fairly small set of primitives, it is easier to implement a CH3 channel than a complete ADI3 device. Internally, a channel device should provide a progress engine to monitor many states between MPI endpoints and asyn-
chronous support for message transport.

![Figure 3.1: MPICH2 Implementation Structure](image)

Our CH3 implementation is based on the point-to-point paradigm and uses the SCTP one-to-many (SCTP-UDP) style socket. In the following subsections, we will discuss various implementation highlights and design tradeoffs during our development cycle. The CH3 device for SCTP and the Hybrid device are denoted as ch3:sctp and ch3:hybrid respectively.

### 3.2.1 Dynamic Connection and Initialization

In this section, we discuss the aspects related to setting the connections for MPI to use SCTP. One goal of our one-to-many implementation is to minimize the number of file descriptors that need to be opened and managed for each MPI process and this can be accomplished by using a UDP (one-to-many) style socket. In the one-to-many design like UDP, it is possible to open up one socket for each MPI process. In addition, it is possible to allow for "lazy connect" where connections are not set up until they are needed. Thus, with dynamic connection, we can dramatically speed up initialization time and allow each MPI
process to start computing as soon as possible.

During initialization, each MPI process opens a single SCTP-UDP style socket for both connection and message transfer purposes. This socket information is saved as a MPICH2 specific business card object\(^1\) and committed to the MPICH2 Daemon. As a result, MPI processes that wish to set up a connection can query the MPICH2 daemon to lookup the specific information, such as the port of the designated process' SCTP-UDP style socket. We use the UDP model to piggyback application level messages with the connection setup process.

\[\text{Figure 3.2: Object/Component view of ch3:sctp}\]

Within MPICH2, each endpoint is encapsulated as a Virtual Connection object (VC) and these objects are pre-allocated but are not connected during initialization. Each VC stores the target endpoint's state and designated MPI Send or Receive message queues on a per SCTP stream basis (Figure 3.2). Each endpoint can have multiple SCTP streams for Send/Receive and each SCTP stream represents a separate connection. The number of streams is adjustable during compilation and we reserve stream 0 to be used for control information transmission when necessary. When one of the streams is set to connected,\(^1\) a business card stores the identity of each MPI process for resource discovery.
the VC state is also set to connected. Our current implementation adjusts the endpoint's specific stream state upon a receive.

Within MPICH2, there is a special type of message called the connection packet. The connection packet contains information of the source endpoint's hostname, process group ID and rank. In order to initiate a connection, one is required to send a connection packet to the specific stream before transmitting any other MPI messages. Upon receiving the connection packet, the endpoint can use the included data to locate all the proper target endpoint's stream states and update them correctly.

The disadvantage of our current implementation is that the connection packet is not necessarily small (it ranges from 40-70 bytes). Moreover, we are sending them on a per stream basis due to the fact that SCTP provides full ordering within a stream but not across streams and we need to guarantee that the connection packet is the first message the target endpoint receives for a particular stream. There are certainly optimizations to avoid sending a connection packet for every stream. For example, one could set a threshold value that limits the number of initialized streams. This would allow the sender to stop sending connection packets for uninitialized streams once it hit the threshold limit.

We believe that with dynamic connections, hardware and software resources can be managed more efficiently. With the use of SCTP-UDP style sockets, the connection process is also simplified because the middleware can manage less states without explicitly calling connect and accept.

3.2.2 Message de-multiplexing and SCTP events

Message de-multiplexing needed to be redesigned, since each MPI process only has one global SCTP-UDP style socket for all general purpose communication. Each endpoint is now mapped to an SCTP association that is managed by the
transport layer. Since an endpoint is represented as a Virtual Connection object in MPICH2, a hash table was used to map SCTP associations to their corresponding VC. In order to utilize multiple SCTP streams within an association, we assigned streams with respect to the context and tag of an MPI message (Figure 3.3). This is a straightforward mapping that obeys the MPI specification since message ordering is only guaranteed for messages of the same tag and context value. At the same time, we export a function hook to allow a user defined stream assignment procedure. Message de-multiplexing needs to be mapped with a pair of {association-ID, SCTP stream number} and the message is also enqueued into a specific queue for a stream belonging to a particular VC.

In order to retrieve the SCTP stack level information for message de-multiplexing, it is necessary to register for interested events with the SCTP stack via setting socket options. The only metadata that we are currently interested in is the association and stream value of each incoming message. The SCTP stack populates this metadata structure upon performing a receive.

Our current implementation statically allocates a message queue per stream resulting in higher memory consumption than needed since not all of the streams are used during runtime. Currently, the message queue itself is implemented as a linked list as only a pointer to the head element is required. If the scalability
of stream management is a concern, a more dynamic stream queue management can be adopted.

### 3.2.3 Progress Engine

Each CH3 device requires a progress engine to transition the middleware's abstraction states. The states that are being managed in our implementation consists of VC and MPI request states. VC states represent the connection state of each endpoint in MPI.COMM.WORLD. In the other case, an MPI request state represents the completeness of application level messages and is divided into two general categories: read and write requests. MPI request management includes message matching done at an upper layer while write requests are passed down to the CH3 device for transmission. In order to map from socket level events to the application level, we introduce the notion of an application level event queue. Each successful socket event, either read or write, will be converted to a corresponding application level event that will result in a state transition for a particular object, VC or MPI Request.

The engine is also responsible for reading the incoming message from the socket buffer and transmitting pending send requests that are being queued from the CH3's base primitives. These routines are represented by Read and Write logic blocks respectively as shown in Figure 3.4. Read logic performs `sctp.recvmsg` to a global buffer along with the corresponding message metadata such as an association ID and stream number that are necessary for message de-multiplexing. The write logic retrieves queued MPI requests from a global send queue and tries to send them on the specified stream for a particular endpoint. Each global send queue element is only removed from the queue when it is completely finished. Notice that the read logic is placed before the write logic to ensure the SCTP-UDP style socket will not cause a deadlock within the
This pipelined design allows us to add more stages to the progress engine, if necessary, and this is where the iWARP integration for the progress engine occurs. These changes are described in Section 3.4.3.

### 3.2.4 Multihoming

In this section, we discuss how the multihoming feature of SCTP is supported within the MPI middleware to provide additional fault tolerance and performance improvement. Our implementation allows a user to provide a configuration file to specify which network interfaces the SCTP-UDP style listener and iWARP should bind. This selective binding capability allows the user to separate MPI traffic from other control traffic.
Ideally, we would like to have all the network traffic over SCTP. However, the network connection code for the MPI daemon was not modified in our implementation because there is not a portable Python binding to native SCTP sockets and most of the MPICH2 utility functions are written in Python. Nonetheless, with the use of a configuration file, we can separate the network traffic if necessary.

The configuration file path is defined as an environment variable on each node and the same file can be shared between all MPI processes. During initialization, the file is parsed and the SCTP-UDP style socket is bound specifically to the IPs that are provided within the configuration file via the use of SCTP's extended `sctp.bindx` call. Currently, each line of input is a combination of hostname and IP(s) delimited with spaces. This is sufficient for CMT to operate as it does not require any static network characteristics, like latency and bandwidth, to be provided in advance. In the case of a missing configuration file, we bind to the same IP address as the daemon.

Future work for CMT is to allow the application to provide feedback to the SCTP stack to schedule messages with respect to its requirements. For instance, we can inform the SCTP stack to send certain application messages on the lowest latency link. CMT currently only schedules messages to maximize bandwidth [13].

### 3.2.5 Process Spawning

MPI-2 introduces new API calls to perform process spawning (`MPI.COMM_spawn`). This is particularly useful if your application requires more processing power as it allows for the dynamic allocation of more MPI processes to transparently join MPI peers. In the case of client-server systems and task-farming jobs, more workers can be created or destroyed on demand and this can potentially utilize
the systems more efficiently and increase the scalability of user applications.

During the process spawning operations, an intracommunicator is created between the new MPI process and its parent process. After the new MPI process is spawned, it is free to join any other MPI communicators and this is called the "connect" phase. The procedure is greatly simplified with our dynamic connection support using an SCTP-UDP style socket because our dynamic connection is uni-directional and it does not require the target endpoint to acknowledge the message to enable it to form a virtual connection. As a result, MPI level connections are formed when a send request is triggered and the SCTP stack guarantees that an association is created between the endpoints.

3.2.6 Limitations

During initial development we decided to focus on correctness over performance. In this section, we discuss some of the limitations of our current implementation that uses an SCTP-UDP style socket [41] and propose some possible solutions.

- **Polling versus Simple Receive**
  
  When each MPI process only has one SCTP-UDP style socket, it makes sense to read from the socket directly during each progress wait iteration rather than performing a `poll/select` to check for incoming events. However, not all reads will return with data and this consumes CPU cycles. It may be possible to be more intelligent by blocking and unblocking the socket dynamically.

- **Memory copying**

  In our current implementation, we are reading everything into a global buffer and later copying it to its proper place. MPI messages have an envelope\(^2\) attached (except when it is part of a long message). The envelope

\(^2\)Recall that an envelope is an MPI level metadata that describes the attached MPI message.
is used to find the corresponding read request if there is any. Instead of sending the envelope and message data separately, a short message with envelope is transmitted in a single send call to minimize the network latency and the receiver can do a full read at once. Clearly, there is a tradeoff between the number of system calls made versus memory copies. This may be insignificant for small messages, but as message size increases, memory bandwidth may become a potential bottleneck. This is a consequence of the SCTP-UDP style socket in which you do not know where the next message is coming from. There are ways to minimize the memory copying overhead, namely to use more optimized memory copy routines, limit the amount that is being copied or to extend the SCTP API in a way that polling on associations is possible.

A more sophisticated design that requires both changes in the SCTP stack and MPI middleware is to allow the SCTP to perform a receive and peek of the next message in a single receive call or even a batch read\textsuperscript{3}. Recall that an MPI message needs to be matched with the attached envelope and with that, the future peek allows the middleware to set up the receive buffer in advance and thus eliminate the memory copy. However, not all future peeks will succeed because the next message may not be ready for reading since it may not have arrived to the kernel yet. On the other hand, a batch read allows the middleware to minimize the number of receive calls, but will increase the frequency of memory copying. Both design changes to the SCTP stack may not be general enough for other applications other than the use of an SCTP-UDP style socket for MPI.

- **Message Fragmentation**

  Since SCTP is message oriented, it is not possible to send a message that is

\[\text{\textsuperscript{3}This suggested change was made to the SCTP standard and is implemented in the FreeBSD's SCTP stack.}\]
larger than the send socket buffer. Therefore, a large MPI message needs to be fragmented manually and a threshold value smaller than the send buffer size needs to be selected. In the final design, we choose 64Kbytes, which is also the long and short message boundary for MPI. This adjustable value was chosen to minimize the number of message fragments and the time required to perform a memory copy at the receiver. MPI also supports the use of message I/O vectors for user-defined datatypes; this complicates our design when the sum of all I/O vector elements is larger than the threshold. In the case of a large message comprised of multiple I/O vector elements, we take a simple approach and send them one by one with respect to the threshold. This is certainly not optimal because of the added system calls to transmit each of the I/O vector elements. An alternative is to modify the I/O vector to have a sum that is equal to the threshold. However, both approaches highlight the shortcoming of SCTP's support for MPI. In a situation dealing with MPI large messages, one would want the transport layer stack to be byte-oriented. An extension to SCTP called, Explicitly End-Of-Record, would allow the user to explicitly mark the end of a message to inform the stack. This would certainly be useful in the case of large message transferring, but was not attempted in this design.

- **MPI\_ANY\_TAG**

MPI often requires only partial ordering with respect to the message's TRC (tag, rank, and context), with one exception, which is the use of MPI\_ANY\_TAG. MPI\_ANY\_TAG allows an MPI endpoint to receive a message with a wildcard match on the tag value. Though wildcard use with a receive is useful when writing an MPI application, it complicates our channel design with SCTP. Recall that our implementation is utilizing
SCTP streams by providing mapping to MPI message tag and context. As a result, messages sent on different streams can overtake each other; this is the intended behavior to avoid potential head-of-line blocking. On the other hand, MPLANY_TAG semantics require the endpoint to receive a message in the order of the corresponding sends and this can be difficult to achieve because SCTP only provides partial ordering across streams (Figure 3.5). One possible solution is to include an application level sequence number or timestamp within the message envelope to perform matching but this would require adding a separate code path to handle this special case. Another alternative would be to assign one of the SCTP streams as the control channel and send all message envelopes on that stream to serialize the order at the receiver. This allows the envelopes to arrive at the endpoint to have a total ordering while the data still flows through different streams. The reassembly of the message may be complicated when the data arrives earlier than the envelope itself.

Figure 3.5: MPLANY_TAG issue

- Multithread support
  The MPI specification allows multiple user application threads to be created. However, it depends on the middleware whether the use of threads
is supported; this can be done by querying the middleware during MPI initialization\textsuperscript{4}. By supporting multiple user threads, this essentially allows concurrent access to the progress engine. Currently, our implementation does not support use of threads as most of the internal data structures are not protected, including the global event queue and send queue. In order to provide full support for multithreading (assuming the MPI middleware needs to guard every MPI related data structure), various locks need to be inserted in all queue structures and in any potentially shared objects such as an MPIRequest. This undoubtedly would add considerable overhead to the runtime as acquiring and freeing the locks are necessary. As a result, we decided not to tackle this problem and focused on implementing as many MPI features as we can for the channel device.

### 3.3 iWARP Implementation

iWARP is an RDMA protocol for wide area networks. Our implementation is based on Ohio Supercomputing Center’s (OSC) software iWARP stack\textsuperscript{5} and is extended with more capabilities \[16\]. The OSC’s iWARP stack is implemented, with respect to the RFC specifications, and is shown to be compatible with actual iWARP capable hardware. In this section, we discuss some of the design changes that were made to add more flexibility and functionality to the software stack. The OSC’s software iWARP stack is denoted as OSC-iWARP.

\textsuperscript{4}MPICH2 CH3 Thread Support: http://www-unix.mcs.anl.gov/\%7Ebalaji/support.html

\textsuperscript{5}OSC iWARP project site: http://www.osc.edu/research/network_file/projects/iwarp/iwarp_main.shtml
3.3.1 Direct Data Placement (DDP) over SCTP

In this section, we discuss the implementation details of DDP over SCTP. The main benefit of using SCTP as the underlying reliable transport layer protocol is that the protocol itself also supports many of the criteria that DDP requires. Recall that DDP is an end-to-end protocol and each DDP segment has the property of being self-describing and self-contained. SCTP is a perfect candidate because SCTP is message oriented. Unlike for the case of TCP where MPA was introduced as an extension to solve TCP/IP's shortcoming in message framing and insufficient checksum. Furthermore, the MPA layer within the software stack is fairly heavyweight; it contains the majority of the code for the iWARP implementation. Developers have been trying to implement MPA as a separate layer on top of TCP/IP, but its performance is suboptimal because of additional copying from layer to layer. In order to solve these resulting performance problems, another strategy is to insert the MPA changes directly within the TCP/IP stack. While this improves the overall performance of MPA-TCP/IP, it also sacrifices the backward compatibility of the TCP stack.

The IETF draft proposed by Caitlin and Randall [12] introduces an adaptation layer to bridge between DDP and SCTP. Part of its design includes defining DDP related chunks within SCTP for connection setup and DDP data transfer. Moreover, it requires the use of SCTP's unordered message delivery. This is certainly a more complete design than our implementation. Nonetheless, our implementation is using the existing data chunk and utilizes the SCTP features as well. We also believe that DDP itself is a fairly thin layer when the underlying transport layer can handle the majority of the complexity. As a result, the DDP layer within the OSC-iWARP is implemented directly on top of SCTP.

Recall that negotiation communication is required for both iWARP capable endpoints to exchange key pairs (for security reasons), network hardware infor-
information, MPA marker locations (in the case of TCP), and other information. Our implementation currently disables the negotiation stage during connection setup for the simplicity of design.

### 3.3.2 DDP header

Since SCTP provides message framing by default, it gives the application layer protocol more flexibility in choosing the segment size. This also allows us to remove two bytes of MPA related fields for the DDP header for both the tagged and untagged models. These two bytes represent the DDP segment length and it is necessary for the MPA to include this information within each DDP header to allow the receiver to function correctly. However, this is not required for SCTP since message boundaries will be preserved. Even though the bytes saved is not overwhelming, SCTP can handle messages better than TCP by managing all the complexity of message framing.

### 3.3.3 DDP to SCTP semantic mapping

Recall that DDP provides two models of message transfer, untagged and tagged. An untagged message is analogous to the traditional send/receive and each DDP untagged segment is defined by a set of queue numbers, DDP level message number and buffer offset. The tagged model allows a DDP segment to be copied to the designated user buffer directly. Therefore, each DDP tagged segment contains a steering tag and offset to match the receiver’s buffer.

In order to utilize multiple SCTP streams, we use a simple, yet intuitive mapping. We use the steering tag to define which stream to send for the tagged model and message number for the untagged model. The software iWARP’s DDP layer assumes that DDP segments will arrive in order and the last arrived segment is the end of the message. However, this is not required as one can
change the DDP layer to support unordered delivery of each DDP segment and this can be supported with SCTP as well by using multiple streams and the \texttt{MSG\_UNORDERED} message flag.

Currently, our DDP over SCTP uses the SCTP-TCP style socket for the underlying connections. This decision was made to preserve the one to one relationship between queue pairs and to minimize the changes to the software iWARP stack. However, we believe that the use of SCTP-UDP style is also feasible, but it would require a more extensive redesign of the send and receive logic. In addition, a more complete implementation would be needed to replace the use of file descriptors to association IDs to avoid an internal lookup within the SCTP stack.

### 3.3.4 Asynchronous support

The OSC-iWARP is mostly synchronous by default, as a result all Verbs calls are blocked except for read requests. This blocking behavior is not appropriate for MPI semantics since it potentially stalls the middleware and prevents it from progressing multiple outstanding requests simultaneously. Due to this limitation, we extended the stack to provide the intended asynchronous behavior. The OSC-iWARP exports two main functions: one for posting send work requests and one for posting receive work requests.

There are two alternatives to provide asynchronous support

- **Single threaded**

  A single threaded design requires each socket related system call to be nonblocking. As a result, when posting a work request to the RDMA layer, it may not be completed immediately and requires probing later to check for completeness. Work request states need to be saved and updated accordingly. This design is analogous to the MPI middleware's progress
engine in which it needs to process upcoming requests. The disadvantage of this approach is that everything is queued upon the progress function being called, which potentially increases the delay to process each work request.

- **A progress thread**
  We decided to introduce a progress thread to simulate the iWARP capable network device. Each work request will be enqueued to specific queues and the progress thread will process them in the same order. Shared data structures such as the completion queue is required to have a mutex when inserting or removing a completion entry. Moreover, work requests can now proceed as soon as possible without waiting. In our current implementation, the progress thread is created at the RDMAP layer and it does not handle read requests. iWARP requires each successful read to have a corresponding buffer. However, if our progress thread is handling read requests, it is possible that the socket buffer has data, but the progress thread does not know where to place the data, because the read request has not been posted yet. This is equivalent to unexpected messages in MPI and requires a more complicated buffering mechanism inside the iWARP stack and it also potentially performs redundant work when integrated with the MPI middleware. As a result, we decided to not have the progress thread progress read requests. Instead, read requests will be progressed when iWARP level polling is called.

One of the disadvantages with a single progress thread design is that queue pairs not related to each other may be blocked. However, this can be solved by introducing a thread per queue pair or threads at a lower level such as DDP. Our use of progress threads is merely to provide the asynchronous support for the exposed API and to simulate an iWARP
Chapter 3. Design and Implementation

hardware device. As a result, performance can be suboptimal at times. Moreover, by introducing threads to the stack, data structures also need to be locked and unlocked appropriately in order to avoid race conditions.

3.3.5 User level API, Verbs

The OSC-iWARP exports a set of functions and semantics that utilize the use of the RDMAP layer called Verbs [4, 26] (Figure 3.6). The API is very similar to the InfiniBand API and the Connection Management feature is not included (Table 3.1). A new version of OSC-iWARP supports OpenFabrics API [2] which is an industrial standardized API for RDMA based devices.

![Figure 3.6: iWARP semantics and modules](image)

The semantics of the verbs used by OSC-iWARP are endpoints are connected via Queue Pairs(QP) and send/receive or write/read requests are defined as work requests via the use of iwarp_qp_post_sq/iwarp_qp_post_rq. Each QP is at-
### Verbs

<table>
<thead>
<tr>
<th>Verbs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iwarp.rnic.open/close</td>
<td>Open/Close a RNIC handle</td>
</tr>
<tr>
<td>iwarp.rnic_advance</td>
<td>Advance call to progress read requests without creating completion entry</td>
</tr>
<tr>
<td>iwarp.qp.create/destroy</td>
<td>Allocate/Destroy a Queue Pair</td>
</tr>
<tr>
<td>iwarp.qp.post.sq</td>
<td>Post a send work request on behalf of the QP (SEND/RDMA.WRITE/TERM)</td>
</tr>
<tr>
<td>iwarp.qp.post.rq</td>
<td>Post a read work request on behalf of the QP (RECV/RDMA.READ)</td>
</tr>
<tr>
<td>iwarp.qp.active_connect</td>
<td>Actively connect to a QP (client mode)</td>
</tr>
<tr>
<td>iwarp.qp.passive.connect</td>
<td>Passively connect to a QP (server mode)</td>
</tr>
<tr>
<td>iwarp.qp.register.fd</td>
<td>Directly register a file descriptor (fd) with the iWARP layer (either server or client mode)</td>
</tr>
<tr>
<td>iwarp.qp.disconnect</td>
<td>Disconnect a QP</td>
</tr>
<tr>
<td>iwarp.cq.create/destroy</td>
<td>Create/Destroy a completion queue</td>
</tr>
<tr>
<td>iwarp.cq.poll/.block</td>
<td>Polling a completion queue</td>
</tr>
<tr>
<td>iwarp.pd.allocate/deallocate</td>
<td>Allocate/Deallocate a Protection Domain (PD) ID</td>
</tr>
<tr>
<td>iwarp.nsmr.register</td>
<td>Memory registration, returns steering tag and a memory region</td>
</tr>
<tr>
<td>iwarp.deregister_mem</td>
<td>Unregister a memory region</td>
</tr>
<tr>
<td>iwarp.deallocate_stag</td>
<td>Deallocate the steering tag, does not free nor de-register memory</td>
</tr>
<tr>
<td>iwarp.create.sgl</td>
<td>Create/initialize the scatter gather list (SGL)</td>
</tr>
<tr>
<td>iwarp.register.sge</td>
<td>Register/Insert the scatter gather entry (SGE) into an SGL</td>
</tr>
</tbody>
</table>

Table 3.1: OSC-iWARP Verbs (Function calls extended from OSC-iWARP are underlined)
tached to two Completion Queues (CQ) for completion notifications: send and receive respectively. There are primarily two types of work requests that iWARP supports: Channel based (Send/Receive) and Memory based (Write/Read). Send/Receive is the traditional Copy-in/out model while the Write/Read allows the direct transfer of the message to user specified buffers. There is also a notion of Protection Domain (PD) and this acts as a context for the registered buffers. The iWARP specification does not define the memory registration (pin/unpin) mechanism. The traditional Send/Receive request will map to the underlying untagged model in DDP, and tagged model for \textit{RDMA.Write/Recv}.

Each work request is uniquely identified by a request ID and depending on which type of work request, it is necessary to specify sufficient information including buffer information in the format of a Scatter Gather Entry (SGE), and steering tags (for the \textit{RDMA.Write/Read}). For each work request, one can also optionally request a completion signal. Polling is performed on the completion queue (CQ) via \texttt{iwarp.cq.poll} and for scalability, a given CQ can be shared among many QPs.

3.3.6 Limitations

- **Polling from completion**

  The poll function provided by the OSC-iWARP verb does not allow batch polling. As a result, each execution of \texttt{iwarp.cq.poll} will result in only one iWARP level completion entry being returned. However, this can easily be extended by changing the API or by exposing the CQ structure to the application.

- **iWARP level Connection Management**

  OSC-iWARP verbs provide only a synchronous version of connect and accept and this is not sufficient for the use of MPI or other applications that
may require a nonblocking connection setup. As a result, we extended from the basis of passive (or active) connect to allow the iWARP layer to directly register a file descriptor regardless of the transport layer protocols (iwarp_qp_register_fd), this emphasizes the flexibility of a software iWARP stack. By directly registering a file descriptor with the iWARP layer, users can potentially create their own connection framework. One future extension for this function is to allow users to specify the mode of operation (TCP or SCTP) and the DDP layer can adapt dynamically to permit the coexistence of both transport layer protocols.

3.4 MPICH2-Hybrid

In this section, we discuss our experience in integrating the software iWARP module within our ch3:sctp device and some of the design choices. The final hybrid CH3 device allows a mixed use of network devices for message transfer based on the higher level functional requirements (see Figure 3.7). Moreover, the introduction of iWARP should not interfere with the operations from the original ch3:sctp channel. Recently, MPICH2 with iWARP integration has been described in [46]. Our implementation differs in that there is no connection management framework, a different Verbs API is used, and we are using SCTP. As a result, challenges are different.

3.4.1 Memory Management

The correct use of a RDMA capable device requires proper management of the memory. In order to receive any message, memory needs to pre-posted and deallocated dynamically. In our current implementation, we allocated a 32 Kbytes buffer for each VC during initialization and this buffer is registered with the iWARP layer. The primary use of this buffer is for untagged message transfer
of general purpose messages such as MPI envelopes or acknowledgments. The memory information is also exchanged during connection setup. As a result, this buffer can also be used for the tagged model when necessary.

### 3.4.2 Connection Management

Each MPI process holds an RNIC handle to the simulated hardware exposed by the software iWARP stack. All the Verbs components (Figure 3.6) are allocated with reference to the RNIC handle. In order for iWARP capable endpoints to communicate with each other, at least one QP for a pair of endpoints must be allocated. The OSC-iWARP does not provide nonblocking connection management routines. As a result, we have decided to implement the connection framework within MPICH2. Furthermore, our connection framework is using the extended iWARP connection API to directly register a file descriptor regardless of the transport layer protocols; this emphasizes the flexibility of a software iWARP stack. This allows us to exploit possible optimization for MPI based connection management. The other alternative would be to use the RDMA-CM interface.
defined by the Open Fabrics Alliance [2], CMA (Communication Manager Abstraction), is a higher-level CM that operates based on IP addresses. This is a more general solution because it is transport independent and it provides an interface that is closer to the use of sockets, but it is also based on the use of specific verbs that is different than our implementation.

In order to prevent the iWARP features from interfering with our original ch3:sctp device, we have allocated another SCTP-UDP style socket for handling RDMA connections. We propose the following two ways to manage iWARP QPs:

- **Static**
  
The idea of static connection management is to allocate a QP for each pair of endpoints during initialization. This is analogous to allocating $N(N-1)$ file descriptors (QPs). Moreover, this is the approach of most MPI-RDMA implementations and is acceptable for small groups of processes. However, as the group of processes becomes larger, the initialization time can be significant.

  Recall that each MPI process is assigned a global rank during initialization and in order to prevent connection collision, we only allow each MPI process to connect to other peers that have higher rank than itself (as shown in Figure 3.8). Hence, each MPI process will be both connecting and accepting during the initialization. Barrier calls are inserted to distinguish between the different connection phases.

  As we have mentioned, SCTP provides both UDP and TCP style socket semantics for backward compatibility to legacy applications. It is also possible to use one style of socket to connect to the other and vice versa. This allows the user to piggyback application level data with an initialization request. Our connect-all mechanism requires specific use of SCTP’s API
and this is not available in TCP. Therefore, the implementation itself is not generalized, but the concept is still applicable. SCTP has a specific call, sctp_peeloff, that allows an association to be peeled off from an SCTP-UDP style socket to form a separate SCTP-TCP style connection. This operation does not require any coordination between endpoints. Hence, we can directly register this newly peeled off association with the iWARP layer and with our new QP connection extension call. The connection phase requires the initiator to send a message to the target’s SCTP-UDP style listener and the message consists of its own rank and the information about the locally allocated 32 Kbytes of memory. The target endpoint receives the message, peels off from the association, sends back its own local memory information back to the sender, and registers with the iWARP layer. The initiator then receives the accept-ACK, peel off and registers the connection with the iWARP layer. In the end, an one-to-one connection is formed between any pair of endpoints.

It is worth noting that this static approach is different from the connection management in our ch3:sctp that supports dynamic connection by default. As a result, we are proposing a dynamic approach for QP management as well.

- Dynamic

Dynamic connection management allows the middleware to allocate QP on demand. This approach, similar to ch3:sctp, can reduce the number of file descriptors (QPs) being allocated. However, it will increase the communication cost for the first MPI message transfer.

The main challenge to deal with dynamic connection management for QP is to handle collisions. In the case of two processes trying to connect to each other at the same time, we use the MPI globally unique rank in
MPI.COMM_WORLD to break ties (Figure 3.9). Notice that collisions do not occur in ch3:sctp due to the fact that it is based on the SCTP-UDP model and connection is done in a per uni-directional stream basis.

Our implementation requires a pair of endpoints to perform an application level 2-way handshake to exchange connection packets in order to ensure that the one-to-one connection is properly formed and ready for iWARP operations. Since we are directing iWARP connection traffic to another SCTP-UDP style socket, we need to periodically poll the socket for connection requests; this code block is added to the progress engine.

Dynamic connection management can help to utilize the hardware resources more efficiently. However, it also adds more complexity to the internal design in which the pending outgoing messages now need to be queued because the QP is not connected. Moreover, it does not need to be implemented within MPI and can be part of an external connection management module. On the other hand, the static connect-all approach is fairly easy to implement and the MPI messages can simply be posted to the iWARP layer. Overall, the static approach can manage less states by sacrificing some resources during initialization.
3.4.3 Progress Engine Extension

Recall that our ch3:sctp device uses pipelining and an application level event mechanism for message progression. An extension to support the iWARP layer is implemented by adding another stage to the progress engine.

The pipelining stage for iWARP basically performs a poll (iwarp_cq_poll) and dequeue on an iWARP level completion entry. Currently in our implementation, every QP is sharing a global completion queue. This is a simple approach to allow the progress engine to have only one CQ to poll for completion. Each completion entry contains the following information: QP entry, type of operation, and amount of data sent or received. Therefore, it can easily be mapped to an application level event. Our application level event handling routine is also extended to de-multiplex iWARP level events based on the QP and the iWARP request type. Since iWARP denotes endpoints by QP, a hash table was added to map a QP to VC and the table is populated during connection setup.
Adding another stage to the progress engine imposes a performance penalty. Our implementation does try to minimize the impact as much as possible. One possible technique is to dynamically activate or deactivate the added stage by closely monitoring if there are any MPI-2 operations being executed.

### 3.4.4 MPI-2 RMA semantic mapping

In this section, we discuss how the MPI-2 one-sided semantics map to iWARP and the corresponding implementation details. MPI-2 defines `get`, `put` and `accumulate` operations for remote memory access (RMA). These operations involve data movement that can be initiated by the action of one process, hence the name remote memory operations (one-sided). Moreover, the MPI-2 RMA specification does not guarantee any performance improvement over send and receive. It merely serves as an extension to the MPI specification to support RDMA capable hardware.

The MPI-2 calls that we will focus on are `MPLWin_create/free` for window creation/destruction, `MPLPut/Get`, and `MPLWin_fence` call to synchronize window access (Table 3.2). `MPLPut` allows an MPI process to directly put specific data to the target's window memory without any intervention, whereas `MPLGet` allows an MPI process to retrieve data from local or remote memory without the target endpoint's attention. We did not implemented all of MPI-2's one-sided communication primitives. For instance, users can perform more scalable window synchronization with `MPLWin_start/complete` and `MPLWin_post/wait`.

Our implementation is based on MPICH2's one-sided functions with some hooks added to intercept the data transmission. Even though it is not a complete rewrite of the RMA functions, it is sufficient to demonstrate how different communication and internal code paths can be used to explicitly separate MPI-
Chapter 3. Design and Implementation

Table 3.2: MPI-2 RMA function prototypes

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Put</td>
<td>[ \text{int MPI_Put}(\text{void }^*\text{origin_addr}, \text{int origin_count}, \text{MPI_Datatype origin_datatype}, \text{int target_rank}, \text{MPI_Aint target_disp}, \text{int target_count}, \text{MPI_Datatype target_datatype}, \text{MPI_Win win})]</td>
</tr>
<tr>
<td>MPI_Get</td>
<td>[ \text{int MPI_Get}(\text{void }^*\text{origin_addr}, \text{int origin_count}, \text{MPI_Datatype origin_datatype}, \text{int target_rank}, \text{MPI_Aint target_disp}, \text{int target_count}, \text{MPI_Datatype target_datatype}, \text{MPI_Win win})]</td>
</tr>
<tr>
<td>MPI_Win_fence</td>
<td>[ \text{int MPI_Win_fence}(\text{int assert}, \text{MPI_Win win})]</td>
</tr>
</tbody>
</table>

1\'s point to point operations from MPI-2\'s one-sided routines.

Before RMA operations can take place, a window object that consists of all the distributed window information must be created with \textit{MPI\_Win\_create}. In order to utilize DDP\'s tagged model, window memory needs to be registered with the iWARP layer during the window creation process. Window creation is a collective operation in which all the participants exchange information about their local memory window, if there is any. The information typically consists of the buffer location, length, and offset. In our case, we include the steering tag for the local memory window for future use. At the end of the process, each participant has the knowledge of others\' memory window and thus forms a set of remote distributed memory. A memory window also needs to be destroyed properly and collectively with \textit{MPI\_Win\_free}. During the process of destroying a window object, the pending RMA operations are completed before exiting. Moreover, each participant needs to unregister its local memory window with the iWARP layer.

MPICH2 adopted a delayed approach to the window synchronization call, \textit{MPI\_Win\_fence}. For instance, MPICH2 tries to queue up all the put/get operations until a later fence call occurs. This method is shown to be more latency tolerant than using simple barrier calls; the delayed approach is more appropriate for TCP and SCTP where latency is relatively high [22]. However, the performance of the window synchronization can be highly dependent on the
underlying hardware. For instance, there is shared-memory hardware that is capable in performing barrier calls more efficiently than reduce-scattering collective call that is being used in the delayed approach.

The window fence call is a collective operation for all participants in the window creation. During the first stage, an MPI process tries to gather the total amount of window operations for all the participants via an MPI collective call. The total count of RMA operations is important because it is used for allocating MPI requests (read/write) and for fence termination. Next, it will process all of the queued put, get and accumulate operations; it is at this point where we intercept the calls to perform iWARP specific operations. The total count of RMA operations is updated throughout the process. The final stage is for the progress engine to wait until all of the RMA operations are completed and the allocated resources are freed accordingly.

MPICH2 performs MPI_Put by bundling an envelope with the corresponding data and this MPI message is then transmitted during the later fence call. Though this can easily be supported with our ch3:sc.tp with its point-to-point primitives, there is a mismatch between the application level semantics and the lower level primitives. As a result, we wish to provide a more direct mapping for MPI_Put by utilizing the iWARP verbs. Recall that each MPI message has an envelope attached to the actual data and in the case of MPI_Put message envelope, it contains information about the target window's buffer location, offset and length. In order to avoid the envelope from being copied to the target's memory window and to utilize both DDP's tagged and untagged model, we decided to decouple the envelope from the data. As a result, the data is being sent by using the tagged model in which it provides a direct semantic mapping via posting a RDMA_Write request to iWARP (Figure 3.10). Recall that the tagged model requires the use of a steering tag and this information can be
retrieved from the local window object. After a data send request is completed, the envelope is then sent to the target by using the untagged model and it serves as a notification to the target that its memory has been updated and is ready for access. The completion notification is necessary because \texttt{RDMA.Write} is completely transparent to the target; this is the intended behavior in the RDMA specification.

![Diagram of MPI_Put via RDMA_Write during fence call](image)

Figure 3.10: \texttt{MPI.Put} via \texttt{RDMA.Write} during fence call

In the case of \texttt{MPI.Get}, it allows a local process to directly read data from a local/remote window and this can be analogous to a \texttt{RDMA.Read} or \texttt{RDMA.Write} request depending on which side is responsible for carrying out the operation. There is a minor complication due to the requirement that the local buffer that originally held the upcoming data may be a different buffer than its local memory window. Therefore, before we issue a \texttt{RDMA.Read/Write} request, we need to register the local buffer memory whenever the buffers differ.
We now discuss how MPI.Get can be implemented with either RDMA.Read or RDMA.Write:

- RDMA.Write

A straightforward way to perform MPI.Get via RDMA.Write is to send the MPI envelope to the target with the steering tag as the application data by using the untagged model. The steering tag is needed to inform the target endpoint where to perform RDMA.Write because the MPI.Get buffer is not necessarily the same as the window. Upon receiving the application metadata at the target endpoint, the receiver can then perform a RDMA.Write back to the sender by using the information from the global memory window and the metadata message. After completing the RDMA.Write request, it will send a MPI.Get response envelope back to the sender to serve as a completion notice (Figure 3.11).

![Figure 3.11: MPI.Get via RDMA.Write during fence call](image-url)
- **RDMA\_Read**

  RDMA\_Read requests requires both the source and target steering tag to be specified. In the underlying design, the iWARP stack handles RDMA\_Read by first issuing an untagged message of the request's metadata to the target (Figure 3.12). The target endpoint's iWARP stack then issues a tagged message transfer of the requested data directly back to the source buffer. After the completion of the RDMA\_Read request, we use the same approach as a MPI\_Put. We send the MPI envelope as an untagged message to the target to serve as a completion notice.

  ![Diagram of MPI\_Get via RDMA\_Read during fence call](image)

  **Figure 3.12:** MPI\_Get via RDMA\_Read during fence call

  The only difference between these two approaches is that for RDMA\_Read, the first untagged message transfer will not notify the MPI middleware as the iWARP stack will handle it internally. In the case of RDMA\_Write, the MPI
layer is responsible for handling metadata messages and performing the RDMA write. As a result, there is a shift in terms of which side is responsible for carrying out the actual message transfer. The actual number of messages transferred at the iWARP level for both approaches for $MPI.Get$ is the same.

### 3.4.5 Limitations

The main problem with MPI-2 remote operations is that it is not completely one-sided. It requires a global synchronization for each memory window. Moreover, one is not allowed to touch the remote memory before the synchronization call and this adds significant overhead.

Recall that we integrated the iWARP poll routines into the progress engine and it is performing a nonblocking poll to prevent deadlock with the other progress engine stages. This is not optimal as it wastes CPU cycles and imposes locking overhead within the kernel. A better solution is to set the blocking mode of iWARP poll dynamically regarding the status of MPI-1 requests. However, this might be a nontrivial condition to detect.

Our current implementation of $MPI.Put/Get$ does not support the use of complex I/O vectors for user defined datatypes due to lack of support in the iWARP software. We can have some preliminary support for I/O vectors by converting each I/O vector element to have proper offsetting to the target buffer, but there still exists issues with packing/unpacking complex datatypes which is out of the scope for this work. As a result, we are only supporting the use of pre-defined datatypes and this limits us from running complex MPI-2 applications.

The experiments conducted in Section 4.2 only make use of primitive types to perform some base evaluations of the MPI-2 one-sided operations.

We are only working with a subset of MPI-2 one-sided operations and there are more synchronization calls that have not been covered in this work. There
are also proposals [45] for the MPI-2 RMA specification to be less restrictive in order to allow local completion. In addition, there are often many ways to implement MPI-2 calls, but they are hardware dependent. Hence, performance is a difficult problem to tackle as it involves all of these variables.
Chapter 4

Evaluation

In this chapter, we evaluate some of the components in our hybrid design and provide some preliminary results.

4.1 Experiment setup

Most of the experiments were conducted on an IBM eServer x306 cluster with four nodes, each node with a 3.2 GHz Pentium-4 processor, 1 gigabyte of RAM, and three GbE interfaces. Of the three interfaces, two private interfaces were on separate VLANs connected to a common Baystack 5510 48T switch, and the third (public) interface was connected to a separate switch, similar to the setup shown in Figure 4.1. Jumbo frames of size 9000 bytes are also used in the CMT experiments to exploit the potential GbE performance.

We are using FreeBSD 6.2 with the SCTP stack installed as a patch\(^1\) to the kernel and this is the most feature rich stack in which the CMT extension is included. In addition, this is also the most actively developed SCTP stack and it has been ported to many other operating system such as MAC/OS. Starting from FreeBSD 7, SCTP will be officially part of the OS distribution. Other implementations exist on Linux\(^2\) and Windows platforms and a user level stack\(^3\) are also available.

\(^1\)A patched version of the stack (http://www.sctp.org), compile as a kernel option
\(^2\)LKSCTP: http://lksctp.sourceforge.net/
\(^3\)User Level SCTP: http://www.sctp.de/sctp-download.html
For the tests with SCTP and CMT we used the standard MPICH2 v1.0.5, which comes with the point-to-point channel (ch3:sctp) and extended to include the hybrid channel (ch3:hybrid). We tested MPICH2 with and without CMT enabled and compared it to TCP from the same release of MPICH2 (ch3:sock). We also tested Open MPI v1.2.1, which supports message striping across multiple IP interfaces [53]. Open MPI was implemented with one and two TCP Byte-Transfer-Layer (BTL) modules to provide a comparison between using one interface versus two interfaces.

For all tests we used a socket buffer size of 233 Kbytes, the default size used by ch3:sctp, and we set the MPI rendezvous threshold for MPI to 64 Kbytes, matching ch3:sctp and Open MPI's default.

Figure 4.1: Cluster Configuration

4.2 MPI_Put/Get performance

In this section, we test the MPI_Put/Get performance over iWARP and compare it to ch3:sctp. The test program measures the time of MPI_Win_fence to synchronize the one-sided operations over various message sizes ranging from
2 bytes to 4 Mbytes between two nodes. We measured the time to perform \textit{MPL\_Win\_fence}, which is where the majority of MPI-2 one-sided time is being spent and the result is shown in Figure 4.2.

![Figure 4.2: MPI\_Put/MPI\_Get latency test](image)

First observation is that the SCTP and iWARP performance for both \textit{MPI\_Get} and \textit{MPI\_Put} share a similar trend. Moreover, our software implementation of iWARP adds an overhead ranging from 2\% to 8\%. As well, notice that the software stack performs considerably worse when the message size increases into the Mbytes ranges. We believe that this is because the DDP layer is fragmenting messages at 16 Kbytes\(^4\). Even though this number is adjustable, it also depends on the send and receive buffer sizes for iWARP endpoints. In addition, we could

\(^4\)This adjustable value was chosen to be 16 Kbytes because it gives the best result in our setup.
allow the DDP to fragment at 64 Kbytes, but the receiver may not be able to receive at the same rate. In fact, we did see a performance decrease when the DDP fragment was set to 32 Kbytes. Another reason for the difference in performance when switching between threads is that our ch3:sctp is performing nonblocking operations by default, consequently when requests are not finished, the progress engine is polling. Polling is necessary because it needs to progress through the iWARP level's requests. However, it also adds overhead when going through the other progress engine stages that are basically idle. We believe that the software iWARP performance can be improved further by adopting a separate thread for reading incoming iWARP level messages. In addition, notice that \texttt{MPI.Get} is typically slower than \texttt{MPI.Put} because of additional metadata messages being exchanged.

Our next test is a synthetic benchmark that we have written from scratch that combines use of both MPI-1 and MPI-2 functions. The attempt is to show, by separating these two independent MPI APIs internally, we can increase efficiency. The comparison is between our ch3:sctp and ch3:hybrid. The benchmark consists of a series of \texttt{MPI.Isend} messages of various sizes and \texttt{MPI.Put} and \texttt{MPI.Get}. \texttt{MPI.Waitall}/\texttt{Waitany} functions and \texttt{MPI.Win._fence} are placed in specific places to allow overlapping between MPI-1 and MPI-2 message transferring. Moreover, this test program performs bi-directional flow of data in which both endpoints are sending and receiving from each other. The result is summarized in Figure 4.3.

The result has shown that the hybrid channel under mixed usage of MPI syntax can perform better than our original SCTP channel. This is mainly because MPI-2 requests are handled separately by a different component, namely iWARP, in the hybrid channel. On the contrary, the SCTP channel treats every type of MPI message the same regardless of the fact that MPI-1 and MPI-2
messages are independent. As a result, MPI-1 and MPI-2 messages are queued together thus having the potential of blocking each other. Moreover, by allowing the iWARP module to handle the MPI-2 messages, we explicitly have two progress engines that are working concurrently and thus increasing the overlap within the middleware.

### 4.3 CMT Related Experiments

In this section, we evaluate the performance when using SCTP with CMT to enable the multirailing support. As most computers have multiple network interfaces, it would be advantageous to be able to use all available resources to maximize the performance. We performed experiments to test both the performance and fault tolerance aspects of the design.

The first test we conducted was with respect to the effect of SCTP’s multihoming feature to user applications. Recall that multihoming allows an application to fail over transparently when one of the links is down. The fail over process is performed by switching over the network traffic from the primary
path to the secondary path in the event of link failure. Since CMT is based on multihoming, it inherits the fail over property as well as providing performance improvements in bandwidth.

To simulate link failures, we created firewall rules\(^5\) within FreeBSD to block specific link traffic at the IP level. The SCTP stack is also tuned to recover within 5 seconds (This can be even shorter, but we chose 5 seconds to distinguish the failover and recovery phases in the result). We also tested a new extension called CMT-PF (Potential Failure for CMT) \(^35\) in which it includes a new destination state called the “Potential Failed” (PF) state, and enhances CMT’s retransmission polices to include the PF state. The test demonstrates how CMT-PF can recover more smoothly than CMT during failures which can be beneficial for cluster computing.

The program that we will be using is a modified version of Iperf\(^6\), a bandwidth measurement tool, that supports SCTP and has been extended to support multiple interfaces. Iperf will report bandwidth values every 1 second and the test runs for 120 seconds in total. During the test, we inject link failures on the server side at specific time intervals while application is still running and the result is shown in Figure 4.4.

One thing to note is the secondary link is not as fast as the primary link due to differences in hardware; this is also a good indication in the graph to show where the switching actually takes place. Another observation is that multihoming actually switches back to the primary path when it recovers and this is the intended behavior of SCTP with preference to the primary path. In the case of link failures, the bandwidth for multihoming will drop to zero for about 2-3s while CMT will transmit data in short bursts (about 62 Mbps) and CMT-PF can smoothly adapt to link failures by predicting the upcoming failures

\(^6\) Iperf: http://dast.nlanr.net/Projects/Iperf/
The next test determined if our implementation of MPI over SCTP with CMT was able to exploit the available bandwidth from multiple interfaces. We used the OSU MPI microbenchmarks [36] to test bandwidth and latency with varying message sizes. We tested with MTUs of 1500 and 9000 bytes and found that using a 9000 byte MTU improved bandwidth utilization for both TCP and SCTP. Figure 4.5 reports our results with MTU 9000.

Since there is no multirailing support for MPICH2-ch3:sock, we have de-
cided to include some test results from Open MPI for comparison, which is another popular MPI implementation. Open MPI performs application striping at the middleware level regardless of underlying hardware, therefore allowing us to compare and contrast the different design approaches. Notice that by comparing two different MPI middleware, it will be difficult to analyze and draw a definite conclusion. As a result, we try to normalize as many parameters as possible such as socket buffer sizes and the large message cutoff\textsuperscript{7}, but the difference in behavior may also be due to the differences in the middleware implementation. Consequently, we only provided the basic observation of the experiment result.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{OSU_Bandwidth_Test.png}
\caption{OSU Bandwidth test}
\end{figure}

As shown in Figure 4.5, MPICH2-ch3: sctp with CMT starts to take advan-

\textsuperscript{7}Because the network paths were not identical due to differences in hardware, we set network characteristics in Open MPI's \texttt{mca-params.conf} file.
tage of the additional bandwidth when the message size exceeds 2 Kbytes. At a message size of 1 Mbytes, MPICH2-ch3:sctp with CMT is almost achieving twice the bandwidth (1.7 versus 0.98 Gbps) in comparison to using one interface, and it is outperforming Open MPI (1.56 Gbps) which is scheduling messages in user-space across two interfaces within its middleware. All single path configurations are approximately the same, with a maximum of 0.98 Gbps; however for configurations using two networks, Open MPI is the best for message sizes 512 to 2 Kbytes bytes but once message size exceeds 4 Kbytes MPICH2-ch3:sctp with CMT is performing better.

One potential drawback with CMT is with respect to the linear scalability of the number of network interfaces. In the experiments, we only show the potential benefits of using two interfaces. However, with two or more interfaces, the host processor may be saturated with interrupt and checksum calculations. There are ways to deal with these challenges, such as enabling interrupt coalescing (will increase packet latency), using jumbo frames, and acquiring hardware to perform checksum offload for SCTP. The other requirement for CMT is to design the network typology in a way that there is no shared bottleneck between network paths, which is essential if CMT is to achieve maximum performance and fault tolerance.

Nonetheless, SCTP with CMT allows user applications to enable transport layer striping with very little effort and this can be very useful for applications that require large bandwidth. More CMT results can be found in our previous work [13].

4.4 General Benchmarks

In this section, we will show some of the results for ch3:sctp in comparison to ch3:sock (the TCP implementation of CH3) with a series of MPI-1 mi-
cro/macrobenchmarks. In some of the experiments, we also compare the difference between our hybrid design and its parent design, ch3:sctp. Recall that ch3:hybrid is based on ch3:sctp and as a result, it should not impose any major overhead as it shares the same MPI-1 code paths as in ch3:sctp.

4.4.1 Microbenchmarks

The first test shown is a basic MPI latency test. The latency test is performed in a ping-pong fashion in which the sender sends a message with a certain data size to the receiver and waits for a reply from the receiver. The receiver receives the message from the sender and sends back a reply with the same data size. Many iterations of this ping-pong test were carried out and average one-way latency numbers were obtained (Figure 4.6). The MPI calls that are used in this test are blocking version of MPI functions (MPISend and MPIRecv).

In the next experiment, we measured bi-directional bandwidth of the MPI middleware. The bi-directional bandwidth test is similar to the bandwidth test in Section 4.3, except that both nodes send a fixed number of back-to-back messages and wait for the reply. In the end, this test measures the maximum sustainable aggregate bandwidth by two nodes and the result is shown in Figure 4.7. This test used the non-blocking version of MPI functions (MPI.Isend and MPI.Irecv).

In summary, for these two microbenchmarks, our implementation performs similar to the ch3:sock in both latency and bi-directional tests. However, the TCP implementation seems to perform slightly better when the message size is small, as both ch3:sctp and ch3:hybrid suffer slightly higher latency. Overall, the difference is fairly insignificant and is within 5% of each other. Since our hybrid design is based on ch3:sctp, our extensions to the design does impose some minor overhead with all of the integration described in Section 3.4, but
we believe this is mainly due to an additional stage being added to the progress engine. Notice there are drops in both graphs at around 64 Kbytes, which is due to the rendezvous protocol taking place for large messages.

4.4.2 Real applications

In this section, we conducted a series of macrobenchmarks that test the performance between ch3:sctp, ch3:hybrid and ch3:sock. We used the NAS Benchmarks (NPB version 3.2) [15]. These benchmarks approximate the performance that can be expected from a portable parallel application. The suite of benchmarks consists of eight programs written in Fortran, and we will be eval-
Chapter 4. Evaluation

Figure 4.7: OSU Bi-directional test (More is better)

As shown in Figure 4.8, MPICH2-ch3:sctp is fairly competitive to the TCP channel. In general, TCP performs slightly better than ch3:sctp and the difference is typically within 25% of each other. Our hybrid design does impose some minor overhead to ch3:sctp which is within 5%. Surprisingly, there are also tests in which our implementations perform better (e.g., MG and IS). In

\[^8\] The remaining two tests, FT (Fourier Transform) and CG (Conjugate Gradient), were having compilation problems with the GNU Fortran 77 compiler.
Figure 4.8: NAS Parallel MPI Benchmarks (Less is better)

general, the NAS benchmarks in class A and B are dominated by short messages
and each test uses a specific communication topology [49] (Chain, Token Ring,
etc). For instance, the average MPI message size for the class A BT test is only
957 bytes. In addition to the small message size, most of the tests are not com­
munication intensive. These results do demonstrate that our implementation
does not handle short messages as well as TCP. This should be investigated
further to determine whether this is due to SCTP or an artifact of our design.
4.5 Summary

Our current software iWARP has a performance issue when message sizes increase to Mbytes range and this needs to be investigated further. The possible shortcomings has been listed in Section 3.3.6 and 3.4.5. Nonetheless, we have shown by designing the middleware to handle independent MPI communication primitives with different components, the application efficiency can be improved.

Over the course of our experiments, the performance of SCTP often varies. In part, this is because the stack is continually under development and we always used the latest version for the most complete features. Hence, it is hard to quantitatively measure the performance difference between TCP and SCTP. We have encountered phases in which the stack was undergoing some extensive code restructuring. There were also times when errors were discovered that needed to be fixed and we would work with the stack developers to sort out the issues. Nonetheless, we have highlighted the implementation limitations of our design in Section 3.2.6. On the contrary, TCP has been extensively tested and optimized. Overall, we are fairly satisfied with our ch3:sctp implementation as it can support most of the MPI-1/2 features and is competitive with TCP.
Chapter 5

Related Work

In this chapter, we discuss various projects and technologies that share some of the same concepts as our hybrid design.

5.1 Hybrid Design Related

Open MPI\(^1\) is another popular open source MPI implementation that is developed and maintained by a consortium of academic, research, and industry partners. Similar to MPICH2, its design consists of a unique layered approach to separate the MPI semantics and message delivery. As a result, it can support different types of network devices during runtime. One unique feature about Open MPI is its ability to selectively bind to multiple different network components at runtime, which is different to MPICH2 in which network device information is determined during configuration process of installation. This allows Open MPI to schedule fragmented messages across multiple heterogeneous network devices [54]. From a component point of view, the lowest module for message transferring is the Byte Transferring Layer (BTL) and each network device has its own implementation. A BTL component is attached to a BTL management layer (BML) that is managed by the PML (point-to-point management layer) and these are all part of the MPI Component Architecture (MCA) in Open MPI. The BTL itself serves as a byte mover and has neither the knowl-

\(^1\)Open MPI project site: http://www.open-mpi.org/
edge of the types of message it is transmitting nor the MPI functions. This is different from our hybrid design in which we define a boundary between MPI function requirements and data striping. We are also utilizing SCTP as much as we can to defer some of the complexity in the MPI middleware transport layer such as connection management and multirail support for both performance and fault tolerance. In addition, the modular design with dynamic binding in Open MPI allows a MPI program to be compiled once regardless of changes to the middleware.

MVAPICH2 is a RDMA device extension to the MPICH2 implementation from the Ohio State University [32]. This implementation supports all major RDMA capable devices such as InfiniBand, and iWARP via various RDMA verbs. The extension itself is designed to optimize MPI operations for RDMA capable devices in order to deliver performance, scalability, fault tolerance and multirail support. It also includes several features like: RDMA fast path [31], utilizing RDMA operations for small message transfer, and RDMA-based one-side operations [27]. Moreover, it has been used in many high performance cluster environments.

Hybrid design often refers to the combined usage of MPI and OpenMP [43]. OpenMP is another popular parallel programming model that supports shared memory multiprocessing and it is an industrial standard that is being promoted by Intel for their multi-core architecture. It is based on the idea of multithreading that enables the runtime environment to maximize parallelism by allocating threads to different processors. A set of preprocessor directives are inserted within the code to indicate the portion that should be executed in parallel and translated by a compiler to produce an explicitly parallel code. This is completely different to MPI in which the user needs to be aware of message passing in order to exploit parallelism. The limitation with OpenMP is often

\footnote{project website: http://mvapich.cse.ohio-state.edu/}
its scalability imposed by the memory architecture and it only performs well in a shared memory multiprocessor environment. OpenMP uses preprocessor directives allowing incremental parallelism to only a portion of the code and the majority of the complexity is hidden from the users. On the other hand, MPI is shown to be efficient in exploiting parallelism across systems. However, the performance of this kind of hybrid design has been shown to be highly dependent on the programming models and hardware architectures such as vector type systems [6, 17].

5.2 CMT related

The CMT extension is also related to projects such as NewMadeleine [9], MVAPICH [32], and Open MPI that support multihomed or multirailed systems in the middleware. However, these projects focus on low latency network architectures such as Infiniband, and not just IP networks. NewMadeleine and Open MPI both support heterogeneous networks; CMT can be used to combine all IP interfaces, which in turn can be used in combination with these heterogeneous network solutions. MuniCluster [33] and RI2N/UDP [37] support multirailing for IP networks at the application layer and use UDP at the transport layer. Our approach is significantly different in that CMT supports multihoming (or multirailing) at the transport layer, as part of the kernel, rather than at the application layer in user-space. These projects often require the network conditions such as latency and bandwidth to be determined statically before hand and the tests are primarily done with basic networking benchmark programs. The result is then statically feed in as an input to the systems for making scheduling decisions. On the other hand, CMT can determine the bandwidth and latency characteristics of the network dynamically and adaptively as part of the transport protocol.
Fault tolerance is an important issue in commodity cluster environments, where network failure due to switches or software (mis)configuration do occur. SCTP has a mechanism for automatic failover in the case of link or path failure. Similar mechanisms have been implemented in MPI middleware [51], but these are unnecessary for SCTP-based middleware. SCTP uses special control chunks which act as heartbeat probes to test network reachability. Heartbeat frequency is user-controllable, and can be tuned according to expectations for a given network. However, at the currently recommended setting [14], it takes approximately 15 seconds to failover—a long time for an MPI application. Since CMT sends data to all destinations concurrently, a CMT sender has more information about all network paths, and can leverage this information to detect network failures sooner. CMT is thus able to reduce network path failure detection time to one second [35]. This improvement with CMT is significant, and future research will investigate if this time can be further reduced for cluster environments.

5.3 RDMA related

An interesting extension for RDMA capable devices is the Socket Direct Protocol (SDP) defined by the Software Working Group (SWG) [25]. It was defined to serve two purposes: to maintain backward compatibility for socket based applications and to deliver the performance of underlying RDMA devices. This extension allows socket applications to run on top of RDMA devices transparently by exporting a pseudo socket-like interface (Figure 5.1).

SDP provides high performance sockets, thus allowing applications to utilize the network capabilities without any modifications. The current SDP standard specifies implementation details for both InfiniBand and iWARP. However, these two implementations differ significantly in their connection setup. Internally,
SDP is responsible for managing all the user level buffers and flow control in order to exploit possible use of zero copying and this is done by exchanging SDP level control messages. Moreover, it currently supports only the TCP/IP style sockets and provides all the necessary features and mapping for managing TCP port space, IP addressing, the Connecting/Accepting model, Out-of-Band data, and common socket options. The current SDP design often performs copy-based communication for small messages and zero-copy communication for large messages. SDP-iWARP holds more promise as it traverses through the traditional IP infrastructure and thus provides the support for wide area networks. Many studies have shown that SDP can provide significant speed improvements in various workloads [11]. A similar approach from Myricom is the Socket-GM Module [34] which is a middleware layer that mimics sockets semantics and replaces the traditional Ethernet protocol to allow for low latency, high speed data transfers.
One other interesting device is called Etherfabric, a network fabric being developed by SolarFlare [5]. This device serves as an Ethernet accelerator by using "onload" technology that runs transport layer protocols within user space to avoid unnecessary kernel crossing and thus avoids copying and reduces latency. This technology has been shown to be able to improve latency substantially over traditional Ethernet network adapters with a kernel based TCP/IP stack. Socket based programs can run without any modification and this is done by hijacking the linking process during runtime to a special set of Etherfabric libraries. Each process will be exposed to a virtual interface that is directly connected to the user level transport layer protocol (Currently only support TCP and UDP).

![TCP Latency test with NetPIPE](image.png)

Figure 5.2: TCP Latency test with NetPIPE (Less is better)
We had some experience with the 1 GbE Ethernet Accelerator as we were trying to port SCTP to run over the device. Figure 5.2 shows an output of running NetPIPE over TCP\(^3\) with/without Ethernet Acceleration. We are only showing output of message sizes from 2 bytes to 8 Kbytes since latency is important for small messages. The result has shown that Ethernet Accelerator can improve the latency by almost 40% for short messages. Compatibility with existing Ethernet devices is the main advantage to its design. This is a completely opposite approach to iWARP and InfiniBand as it tries to redesign and improve the existing Ethernet infrastructure.

\(^3\)NetPIPE website: [http://www.scl.ameslab.gov/netpipe](http://www.scl.ameslab.gov/netpipe)
Chapter 6

Future work and Conclusion

In this thesis, we have implemented a hybrid design of MPI middleware as a proof of concept model. Our model exploits one type of device, namely Ethernet based adapters and is shown in only one kind of functional decomposition. In the future, we are interested in exploring more functional mapping with respect to more network devices and their capabilities. For instance, MPI collectives and MPI long message transferring are both good candidates.

The hybrid design also proposes future research in designing the MPI middleware with multiple internal threads. Though a thread model for MPI middleware may be overly complicated, it may allow more overlapping between the internal components and thus improve latency.

We have also shown how SCTP can be used to support both socket and iWARP semantics. In the future, we would like to explore the fault tolerant aspect of the protocol with respect to MPI. For instance, we can monitor the association's status via notifications. This might help middleware to make it more reactive to events indicating abnormal link conditions.

We are currently exploring the possible use of CMT to improve the latency for MPI messages [13]. CMT was originally designed to improve bandwidth. In order to reduce latency, this would require SCTP to make more dynamic scheduling decisions other than round robin across available links. Moreover, we also want CMT to be dynamic and adaptive in a way that it can switch between these two modes of operations during runtime.
Even though traditional Ethernet devices along with transport layer protocols have been criticized for lack of performance, we have explored different techniques to improve its overall efficiency from many areas: application layer, transport layer and hardware layer with multirailing support. Moreover, there exist roadmaps [7] in manufacturing for 40 and 100 GbE devices that will be targeting the backbone infrastructure in large scale data centers such as the type of clusters deployed at Google. With its low cost, availability, and the deployment of IPv6, we believe that Ethernet based devices will continue to evolve and become more prevalent in the near future.
Bibliography


    solarflare.com.

[6] Laksono Adhianto and Barbara Chapman. Performance Modeling of Com-
    munication and Computation in Hybrid MPI and OpenMP Applications.
    The 12th International Conference on Parallel and Distributed Systems
    (ICPADS), 2006.

[7] Ethernet Alliance. Overview of Requirements and Applications for 40Gi-
    gigabit and 100 Gigabit Ethernet. http://www.ethernetalliance.org/
    technology/white_papers/Overview_and%20Applications2.pdf.


[9] Olivier Aumage, Elisabeth Brunet, Guillaume Mercier, and Raymond
    Namyst. High-Performance Multi-Rail Support with the NewMadeleine


[47] Michael Scharf and Sebastian Kiesel. Head-of-line Blocking in TCP and
SCTP: Analysis and Measurements. In GLOBECOM ’06. IEEE, vol., no.,
pp.1-5, November 2006.

[48] Randall R. Stewart and Qiaobing Xie. Stream Control Transmission Pro-
tocol (SCTP): A Reference Guide. Addison-Wesley Longman Publishing

[49] Jaspal Subhlok, Shreenivasa Venkataramaiah, and Amitoj Singh. Charac-
terizing NAS benchmark performance on shared heterogeneous networks. In
IPDPS ’02: Proceedings of the 16th International Parallel and Distributed
Processing Symposium, page 91, Washington, DC, USA, 2002. IEEE Com-
puter Society.

[50] University of Mannheim, University of Tennessee, NERSC/LBNL. Top 500

[51] Abhinav Vishnu, Prachi Gupta, Amith R. Mamidala, and Dhabaleswar K.
Panda. A software based approach for providing network fault tolerance
in clusters with uDAPL interface: MPI level design and performance eval-
uation. In SC ’06: Proceedings of the 2006 ACM/IEEE conference on

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

G.E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J.J. Dongarra, J.M.
Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine. Open
MPI's TEG point-to-point communications methodology: Comparison to