

A Hybrid MPI Design using SCTP and iWARP

Mike Tsai and Brad Penoff and Alan Wagner

Department of Computer Science

University of British Columbia

Vancouver, British Columbia

Email: {myct,penoff,wagner}@cs.ubc.ca

Abstract— Remote Direct Memory Access (RDMA) and point-to-point network fabrics both have their own advantages. MPI middleware implementations typically use one or the other, however, the appearance of the Internet Wide Area RDMA Protocol (iWARP), RDMA over IP, and protocol off-load devices introduces the opportunity to use a hybrid design for MPI middleware that uses both iWARP and a transport protocol directly. We explore the design of a new MPICH2 channel device based on iWARP and the Stream Control Transmission Protocol (SCTP) that uses SCTP for all point-to-point MPI routines and iWARP for all remote memory access routines (i.e., one-sided communication).

The design extends the Ohio Supercomputer Center software-based iWARP stack and our MPICH2 SCTP-based channel device. The hybrid channel device aligns the semantics of the MPI routine with the underlying protocol that best supports the routine and also allows the MPI API to exploit the potential performance benefits of the underlying hardware more directly. We describe the design and issues related to the progress engine design and connection setup. We demonstrate how to implement iWARP over SCTP rather than TCP and discuss its advantages and disadvantages. We are not aware of any other software implementations of iWARP over SCTP, nor MPI middleware that uses both iWARP verbs and the SCTP API.

I. INTRODUCTION

MPI added support for shared-memory programming in MPI-2 with the introduction of the Remote Memory Access (RMA) routines for one-sided communication. RMA semantics differ from the point-to-point messaging semantics in that processes can read and write remote memory without having to synchronize the sender and receiver. A motivation for adding the RMA routines was to take advantage of hardware devices supporting Remote Direct Memory Access (RDMA). Currently, InfiniBand is the most widely used RDMA-based standard. The success of RDMA spurred the IETF standardization of RDMA over IP, i.e., Internet Wide Area RDMA Protocol (iWARP). iWARP and the recent appearance of intelligent network interface cards supporting TCP off-load and iWARP, introduce new opportunities for using IP transport protocols in the MPI middleware.

For RDMA over IP, iWARP verbs can be used by an iWARP stack defined over one of two reliable transport protocols, namely TCP or the Stream Control Transmission Protocol (SCTP). As a result, both iWARP verbs and the underlying transport protocol can be used directly inside the middleware. We explore the idea of using both iWARP verbs and SCTP directly in a hybrid middleware design, where the semantics of the MPI routines are matched with the underlying protocol.

We designed and implemented an MPICH2 channel (called `ch3:hybrid`) that uses iWARP verbs to use RDMA for RMA routines and SCTP directly for point-to-point communication; this is shown in Figure 1.

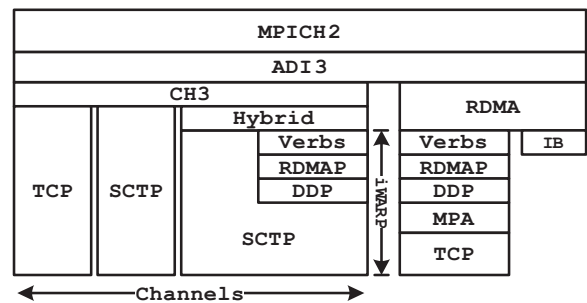


Fig. 1. Protocol Diagram for MPI, iWARP and SCTP

Hybrid MPI designs using several protocols have been investigated for InfiniBand [1]. Our focus is on IP-based protocols and in particular the use of iWARP and SCTP. There are TCP-based iWARP devices available, but those MPI implementations use the iWARP RDMA mechanisms for everything as an alternative to using the TCP (or SCTP) sockets API [2]. Thus designs are “all or nothing” in that all communication is implemented over a point-to-point protocol or an RDMA protocol. Although dependent on the underlying hardware, both mechanisms have their advantages and that ultimately depends on the application [3]. Rather than choose between one or the other, matching the MPI function to the appropriate underlying protocol aligns the programmer’s view of the semantics of the operation with its implementation. This does not preclude later extending our design to other aspects of the middleware, for example use of RDMA for long messages; here we focus only on using iWARP for the RMA routines.

A second part of the design was to use SCTP as the underlying transport layer protocol for both parts of our hybrid channel device. We believe that SCTP has several advantages for MPI and iWARP. SCTP is message-based and reliable plus has built-in support for multi-homing and multi-streaming which have been shown to be beneficial for MPI [4]. For iWARP, SCTP simplifies the implementation since the Direct Data Placement (DDP) layer [5] in the iWARP stack requires a reliable message-based transport protocol to preserve the self-contained and self-describing property of each DDP segment.

Although TCP is the most common choice for iWARP, it requires an additional Marker PDU Aligned (MPA) layer [6] for message framing whose presence can lead to TCP conformance problems [7]. By using SCTP as the transport layer protocol for iWARP, difficulties related to the MPA layer are eliminated and SCTP features for multi-streaming and multi-homing can be exploited to provide scalability and the necessary levels of fault tolerance, respectively [8].

For bandwidth-intensive applications, SCTP with the Concurrent Multi-path Transfer (CMT) extension can provide message striping across the available links at the transport layer [9]. We have shown how CMT can be beneficial for large message transfer and additional fault tolerance within an SCTP-based MPI middleware [10]. Research has shown that for the MPA, the placement of message markers on a TCP stream is difficult for a complete iWARP stack that supports out-of-order packet arrival and multi-path systems [11]. We believe by using SCTP for iWARP, multi-path support can be deployed with very little cost, since support for it already exists in the protocol.

The main disadvantage of using SCTP is that it is still a relatively new protocol. Consequently, the stacks are not as mature as TCP on all operating systems, and thus there are fewer applications using SCTP. However, all MPI programs can be run using SCTP as there is an SCTP channel device in MPICH2 as well as an SCTP Byte Transport Layer within the Open MPI main trunk. SCTP is available for most operating systems, including Linux, and, although the FreeBSD stack is the most mature and feature-rich, the Linux stack continues to mature.

SCTP is only one of two transport protocols to be standardized by the IETF in the last 30 years. Since it is relatively new, there is still the opportunity to explore new features as well as test and experiment with its existing features. In particular, although an iWARP stack using SCTP has been defined, we are not aware of any work that reports on implementations using it.

The main contributions of this work are to support SCTP in iWARP and to explore the issues that arise in a hybrid design where a decomposition at the MPI library-level is implemented such that the point-to-point and RMA operations 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 Supercomputer Center's software implementation of iWARP [12] (OSC-iWARP [13]). As is, OSC-iWARP only supports TCP but we extend the implementation to use SCTP and threads.

This paper is laid out in the following structure. In Section II, our extensions to OSC-iWARP are described as well as how these extensions are then used to implement the MPI RMA functions in our hybrid MPI design. Section III then describes some of the issues encountered with this `ch3:hybrid` design. An experimental evaluation of this hybrid design is presented in Section IV, and conclusions are given in Section V.

II. DESIGN OF THE HYBRID CHANNEL DEVICE

The hybrid design (`ch3:hybrid`) is based on our MPICH2 SCTP CH3 channel device (`ch3:sctp`) and the user-level OSC-iWARP stack. The OSC's iWARP stack is implemented with respect to RFC specifications ([14], [15], [16]) and is compatible with iWARP-capable hardware [12]. Our SCTP CH3 implementation is based on point-to-point messaging and uses the SCTP one-to-many style (SCTP UDP-style) socket and takes advantage of SCTP's multi-streaming and multi-homing features. A description of our SCTP-based MPI middleware is given in [4].

For the hybrid design, we implemented a channel device (`ch3:hybrid`) that uses SCTP for the point-to-point MPI operations and iWARP (over SCTP) for the MPI RMA operations. The target of RMA operations in MPI is called a *window object*, a distributed shared memory space formed from the local memory from each participating process. We implemented the following MPI-2 routines:

- **MPI.Win.create/free:** Creates and frees a window, the local portion of the memory available for RMA. This is a collective over the communicator and returns a window object.
- **MPI.Get/Put:** Gets and puts data into the window.
- **MPI.Win.fence:** Used to synchronize the windows.

There are other routines like `MPI.Win.start/complete` and `MPI.Win.post/wait`, which allow further synchronization control over the window. The routines implemented were sufficient to demonstrate how different communication and internal code paths can be used to explicitly separate MPI-1's point-to-point operations from MPI-2's one-sided routines (RMA).

In the remainder of this Section, we describe the mapping of these operations onto iWARP.

A. iWARP - Verbs, RDMA, and DDP

The iWARP standard defines a set of *verbs* that describe the RDMA operations provided by iWARP. The verbs interface is implementation-specific and defined by the uppermost RDMA layer in the OSC-iWARP stack. The RDMA provides iWARP endpoints, called Queue Pairs (QPs) that are used to read and write remote memory. In addition to the queue pairs, associated with each QP is a pair of Completion Queues (CQ), which are for notifications. The QP is for reading and writing data while the CQ can be used to determine when a remote read or write has completed. iWARP supports memory-based operations (*RDMA.Write/Read*) as well as channel-based operations (*RDMA.Send/Recv*), i.e., message passing.

Window objects are implemented as a collection of QPs between all the processes that are part of the communicator. We extended the OSC connection framework to do out-of-band connection setup where we only require registering a file descriptor with iWARP; details of connection setup are given in Section III-B. The connection setup required the creation of QPs and to register them with iWARP is done by `MPI.Win.create`. Once the window object exists,

MPI_Get/Put operations on a specific window in MPI are mapped onto iWARP RDMA_Read/Write verbs acting on the associated QPs.

Although RDMAP implements the verbs interface to iWARP, it is a relatively thin layer and relies mostly on the DDP. The DDP layer is a self-describing reliable data packet transfer protocol that allows for the direct assembly of the message in user buffers, which makes receive-side zero copy possible when the appropriate hardware support is present.

The DDP supports two message types, tagged (for RDMA read and write) and untagged (for standard send and receive). The tagged message type allows a DDP segment to be directly copied into the designated user buffer. The receiver’s destination buffer is identified by its tag, called the *steering tag*, which must be setup prior to any RDMA operations. The steering tag and message offset information is specified in the DDP header and used by the remote side to place the data directly into the receive-side buffer. In the upper layer protocol, RDMAP, DDP messages are created as a result of work requests, which require a request ID and other buffer information in the form of a Scatter Gather Entry. One can also optionally request a completion signal, where polling for the signal is performed on the completion queue (CQ); for scalability, a given CQ can be shared among many QPs.

The channel-based send/receive verbs, which use DDP untagged message types, are used internally inside the iWARP stack to communicate CQ events between iWARP end-points. The untagged messages require a pre-allocated buffer on the remote side to accept messages and use the same copy-in/copy-out semantics of message passing.

B. Mapping RMA Functions to iWARP

1) *MPI_Win_create/free*: Before remote memory operations can take place, a window object that consists of all the distributed window information must be created with *MPI_Win_create*. As well, in order to use DDP’s tagged model, local memory associated with the window 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. The information typically consists of the buffer location, length, and offset. In our case, we include the steering tag for the local memory window, which is used in the implementation of get and put. At the end of the process, each participant has the knowledge of others’ memory windows, which taken together forms the MPI window object, i.e., shared distributed memory object.

Freeing a memory window object is also a collective operation, *MPI_Win_free*. During the process of destroying a window object, all pending RMA operations must be completed before exiting. Each participant also needs to unregister its local memory window with the iWARP layer.

2) *MPI_Win_fence*: MPICH2 adopted a delayed approach to the window synchronization call, *MPI_Win_fence*; it queues up all the put/get operations until a later fence call occurs. This method is more latency tolerant than using simple

barrier calls and the delayed approach is more appropriate for TCP and SCTP where latency is relatively high [17].

In the first stage of the fence operation, an MPI process gathers 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 internally for allocating MPI resources and for fence termination. In the second stage, it processes all of the queued RMA 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 MPI progress engine to wait until all of the RMA operations are completed and the allocated resources are freed accordingly.

3) *MPI_Put*: As explained, we used MPICH2’s delayed put where the call to *MPI_Put* simply bundles an envelope with the data, which is then transmitted during a later call to *MPI_Win_fence*. The envelope 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, we decoupled the envelope from the data and sent it later as an untagged DDP message.

Figure 2 shows the messaging that occurs during the put operation. The figure shows the messages transmitted from the middleware to an iWARP thread on one side to the corresponding software on the receiver side. As shown in

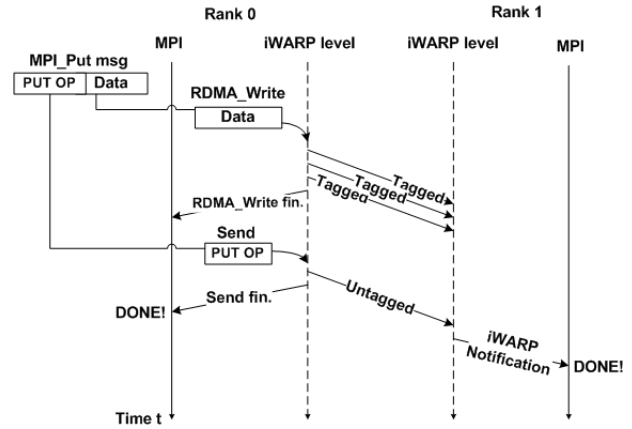


Fig. 2. MPI_Put via RDMA_Write during the fence call.

Figure 2, the MPI middleware first makes an *RDMA_Write* request to iWARP. We use the information already registered with iWARP to retrieve the steering tag for the write request. Using the steering tag the DDP layer transmits the data, as one or more segments, and deposits the data in the memory associated with the steering tag of the target. Notice there is no interaction with the MPI middleware during this part of the operation. Once the data is sent, we send the envelope to the target, which serves as a notification that its memory has been updated and is ready for access.

4) *MPI_Get*: Reading data from a window object is done by issuing an *RDMA_Read*. There is a minor complication due to the requirement that the local buffer that is the data sink

for the incoming data may be a different buffer than its local memory window. Therefore, before we issue an *RDMA_Read* request, we first register the local buffer memory with iWARP whenever it is not the same as the local window.

Messaging for the *MPI_Get* operation is shown in Figure 3. The iWARP stack handles an *RDMA_Read* by first issuing an

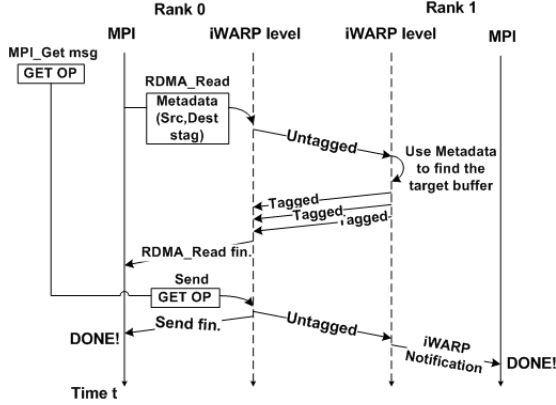


Fig. 3. *MPI_Get* via *RDMA_Read* during fence call

untagged message of the request’s metadata to the target. The target endpoint’s iWARP stack then sends, using the steering tag obtained from the source, one or more tagged segments which are deposited directly into source memory. As with *MPI_Put*, after the completion of the *RDMA_Read* request, we send the MPI envelope as an untagged message to the target to serve as a completion notice.

III. DESIGN ISSUES

In the following section, we discuss the major design issues that arose in the implementation of our *ch3:hybrid* channel device. We discuss possible trade-offs, enhancements and limitations to the current design. The following issues are discussed:

- Creation of a second thread to *simulate* the execution of the iWARP device and integration of that additional thread into the progress engine.
- Changes needed to support static and dynamic out-of-band connection management.
- SCTP-related features of the design.

Throughout, we discuss some of the limitations and possible extensions to the design.

A. Asynchronous Message Progression

The original *OSC-iWARP* was implemented as a single thread where the API exported by the verbs layer has functions to create a connection, manage QPs, and write work requests and poll for their completion. Except for read requests (i.e., *RDMA_recv/read*), all calls are blocking. For read requests, there are functions in the API to poll for completion as well as an advance function to progress requests. In many respects the single-threaded implementation is similar to its own *MPICH2* channel device.

In our hybrid channel device, we implemented iWARP as a separate thread inside *MPICH2*. The second thread essentially *simulates* an iWARP-capable network device with the ability to independently progress work requests. This approach minimized changes to the iWARP stack, since it keeps the original single-threaded *OSC-iWARP* design. As well, it avoids having to progress iWARP operations inside the MPI middleware, which may make it easier to replace iWARP inside the hybrid device with actual hardware. Because iWARP runs as a separate thread, we needed to add mutexes to protect access to data structures, such as the completion queue, that are shared between the two threads (iWARP and MPI). The iWARP thread is created in the RDMAP layer to progress Verbs layer work requests. The interaction between the main and RDMAP thread is handled in a producer and consumer manner.

In the case of read requests, which are non-blocking in *OSC-iWARP*, we decided to poll for their completion inside the progress engine of the hybrid device. If we were to handle this inside the iWARP stack, then it is possible for the socket buffer to contain data corresponding to a read request that has not yet been posted. As with unexpected messages in the MPI middleware, since the iWARP stack can not place the data, it needs to internally buffer the message. This would have required adding a more complicated buffering mechanism inside the iWARP stack and potentially performs redundant work when integrated with the MPI middleware. Polling for the completion of read requests in the hybrid device avoided these complications and still allowed work requests to proceed asynchronously.

1) *ch3:hybrid Progress Engine*: Figure 4 shows the overall design of the the progress engine of our hybrid channel device. The state machine implemented by the progress engine

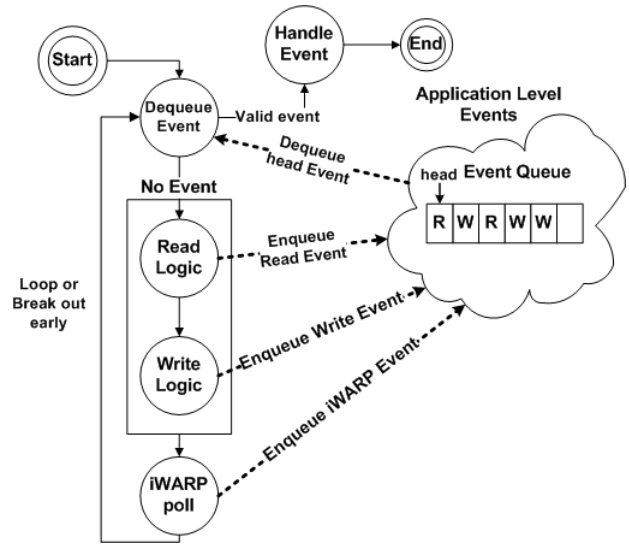


Fig. 4. *ch3:hybrid* Progress Engine Design

responds to events associated with the virtual connection (VC) state of the endpoints for *MPI_COMM_WORLD* and the state

of outstanding MPI message requests. In order to map from socket-level events for SCTP and iWARP 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 first two stages in the pipeline, read block followed by the write block, are responsible for reading incoming messages from the socket buffer and transmitting pending send requests; these are the only two blocks in `ch3:sctp`. The difference in `ch3:hybrid` is that only non-RMA functions use these two original blocks. The Read logic performs `sctp_recvmmsg` 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.

For the RMA functions, a final stage was added to the pipeline in `ch3:hybrid` to support iWARP. The stage maps an iWARP completion entry to the application level event, and also handles the new application level events. The iWARP stage basically performs an iWARP level poll and dequeues a completion entry. In our implementation, all send and receive QPs share one 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; the table is populated during connection setup (described in Section III-B).

The addition of another stage to the progress engine adds to the overall overhead of the progress engine. We try to minimize the impact as much as possible. One technique to avoid much of the overhead is to only execute the stage when there are active RMA operations, thus incurring the extra cost only when necessary. One disadvantage of the current method of polling is that the `OSC-iWARP` verbs do not allow batch polling; each execution of iWARP poll results in the return of at most one iWARP level completion entry. However, it is possible to extend the OSC verbs API for batch polling or expose the CQ structure to the application to avoid this limitation.

2) *Alternative Designs*: One alternative design would be to not use an additional thread and keep the single-threaded progress engine. The blocking behavior of `OSC-iWARP` calls resulted in blocking socket calls to TCP (or in our case SCTP). This is not acceptable for MPI since it implies that the middleware stalls, waiting indefinitely on the socket call to return, and preventing it from progressing other outstanding

message requests. It is possible to change these calls to be non-blocking but this requires recording and managing the state associated with these non-blocking socket calls. We avoid these complications by keeping iWARP as a separate thread.

A second alternative is to have a multi-threaded design with a separate thread for read and one for write [12]. One can consider adding even more threads to handle each independent queue pair. Since a single progress thread handles one QP at a time, they do not progress independently and it is possible for one QP to delay others. Although not realistic for TCP, since it may require a socket for each separate QP, it may be feasible with SCTP using one-to-many style sockets. We did not have time to explore this alternative. It would have required extensive changes to the iWARP stack as well, to use one-to-many sockets; this is discussed further in Section III-C.

B. Connection Management

`OSC-iWARP` does not provide non-blocking connection management routines so we extended its API and implemented our own connection management framework within `ch3:hybrid`. Our extension adds an API call (`iwarp_qp_register_fd`) to the verbs to directly register a file descriptor regardless of the transport layer protocol; 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, which is exactly what we did in `ch3:hybrid`.

An alternative to the approach here would be to use the RDMA-CM interface defined by the Open Fabrics Alliance [18]. 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 are different than `OSC-iWARP`'s.

Each MPI process holds an RDMA capable Network Interface Card (NIC) handle to the simulated hardware exposed by the software iWARP stack. All of the components in `OSC-iWARP` are allocated with reference to the NIC handle. In order for iWARP capable endpoints to communicate with each other, at least one QP for a pair of endpoints must be allocated.

Recall, the goal of our `ch3:hybrid` design was to use iWARP verbs for the RMA routines, and to use the SCTP sockets API for the point-to-point MPI operations. In order to prevent the integrated `OSC-iWARP` thread from interfering with our original `ch3:sctp` device, we created another SCTP one-to-many style socket within our hybrid that was specifically used for handling new iWARP QP connections. We considered both static and dynamic techniques for managing the iWARP QPs.

1) *Static Connection Management*: In the static case we allocate a QP for each pair of endpoints during initialization. This requires allocating $N(N-1)$ file descriptors for the QPs and is acceptable for small collections of processes as long

as the initialization cost is not too significant. This is also the technique used in many MPI-RDMA implementations [2].

We took advantage of SCTP’s “peel-off” ability to simplify the initialization of the QPs. As mentioned, SCTP supports both UDP and TCP style sockets and it allows these two sockets to connect and communicate. Our `connect_all` procedure starts by creating an SCTP one-to-many style socket and the connection initiator sends a message to the target’s SCTP one-to-many listener consisting of its rank and the information about the locally allocated 32 KBytes of memory assigned for the QP. Once all messages have been exchanged, by the one-to-many style sockets, we use `sctp_peeloff` to peel off the association (connection) for the initiator to obtain a separate SCTP TCP-style socket for each iWARP QP. This operation does not require any coordination between endpoints, avoiding an explicit `connect` and `accept`. Hence, we can directly register this newly peeled off association with the iWARP layer and with our new connection extension call. The end result is $N(N - 1)$ TCP-style sockets for use with iWARP (i.e., $N - 1$ sockets on each of the N processes).

The static approach is used only for initializing the iWARP sockets. The connection setup in `ch3:hybrid` for the point-to-point communication is dynamic; this is the same technique used by our original `ch3:sctp` device.

2) *Dynamic Connection Management*: We also explored dynamic connection management where QPs are allocated by the middleware on demand. This approach, similar to that in `ch3:sctp`, reduces the number of file descriptors (QPs) allocated, however, it increases the communication cost for the first RMA data transfer.

The first delayed RMA operation during the fence routine sends a connection packet to the one-to-many SCTP socket specially allocated for iWARP connections. In SCTP, the association is formed when the first message arrives. SCTP allocates all the necessary structures and performs a 4-way handshake to form a reliable connection. We can not simply send a connection packet and then immediately peel-off the association because SCTP may not have finished creating the association. Rather than re-trying, we peel-off only after we have received a connection acknowledgement. In the case of simultaneous connection requests, as shown in Figure 5, when

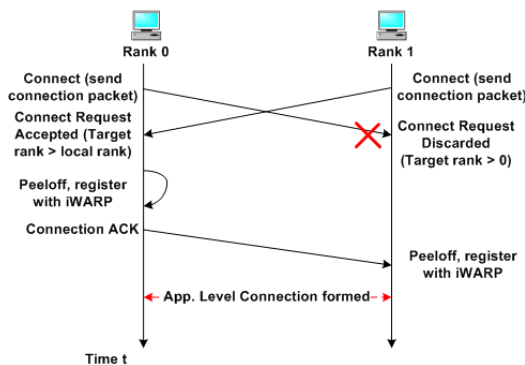


Fig. 5. Dynamic Connection Setup for QPs

a connection request has already occurred, we use their MPI globally unique rank in `MPI_COMM_WORLD` to determine which side sends the acknowledgement. Since we direct iWARP connection traffic to a separate SCTP socket, we need to periodically poll that socket for connection requests; this code block is added to the progress engine in `ch3:hybrid`.

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 easy to implement and the MPI messages can simply be posted to the iWARP layer. Overall, the static approach can manage fewer states by sacrificing some resources during initialization.

C. SCTP Related Issues and Limitations

One of the main benefits of using SCTP rather than TCP as the lower level protocol for DDP is that it already supports many of the criteria required by DDP. SCTP is message oriented and DDP segments can be easily encapsulated into SCTP messages.

In the case of TCP, because it is stream-oriented, to have a feature-complete iWARP stack it is necessary to insert an MPA layer between DDP and TCP in order to insert markers to the iWARP header into the TCP data every 512 bytes to support out-of-order packet arrival in the event of middlebox fragmentation; the presence of the markers has been shown to impact in-order performance [11]. As a result of its complexity, the MPA layer within the software stack contains the majority of the code in the `OSC-iWARP` implementation. Developers have implemented the MPA as a separate layer on top of TCP/IP, but its performance is suboptimal because of additional copying from layer to layer. To avoid copying, another strategy is to insert the MPA changes directly into the TCP/IP stack. While this improves the overall performance of MPA-TCP/IP, it also can result in compatibility problems between TCP stacks [7]. Using SCTP for iWARP eliminates the complications presented by the MPA.

In addition, using SCTP gives the application layer protocol more flexibility in choosing the segment size and allows us to remove the two byte DDP segment length field. This field is necessary only because of the MPA layer and is already present in SCTP. As well, SCTP already has a CRC32c field and this can be eliminated as well from the DDP header.

There is an IETF RFC by Bestler and Stewart [19]¹ that describes an adaptation layer for DDP and SCTP. The adaptation layer is a very thin layer that takes advantage of SCTP’s extensibility. The RFC defines a DDP chunk type within SCTP for connection setup and data transfer and restricts the connection to carry only DDP segments using SCTP’s

¹We are unaware of a publicly available SCTP stack that implements this RFC.

unordered message delivery. An adaptation layer is needed when the connection setup is done within DDP. It was not necessary in our case since there was a priori knowledge that each endpoint could perform its connection setup by the MPI middleware’s use of our extended `OSC-iWARP`. The DDP adaptation layer for Sctp does provide a more general overall solution for using DDP.

One simplification made in the design was that we disabled the iWARP negotiation stage that normally would occur during connection setup. The negotiation communication between two iWARP capable endpoints is in order to exchange key pairs for security, network hardware information, MPA marker locations (in the case of TCP), along with other information.

We based our software iWARP implementation on the user-level version of the `OSC-iWARP` stack. There is also a Linux kernel version in the same distribution with reportedly better performance [20], however the user-level implementation is portable to other OSes. Second, since our modified `OSC-iWARP` stack executes on top of Sctp, it is only the verbs, RDMAP and DDP layers that execute in user-level; TCP requires the MPA to do a lot of work that Sctp does already in the kernel, for example message framing.

A final issue relates to how we mapped the steering tag and offset to Sctp streams. Streams provide independent in-order flows between endpoints, all managed by a common set of congestion control parameters. They allow ordering constraints between messages in an association to be better specified. In order to utilize multiple Sctp streams in iWARP, we use the steering tag to define which stream to send for the tagged (RDMA) model and the message identifier for the untagged model. The OSC’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; this can be supported with Sctp using the same stream per message portion but using the `MSG_UNORDERED` message flag to decouple data placement and message order delivery, exactly as the DDP requires.

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 one-to-many 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 iWARP stack.

IV. EXPERIMENTAL EVALUATION

We evaluated the hybrid design by comparing it to the original Sctp point-to-point channel device. Since iWARP was implemented entirely in software, there is added overhead that could be eliminated with hardware-assist.

The experiments were conducted on an IBM eServer x306 cluster with four nodes, each node with a 3.2 GHz Pentium-

4 processor with 1 GBytes memory and GbE interfaces. We are using the `sctp.org` Sctp stack on FreeBSD 6.2. We focused on the performance of `MPI_Put/Get` in `ch3:hybrid` versus `ch3:sctp`. The test program in Figure 6 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 machines. We measured the average wall clock time of 10 runs to perform `MPI_Win_fence`, which is where the majority of MPI-2 one-sided time is being spent. The first observation is that the Sctp and iWARP

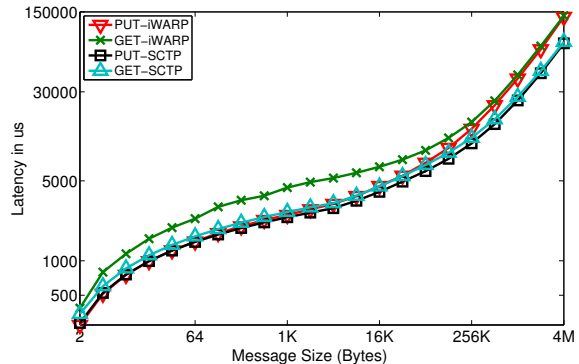


Fig. 6. `MPI_Put/MPI_Get` Latency Test for `ch3:hybrid` (iWARP) versus `ch3:sctp`(Sctp)

performance for both `MPI_Get` and `MPI_Put` share a similar trend. Moreover, our software implementation of iWARP adds an overhead ranging from 2% to 8%.

As well, notice that compared to the implementation using just Sctp, the performance of RMA operations implemented over `OSC-iWARP` degrades more rapidly as the message size increases into the MByte range. We believe that this is because the DDP layer is fragmenting messages at 16 KBytes². Even though this number is adjustable, it also depends on the send and receive buffer sizes for iWARP endpoints. One could allow the DDP sender to fragment at 64 KBytes, but the receiver may not be able to receive at the same rate.

Another reason for the difference in performance is that our `ch3:sctp` is performing non-blocking 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 requests, as was shown in Figure 4. However, it also adds overhead when going through the other progress engine stages, which with this application, are mostly 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 `MPI_Get` is typically slower than `MPI_Put` because of the additional metadata messages being exchanged.

We designed a synthetic benchmark program to test the performance with a combination of point-to-point and one-sided

²This value was set to 16 KBytes because it gives the best result in our setup.

communication. The benchmark consists of two processes sending combinations of `MPI_Isend` messages of various sizes and `MPI_Put` and `MPI_Get` to each other. Routines `MPI_Waitall/Waitany` and `MPI_Win_fence` are placed in specific places to allow overlapping between the types of communication.

Surprisingly, for this benchmark, we were able to obtain better performance with `ch3:hybrid` than `ch3:sctp` (3.8 seconds with `ch3:hybrid` compared to 4.5 seconds with `ch3:sctp`, average wall clock time with 10 runs). The reason for the better performance however is because in `ch3:hybrid`, the one-sided communication is queued separately from the point-to-point communication and thus each can progress independently from each other. Although there is overhead from adding iWARP events to the progress engine, the additional thread allows for more concurrency between the types of communication leading to improved performance for this particular benchmark since it mixes RMA calls with non-blocking point-to-point calls.

V. CONCLUSIONS

We designed and implemented a hybrid MPICH2 channel device using SCTP for point-to-point communications and iWARP over SCTP for one-sided communication. The design required several changes to the `OSC-iWARP` stack for SCTP as well as changes to the connection framework. We implemented iWARP in a separate thread and flexibly incorporated iWARP events into the `ch3:hybrid` progress engine making it easy to use any combination of SCTP and iWARP routines. The design could even be extended to include TCP or UDP as well. We explored both static and dynamic connection setup options and took advantage of SCTP-related features such as peeling off an association.

As expected, the additional protocol layers add overhead and without hardware-assist cannot exploit the copy avoidance benefit of one-sided communication. The overhead was small, on the order of 5-8%. Mixing a variety of RMA with non-blocking point-to-point MPI routines was an example that illustrated the advantage of having a second thread, where the added concurrency leads to overall better performance. One performance problem with the current design is the increased overhead for large messages, which is due to the buffer sizes and message sizes in DDP and SCTP.

There are a number of simple extensions to the current design that make it more complete. First, the design can easily be extended to all one-sided communication; use of iWARP may also be advantageous for long messages and some collectives as well. A second extension is to change the DDP layer in `OSC-iWARP` so that it can map DDP protocol segments on to streams to allow for unordered delivery on the receiving side. A third extension is to revisit connection setup using the DDP and SCTP adaptation layers to potentially allow for a more dynamic, open environment.

Even though traditional Ethernet devices along with transport layer protocols have had much higher latencies than non-Ethernet devices, that may be changing with the appearance of

10GbE and both on-load and off-load protocol stacks. Support for RDMA over Ethernet (iWARP) extends the applicability of Ethernet to shared-distributed memory type applications. To date, there are no iWARP off-load devices using SCTP and we know of no other software implementation of MPI with SCTP and iWARP. As described in Section III-C there are several advantages to iWARP over SCTP, which may be even more appropriate in a cluster or data center type environment.

REFERENCES

- [1] A. R. Mamidala, S. Narravula, A. Vishnu, G. Santhanaraman, and D. K. Panda, "On using connection-oriented vs. connection-less transport for performance and scalability of collective and one-sided operations: trade-offs and impact," in *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2007, pp. 46–54.
- [2] S. Narravula, A. R. Mamidala, A. Vishnu, G. Santhanaraman, and D. K. Panda, "High Performance MPI over iWARP: Early Experiences," in *Int'l Conference on Parallel Processing*, September 2007.
- [3] G. Shainer, "Why Compromise?" <http://www.hpcwire.com/hpc/955288.html>, October 2006.
- [4] H. Kamal, B. Penoff, and A. Wagner, "SCTP versus TCP for MPI," in *Supercomputing '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005.
- [5] H. Shah, J. Pinkerton, R. Recio, and P. Culley, "Direct data placement over reliable transports," <http://www.ietf.org/internet-drafts/draft-ietf-rddp-ddp-04.txt>, February 2005.
- [6] P. Culley, U. Elzur, R. Recio, S. Bailey, and J. Carrier, "Marker PDU Aligned Framing for TCP Specification," <http://www.ietf.org/internet-drafts/draft-ietf-rddp-mpa-08.txt>, October 2006.
- [7] S. Pope and D. Riddoch, "End of the Road for TCP Offload," Solarflare," Technical Report, April 2007.
- [8] C. Bestler, "Multi-stream MPA," *Cluster Computing, 2005. IEEE International*, Sept. 2005.
- [9] J. R. Iyengar, K. C. Shah, P. D. Amer, and R. Stewart, "Concurrent multipath transfer using SCTP multihoming," in *SPECTS 2004, San Jose*, July 2004.
- [10] B. Penoff, M. Tsai, J. Iyengar, and A. Wagner, "Using CMT in SCTP-based MPI to exploit multiple interfaces in cluster nodes," in *Proceedings, 14th European PVM/MPI Users' Group Meeting*, Paris, France, September 2007.
- [11] P. Balaji, W. Feng, S. Bhagvat, D. K. Panda, R. Thakur, and W. Gropp, "Analyzing the Impact of Supporting Out-of-Order Communication on In-order Performance with iWARP," in *In the IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, November 2007.
- [12] D. Dalessandro, A. Devulapalli, and P. Wyckoff, "Design and Implementation of the iWarp Protocol in Software," in *Parallel and Distributed Computing and Systems (PDCS), 2005*, November 2005.
- [13] P. Wyckoff, D. Dalessandro, and A. Devulapalli, "Software implementation and testing of iWarp protocol," http://www.osc.edu/research/network_file/projects/iwarp/, March 2007.
- [14] "RDMA Consortium. Architectural specifications for RDMA over TCP/IP," <http://www.rdmaconsortium.org>.
- [15] S. Bailey and T. Talpey, "The Architecture of Direct Data Placement (DDP) and Remote Direct Memory Access (RDMA) on Internet Protocols," <ftp://ftp.rfc-editor.org/in-notes/rfc4296.txt>, December 2005.
- [16] R. Recio, P. Culley, D. Garcia, J. Hilland, and B. Metzler, "An RDMA protocol specification," <http://www.ietf.org/internet-drafts/draft-ietf-rddp-mpa-08.txt>, April 2005.
- [17] W. D. Gropp and R. Thakur, "An Evaluation of Implementation Options for MPI One-Sided Communication," in *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, 2005, pp. 415–424.
- [18] "OpenFabric Alliance," <http://www.openfabrics.org/>.
- [19] C. Bestler and R. Stewart, "Stream Control Transmission Protocol (SCTP) Direct Data Placement (DDP) Adaptation," IETF RFC 5043, October 2007.
- [20] D. Dalessandro, A. Devulapalli, and P. Wyckoff, "iWarp Protocol Kernel Space Software Implementation," in *IPDPS 2006*, April 2006.