IMPLEMENTATION AND COMPARISON OF ISCSI OVER RDMA

BY

ETHAN BURNS

B.S., University of New Hampshire (2006)

THESIS

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Master of Science

in

Computer Science

May 2008

This thesis has been examined and approved.

Thesis director, Robert Russell,
Associate Professor of Computer Science

Radim Bartoš,
Associate Professor of Computer Science

Robert Noseworthy,
UNH-IOL, Chief Engineer

Barry Reinhold,
President, Lamprey Networks

Date

ACKNOWLEDGMENTS

I would like to thank Dr. Robert D. Russell for all of his help throughout this entire project. I would also like to thank my committee for their guidance and advice, and Mikkel Hagen for his critiques and suggestions. Finally, I would like to thank the iSCSI, OFA and iWARP groups at the University of New Hampshire InterOperability Lab for their vast knowledge and technical help.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

IMPLEMENTATION AND COMPARISON OF ISCSI OVER RDMA

by

ETHAN BURNS
University of New Hampshire, May, 2008

iSCSI is an emerging storage network technology that allows for block-level access to disk drives over a computer network. Since iSCSI runs over the very ubiquitous TCP/IP protocol it has many advantages over its more proprietary alternatives. Due to the recent movement toward 10 gigabit Ethernet, storage vendors are interested to see how this large increase in network bandwidth could benefit the iSCSI protocol.

In order to make full use of the bandwidth provided by a 10 gigabit Ethernet link, specialized Remote Direct Memory Access hardware is being developed to offload processing and reduce the data-copy-overhead found in a standard TCP/IP network stack. This thesis focuses on the development of an iSCSI implementation that is capable of supporting this new hardware and the evaluation of its performance.

This thesis depicts the approach used to implement the iSCSI Extensions for Remote Direct Memory Access (iSER) with the UNH iSCSI reference implementation. This approach involves a three step process: moving UNH-iSCSI from the Linux kernel to the Linux user-space, adding support for the iSER extensions to our user-space iSCSI and finally moving everything back into the Linux kernel. In addition to a description of the implementation, results are given that demonstrate the performance of the completed iSER-assisted iSCSI implementation.

CHAPTER 1

INTRODUCTION

1.1 The iSCSI Protocol

The *Internet Small Computer Systems Interface* (iSCSI) [12] protocol is an emerging *Storage Area Network* (SAN) technology that uses the *Internet Protocol* (IP, specifically TCP/IP) as its underlying fabric. Since iSCSI uses the ubiquitous TCP/IP protocol instead of a specialized network fabric it has a lower total cost of ownership than other SAN technologies [7]. In addition to being less expensive than its competitors, iSCSI also has the ability to provide networked storage to home users over Local Area Networks (LANs) or across the wide area Internet [16] without requiring any specialized hardware.

The iSCSI protocol has yet to be adopted for wide-spread deployment. Part of the reason for this is because of the huge investment that most companies already have in *Fibre Channel* (FC), the current leading SAN technology. Another reason that iSCSI has yet to see wide-spread deployment is that the IP networks that it uses don't currently match the speed of other specialized fabrics. IP networks typically run at either 100 megabits per second (Mbits/s) or one gigabit per second (Gbit/sec) speeds, where as FC fabrics can run at about 4-8 Gbits/s, and Infiniband [2] (another competing technology) can reach speeds in the 40 Gbits/s range.

1.2 Remote Direct Memory Access

The current 10 Gbit/s Ethernet (10GigE) standard allows IP networks to reach competitive speeds, however, standard TCP/IP is not sufficient to make use of the entire 10GigE bandwidth. The reason that TCP/IP has a difficult time filling a 10GigE link on its own is because of data copying, packet processing and interrupt handling done on the CPU. In a traditional TCP/IP network stack an interrupt occurs for every packet being received, data is copied at least once in the host computer's memory and the CPU is responsible for processing packet headers for all incoming and outgoing packets. In order to get rid of these inefficiencies, specialized *Remote Direct Memory Access* (RDMA) hardware is required. RDMA hardware uses the *iWARP* protocol suite [4] [13] [11] to move data directly from the memory of one computer to the memory of a remote computer without extra copies at either end. RDMA hardware will reduce the number of data copies that are required by the TCP/IP network to one-per-host (between the memory and the wire) and it will offload a bulk of the network-processing from the CPU. This means that applications with RDMA support will have lower CPU usage and the ability to utilize a full 10GigE link.

Figures 1-1 and 1-2 give a depiction of the difference between traditional TCP and iWARP on the receiver end. Notice that, in Figure 1-1, the traditional TCP example, the CPU is interrupted for each TCP packet that it receives, it then needs to process the TCP packet and put the data at the correct position in the reassembly buffer. After the data is assembled the CPU copies it into the application's receive buffer. With RDMA (Figure 1-2), the CPU is not involved; the RNIC assembles the TCP data straight into the application's final receive buffer, no extra copying is required. This frees up the CPU to perform other tasks while removing the overhead involved in copying, interrupt handling and processing of TCP packets.

Figures 1-3 and 1-4 give a depiction of the difference between traditional TCP and iWARP on the sender side. In Figure 1-3 the application copies data into the operating system's outgoing TCP buffer. The data must remain available in the operating system

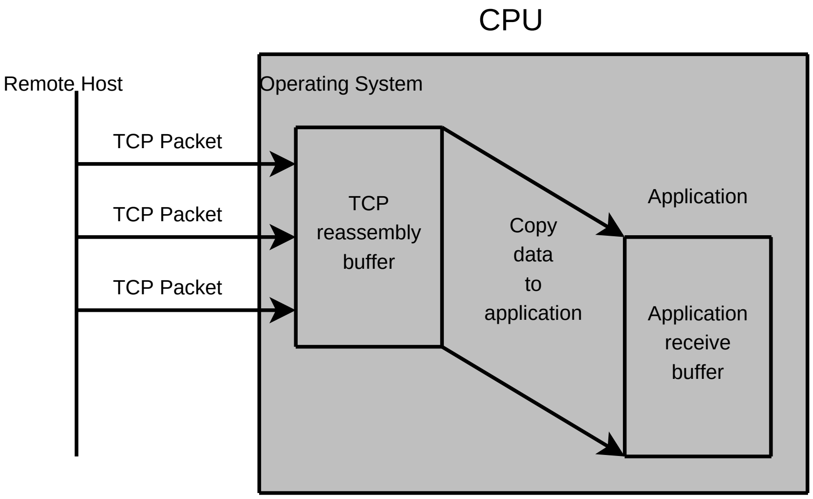

Figure 1-1: Traditional TCP Receiving Data

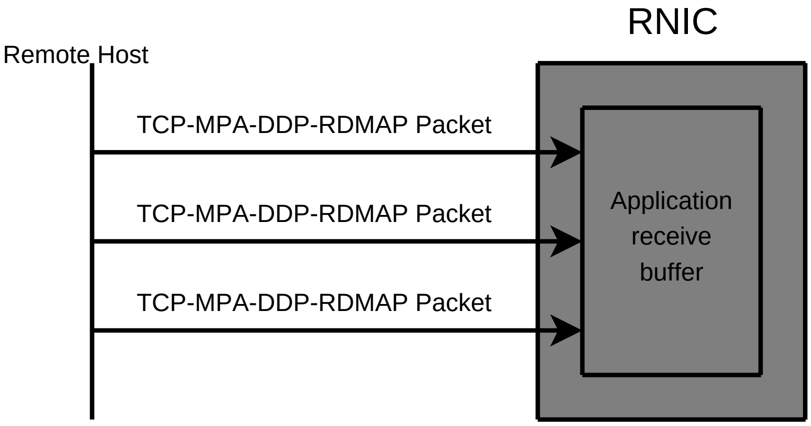

Figure 1-2: RDMA Receiving Data

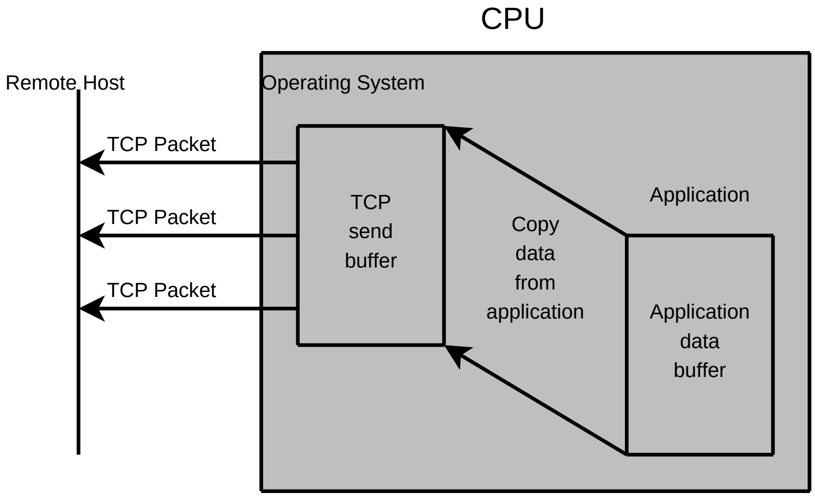

Figure 1-3: Traditional TCP Sending Data

because data loss may require TCP retransmissions. Since retransmissions are hidden from the application, the operating system must retain its own copy of the data in case the application destroys the data buffer before all of the data is acknowledged by the receiver. The semantics of RDMA prevent the application from destroying the buffer before all of the data has been transmitted. In Figure 1-4 the RNIC is able to send data directly from the application's buffer without the need of an extra copy.

1.3 iSCSI Extensions for RDMA

One of the difficulties with RDMA is that its usage differs from traditional TCP/IP and applications may need some re-designing to support it. In the case of iSCSI, a new IETF standard has been created that defines the *iSCSI Extensions for RDMA* (iSER) [8]. iSER describes a set of operational primitives that must be provided by an iSER implementation for use by an iSCSI implementation, and a description of how a conformant iSER-assisted iSCSI session must operate. Some of the design goals of iSER were to require minimal changes to the SCSI architecture and minimal changes to the iSCSI infrastructure while retaining minimal state information [8]. With the iSER extension an iSCSI implementation

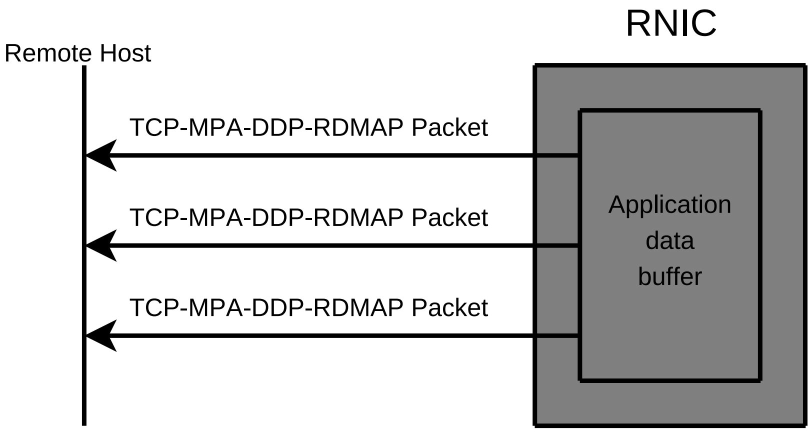

Figure 1-4: RDMA Sending Data

can make use of general-purpose RDMA hardware for very-high-speed data transfers. Figure 1-5 shows the differences in the network stack between iSER-assisted iSCSI and traditional iSCSI. Notice that in iSER-assisted iSCSI there are more layers in the stack, however, all layers under the iSER layer are handled by the specialized RDMA hardware, not the CPU. Also note that there is a possibility for future implementations to include the iSER layer directly on the RNIC too, leaving only the iSCSI layer for the CPU to handle.

1.4 Thesis Goals

The goal of this thesis was to build a working iSER-assisted iSCSI implementation for Linux using as much preexisting software as possible and to evaluate its performance in as many situations as time allows.

1.5 Outline

- Chapter 2 of this paper gives the background information on the technologies involved in this project. This chapter, briefly, describes the SCSI and iSCSI protocols, gives a description of Remote Direct Memory Access and tells how they all tie together with

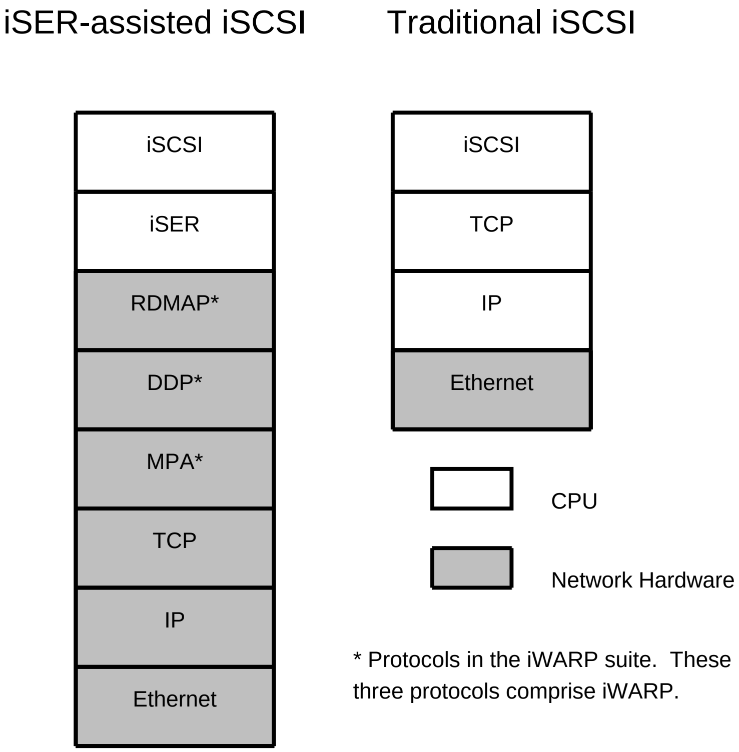

Figure 1-5: iSCSI-iSER-RDMA stack

the iSCSI Extensions for RDMA.

- Chapter 3 of this paper describes the approach that we used to accomplish our goal for implementing an iSER-assisted iSCSI. This chapter also describes some of the difficulties that we encountered while implementing this project.

- Chapter 4 of this paper describes the benchmarks and comparisons that we were able to run between different variations of iSER-assisted iSCSI and traditional iSCSI and the results that we got from them.

- Chapter 5 of this paper presents the conclusions that we have drawn from the results of this thesis. This chapter also describes work that future projects may perform based on what has been done in this thesis.

CHAPTER 2

BACKGROUND

There is a large number of protocols that are mentioned through this document. This chapter gives a brief background of the purpose of each of these protocols. Figure 2-1 gives an overview of where each protocol lies in the protocol stack when using iSER-assisted iSCSI.

2.1 SCSI

The Small Computer Systems Interface (SCSI) (the top of the protocol stack in Figure 2-1) is an architecture for connecting peripheral devices to computers. The SCSI architecture includes, not only a command protocol, but also a physical transport specification. The most popular transport for the SCSI command protocol is one called parallel SCSI. This transport system uses a set of parallel wires in order to transport entire words between two connected devices. Traditionally, high-end systems use the parallel SCSI transport to communicate with storage systems, such as disks or tape drives. One of the big advantages of using the SCSI architecture is that it is well supported by all major operating systems and by a very large number of devices used in high-end computing systems. For this reason the SCSI protocol has been very largely adopted in storage systems.

The SCSI protocol has two different classifications of devices: *initiators* and *targets*. A SCSI initiator device is one that constructs commands using Command Descriptor Blocks, or just CDBs for short, and a SCSI target device services SCSI commands received from an initiator. Typically an initiator will be the Host Bus Adaptor (HBA) in a computer (along

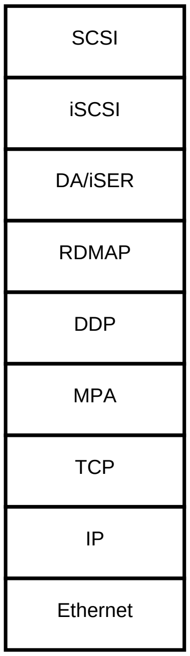

Figure 2-1: The iSER-Protocol Stack (with iWARP)

with the software that drives it) and the target will be a peripheral device (such as a disk drive, tape drive controller or even a printer). While the SCSI protocol contains a large number of different commands, the most typical commands are for querying the status of a device (checking if it is ready, obtaining its storage capacity, etc.), reading data from a device or writing data to a device. Commands that read data from a target device are called SCSI READ commands, and commands that write data to a target device are called SCSI WRITE commands[1].

While parallel SCSI is a very popular method for connecting peripherals to a computer, it has been known to suffer from some scalability issues. Parallel SCSI has a few limitations that create problems in modern storage environments, namely an upper limit on the number of connected devices and a length limitation on cabling. The parallel SCSI protocol has

[1] There are also bidirectional commands that do both a read and write as part of a single command, but those are not discussed in this document.

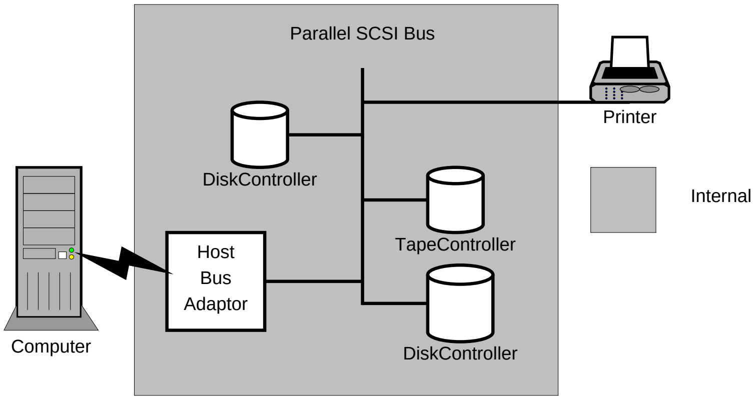

Figure 2-2: Devices Connected Using Parallel SCSI

many different varieties, however, the upper limit on the number of connected devices is never more than sixteen and the maximum length of cables is around 25 meters. Figure 2-2 shows a simple SCSI configuration with a few peripheral devices. Notice that the devices are connected to the parallel SCSI bus which is internal to the computer (with the exception of the printer). This configuration severely limits the number of devices that can be connected to a single computer.

In modern computing centers where petabytes of storage are required, these upper limits, imposed by parallel SCSI, are insufficient. With a maximum of sixteen peripherals per server, entire clusters of storage servers are often required to provide the desired storage capacity. With a limited distance of 25 meters, these storage servers must be in relatively close proximity, and alternative transport methods are required for backing up data to remote locations.

2.2 iSCSI

At the time that the SCSI protocol was created network bandwidth was considered a scarce resource and specialized I/O channels (such as parallel SCSI) were used for connecting computers to devices. Today, however, network bandwidth is plentiful and the need for large amounts of data storage has grown. In order to respond to these changes a new set of technologies called *storage area networks* (SANs) have been created. In a storage area network, storage devices are connected to storage servers over external serial network hardware, instead of internal parallel I/O channels. The advantage of using networking technology instead of I/O channels is that networks allow for a much greater scalability. With a storage area network there is virtually no limit on the number of devices that can be connected to a storage server. Additionally there is much greater distance supported by networking technology, and networks can span a very large area.

The Internet Small Computer System Interface (iSCSI) [12] protocol (which resides below SCSI on the protocol stack shown in Figure 2-1) was created as a cheaper alternative to the more expensive Fibre Channel technology that has, traditionally, been used to implement SANs. The iSCSI protocol uses a standard TCP/IP network as the physical transport for the very well known SCSI command protocol. Since TCP/IP networks can have a limitless number of connected devices and can span a, virtually, unbounded distance this enables SCSI to be used in a much more scalable environment. Figure 2-3 shows an iSCSI configuration. Notice that in this figure that the devices are all connected externally to the computer over a practically limitless network cloud. Another advantage of the iSCSI protocol is that IP networks are, currently, the most popular networking technology. This means that the hardware, equipment, tools and people with experience and knowledge about IP networks are readily available and are less costly than more specialized network fabrics.

The iSCSI protocol consists of iSCSI *sessions* between a target and an initiator device. Each iSCSI session can consist of multiple connections (mostly for redundancy and error recovery purposes), and each connection has three distinct phases: *login phase*, *full feature phase* and *logout phase*. The login phase of an iSCSI connection is used for the initiator and target devices to authenticate and negotiate parameters that will be used for the remainder of the connection[2]. After the login phase is complete the devices transition into a full

[2] It is also possible for some of these parameters to be re-negotiated after the login phase has completed.

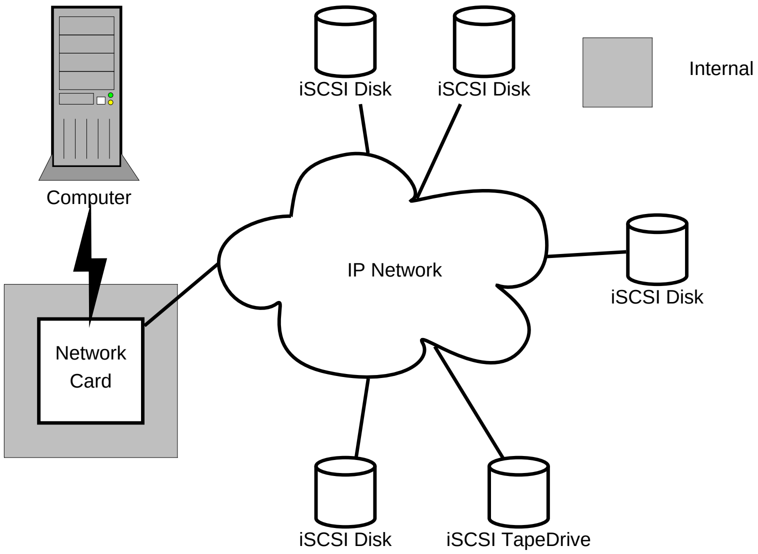

Figure 2-3: Devices Connected Using iSCSI

feature phase where SCSI commands and data are sent back-and-forth between the devices using iSCSI Command Protocol Data Units (PDUs) that contain embedded SCSI CDBs. Generally a connection will remain in the full feature phase for an extended period of time (the uptime of the computer, for example). When the devices are finished with their connection (typically when the initiator *unmounts* a disk or is shutdown) a simple logout exchange is performed.

2.2.1 Login Phase

The iSCSI login phase is used in order for the target and initiator to form a network connection, to communicate iSCSI specific parameters and, optionally, to authenticate each other. In this phase of a connection there are no SCSI CDBs transferred, instead special iSCSI PDUs are used. The login phase is for iSCSI only and would not take place in a SCSI session using a parallel SCSI bus as the physical transport.

During the login phase of an iSCSI connection, the initiator and target negotiate parameters that will be used for a connection by exchanging iSCSI LoginRequest PDUs and iSCSI LoginResponse PDUs. iSCSI LoginRequest PDUs are transmitted by the initiator device and each LoginRequest received by the target is responded to with an iSCSI LoginResponse PDU. These special PDUs contain text strings that associate values with iSCSI parameters called *keys*. Both the initiator and target use these text strings to negotiate or declare certain values that will be used for the new connection. The iSCSI protocol defines a large set of negotiable parameter that can effect the performance, memory requirements and error handling a in given connection and/or session[14].

There are two types of iSCSI keys: *declarative* and *negotiable*. Declarative keys declare an un-negotiable value for an iSCSI parameter, for example the TargetAlias key is a declarative key in which the target declares a human-readable name or description that can be used to refer to it. An example of a negotiated key is the HeaderDigest key where the target and initiator offer a list of header digest algorithms (including “None” which, if selected, means that no header digest is used for this connection) that they support, and the most preferable algorithm that they have in common is selected. iSCSI keys can also have a variety of different types of values. Some keys, such as the HeaderDigest key, take a list of values in order with the most preferable first while others may take either a simple numeric value or a boolean value (“Yes” or “No”).

Keys in which the originator offers a list of values must always contain the value “None” in the list in order to indicate the lack of a certain functionality. When responding to a key where the value is a list, the responder must choose the first value in the list that it supports (since the “None” value is guaranteed to be in the list, it is always possible to respond to a list of values key, even if the functionality is not supported). To keys in which the originator offers a numeric range the reponder must reply with a single value that is within the given range. In the case of a numeric range, the key can either be a *minimum function* or *maximum function* where the responder chooses either the minimum or maximum of the desired values respectively. Keys that contain boolean values are also negotiated based on one of two possible functions: *AND* and *OR*. With the AND function, if the originator offers the value “No” then the responder *must* reply with the value “No”, otherwise, the responder can reply with either value (“Yes” or “No”) depending on its desires or configuration. With the OR function, if the originator offers the value “Yes”, the responder must reply with the value “Yes”, otherwise, the responder can reply with either the value (“Yes” or “No”) depending, again, on its desires or configuration. These value types and negotiation functions give the administrator of an iSCSI environment a great deal of flexibility when deploying a storage network.

Once the initiator is finished negotiating all of its parameters, it sets the transition bit (or T-bit) of its next LoginRequest PDU to 1, and the next stage (NSG) field to the desired next stage in order to request a stage transition. The target is then able to continue negotiating parameters if it desires, or it can also set the T-bit and NSG field in its next LoginResponse PDU to transition to a new phase. Figure 2-4 shows a very simple iSCSI login phase where the devices do a single exchange of a set of key=value pairs and then transition to full feature phase (FFP). In Figure 2-4 the first LoginRequest/LoginResponse exchange is used to negotiate a set of required and optional iSCSI keys for the connection. Both of these PDUs have the T-bit set to zero, which indicates that there is no state transition request. During an iSCSI Login phase there may be many more of these LoginRequest/LoginResponse exchanges, in order to allow the devices to negotiate more keys, or to perform authentication during a special security negotiation phase (which may be transitioned to using the T-bit and NSG field). In the second LoginRequest in Figure 2-4, the initiator sets the T-bit to one and the next stage field to full feature phase to tell the target that it is ready to make a stage transition. The target replies with a LoginResponse with the T-bit also set to one and the next stage field set to full feature phase, which accepts the transition and the devices proceed into the full feature phase of the iSCSI connection.

2.2.2 Full Feature Phase

The iSCSI full feature phase is where the devices transfer encapsulated SCSI commands and data. During full feature phase the initiator creates and transmits iSCSI Command PDUs which contain SCSI CDBs that were generated by the SCSI initiator layer and passed down

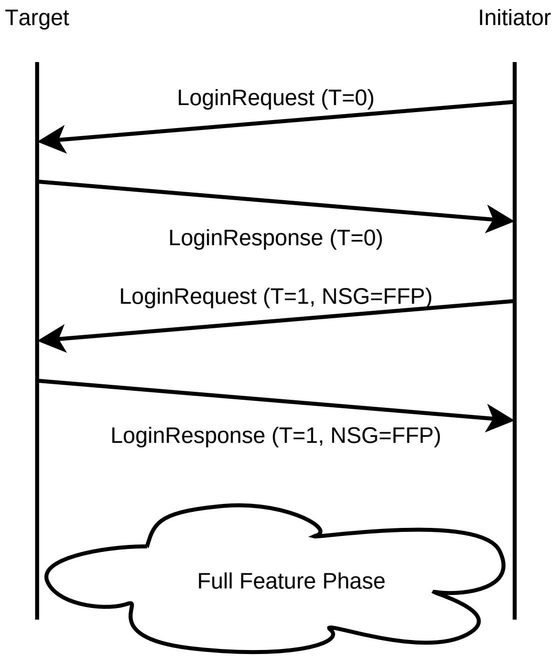

Figure 2-4: A Simple iSCSI Login

to iSCSI for transport. For each iSCSI Command PDU which is received by the target, the target will perform a service for the initiator (which may involve a data transfer in either direction) and will conclude the service with an iSCSI Response PDU that contains the SCSI response and status (typically either 0x00 for success or 0x02 for an error).

Certain SCSI Commands may make use of an optional data transfer phase. For example, in a SCSI READ or SCSI WRITE command data will be transferred to or from the initiator respectively. Since iSCSI target devices are expected to be peripherals with limited resources (such as an iSCSI enabled disk controller), all data transfers are controlled by the target. For READ commands, the target device sends iSCSI Data-In[3] PDUs to the initiator device. Since the target is sending the data in this situation there is no special handling required,

[3] The direction of SCSI and iSCSI commands are relative to the initiator device. Therefore, the Data-In PDU is *sent* by the target and *received* by the initiator. Likewise, Data-Out PDUs are *sent* by the initiator and *received* by the target.

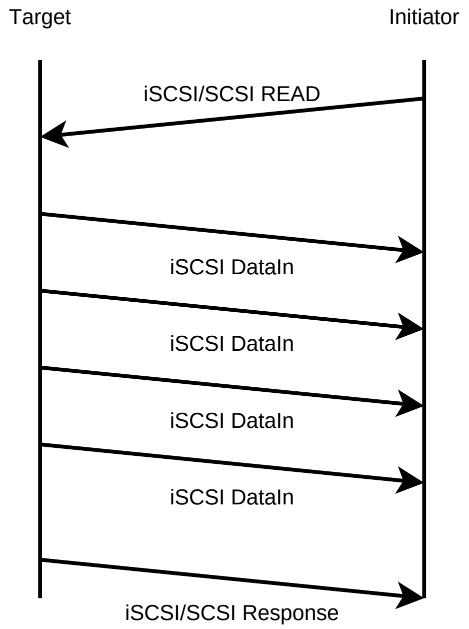

Figure 2-5: An iSCSI Encapsulated SCSI READ Command

the target has control of the transfer by default since it can select how much and how fast to send its PDUs. Figure 2-5 shows an iSCSI READ. In this figure, the initiator begins the read by sending an iSCSI encapsulated SCSI READ command to the target, the target then sends the read data to the initiator using Data-In PDUs and finishes the command with an iSCSI encapsulated SCSI response. This response tells the initiator that the read is complete. For WRITE commands a special Ready2Transfer (R2T) PDU is used for the target to signal to the initiator that it is ready to receive a set of Data-Out PDUs containing the data being written. R2T PDUs allow the target to maintain control over when the data is transferred and how much data is transferred at a time. Figure 2-6 shows an iSCSI WRITE with two R2T PDUs.

In order to allow the devices to have better control over the flow of data, all iSCSI data transfers happen in *bursts* of a certain size called MaxBurstLength (which is negotiated during the connection login phase). For READ commands, where the target sends a set of Data-In PDUs to the initiator, the bursts are not noticeable since one burst is immediately

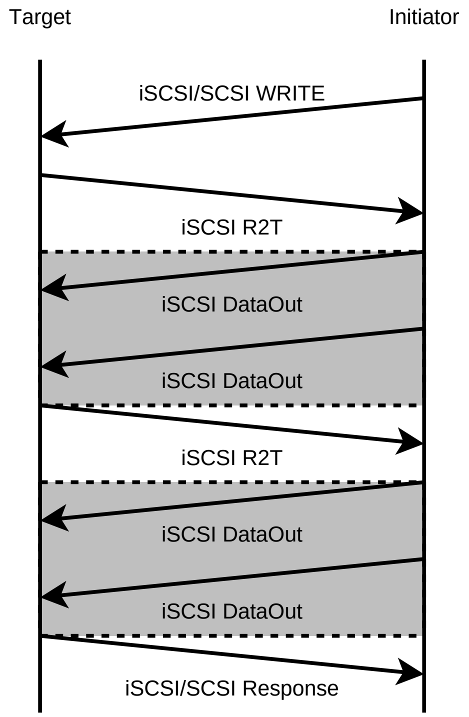

Figure 2-6: An iSCSI Encapsulated SCSI WRITE Command

followed by the next. For WRITE commands, however, the MaxBurstLength defines the maximum size of data that can be requested by a single R2T PDU. In Figure 2-6 the highlighted PDUs makeup a single data burst. In this figure there are two bursts of Data-Out PDUs, which are dictated by the R2Ts transmitted by the target. Additionally, the MaxRecvDataSegmentLength parameter (again negotiated during the login phase) dictates the maximum size of a single Data-In or Data-Out PDU. This means that any given burst of data may contain $\frac{MaxBurstLength}{MaxRecvDataSegmentLength}$ Data-In or Data-Out PDUs. Previous work has been done to evaluate these values and other negotiable values that iSCSI offers to see what their effect is on performance [14].

2.2.3 Logout Phase

When the initiator has finished with a connection, it sends an iSCSI LogoutRequest PDU to the target. Upon receiving a LogoutRequest the target device will respond with an iSCSI

LogoutResponse PDU and the devices will tear down the connection.

2.3 RDMA

The speeds of networking technologies in today's high-performance computing environments are rapidly growing. In high-end systems, the CPU has replaced the network as the new "bottle neck". With technologies like 10 Gigabit/sec Ethernet (10GigE) modern CPUs lose the ability to keep enough data flowing into the network to make full use of the available bandwidth. With a reliable transport system like TCP, the operating system must make and retain copies of all outgoing data so that it is able to retransmit it if the data does not make it, without error, through the network to its destination. On the receiving side, the operating system must maintain a reassembly buffer where, possibly out of order, data is stored before being copied into the final receive buffer. In addition to all of the overhead involved in copying data, the operating system is constantly interrupted each time a new segment of data arrives on the network. At each network interrupt, the operating system must copy the data off of the network card and process the header information to determine how to interpret the data. With all of this interrupting, copying and processing a typical CPU is only able to make use of about half of a 10GigE connection. Not only is the CPU insufficient to use all of the available bandwidth, but it also must spend a lot of its time maintaining the network traffic instead of handling other processing that the system may need done.

To fix the problems stated above, specialized Remote Direct Memory Access (RDMA) hardware has been developed to alleviate the need for data copying, and to offload network processing from the CPU. In an RDMA protocol auxiliary information travels across the network with the data so that it can be immediately placed in its final receive buffer. Along with this auxiliary information, specific segment sizing and special markers can be used to allow detection of packet boundaries to eliminate the need for a lot of reassembly in the case of out of order data. These two features allow RDMA hardware to perform "zero-copy" transfers directly from the memory of one computer to the memory of another.

Although there are other RDMA technologies (such as Infiniband), the one that we are mostly concerned with for this project is iWARP [4] [13] [11]. iWARP is a protocol suite that allows for RDMA across a TCP/IP network (such as those used with iSCSI). The iWARP protocol suite is divided into three different protocols:

1. Marker PDU Aligned Framing (MPA) [4]

2. Direct Data Placement (DDP) [13]

3. Remote Direct Memory Access Protocol (RDMAP) [11]

2.3.1 Marker PDU Aligned Framing

Since TCP is a stream oriented protocol, *upper layer protocols* (ULP)[4], that require the detection of record boundaries, must use a reassembly buffer to store received TCP information while looking for message boundaries. If the header information of an upper layer frame arrives late (due to lost or out of order TCP segments), it will be necessary for the ULP to delay processing of all subsequent TCP segments until the one containing the header arrives. This is because the ULP will not be able to determine where the next frame begins since that information is contained in a header that has not yet arrived.

The MPA protocol resides directly above the TCP layer in the protocol stack (refer to Figure 2-1) . MPA defines a method of reliably tracking frame boundaries within a TCP stream by using fixed sized frames and by optionally embedding markers at known offsets in a TCP stream. Using MPA on top of TCP provides the upper layer protocol with a method of sending fixed sized records which, *in the optimal case*, will arrive intact embedded within a single TCP segment. With Marker PDU Aligned Framing a receiver is not required to

[4] The term upper layer protocol refers to any protocol that resides above the current protocol on the stack shown in Figure 2-1. In the case of MPA, the upper layer protocol refers to DDP. In the case of DDP, the upper layer protocol refers to RDMAP. Finally, in the case of RDMAP, the upper layer protocol refers to any protocol that is making use of the RDMA stack in order to gain high-performance network usage.

reassemble data from the TCP stream, which allows an ULP (such as DDP) to have the ability to skip using intermediate storage and to directly place received data into memory.

In addition to providing fixed sized framing, the MPA protocol also provides padding for 4-byte alignment and an optional *cyclic redundancy check* (CRC). The CRC can be used in order to determine if the received packet has been modified during transmission. The purpose of the CRC in the MPA layer is to provide extra checking for errors that may have been missed by the TCP layer. If the CRC option is not enabled, the MPA header still contains the CRC field, it is just left unused.

In practice, the MPA markers and CRCs are not used. In, what the MPA specification refers to as an optimal situation, the MPA layer works very closely with the TCP layer in order to guarantee that each MPA frame will be contained in a single TCP segment. In this optimal case, MPA markers provide little benefit, since reassembly is never needed to reconstruct an MPA frame. In current hardware, this optimal case is implemented and markers would only add extra overhead. Additionally, the optional MPA CRCs are also unused in current hardware.

2.3.2 Direct Data Placement

Above the MPA layer in the iWARP stack (see Figure 2-1), is the Direct Data Placement protocol (DDP). DDP allows an upper layer protocol (such as RDMAP) to send data directly to a destination receive buffer without the need for an intermediate copy. DDP works with MPA to segment data into small frames where each frame will fit directly into a single TCP segment. Using DDP with MPA allows for frames to be directly placed in the receiver's memory without the need for buffering or reassembly even when TCP segments are lost or arrive out of order. Without the use of MPA framing, lost or out of order TCP segments would require DDP to use reassembly storage in order to reconstruct the frames and find the header with the memory placement information. Figure 2-7 shows a theoretical DDP frame that does not use MPA. Each block in this figure represents a TCP segment. Notice that if the TCP segment labeled "A" does not arrive first, the other segments must

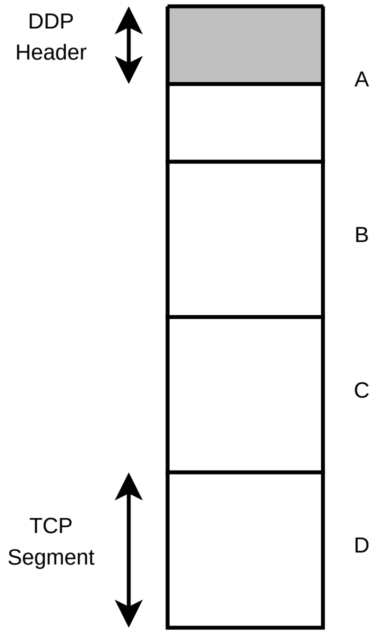

Figure 2-7: A Theoretical DDP Frame Without MPA

be buffered until segment "A" arrives with the DDP header. Figure 2-8 shows what a DDP frame using MPA would look like[5]. Notice that the entire DDP frame fits into just a single TCP segment, so even if one is lost there is always a DDP header available in the next. This is the ideal behavior of TCP, MPA and DDP, depending on the implementation, MPA frames may not fit exactly into a single TCP segment. In these situations, the MPA stream markers can be used to help detect frame boundaries in the case of packet loss.

DDP provides two different mechanisms to its upper layer for transferring data: *Tagged Buffer* transfers and *Untagged Buffer* transfers. In a tagged buffer transfer, the upper layer protocol must advertise the buffer that will be used in the transfer to its peer. The peer then has the ability to perform a direct, memory-to-memory transfer into the advertised buffer. For untagged buffer transfers, the receiver must explicitly post untagged receive buffers on a queue for the peer to write data to. These buffers are posted in the order that

[5] Note that the size of the single TCP segment shown in Figure 2-8 would be the same size as the smaller segments (labeled A-D) in Figure 2-7, however Figure 2-8 is enlarged to show the headers more clearly.

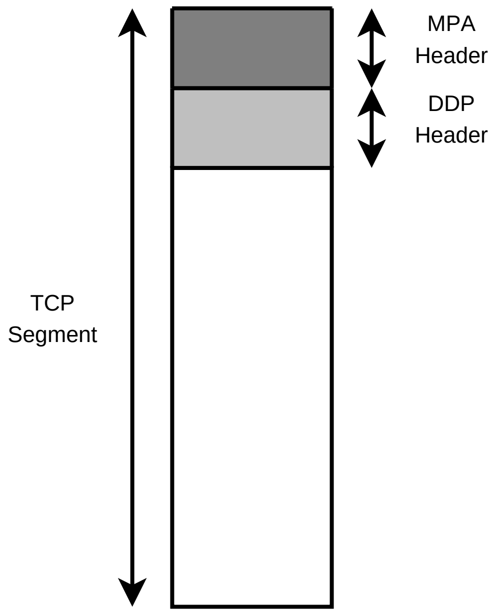

Figure 2-8: A DDP Segment Using MPA

the receiver would like them to be consumed by the RDMA hardware upon receiving an untagged transfer. Untagged transfers also require the upper layer to perform flow control, since a transfer will fail if there is not an appropriately sized receive buffer available at the front of the queue to accept an incoming message.

DDP uses the term *data sink* to refer to the the destination of a data transfer and the term *data source* to refer to the source of a data transfer. DDP also defines the notion of a *Steering Tag* (Stag) which is an opaque identifier for a tagged buffer. When performing a tagged buffer transfer, the data source uses the Stag from the data sink to transfer data directly into the data sink buffer. In order for the data source to know the Stag of a data sink buffer, an upper layer protocol at the data sink is required to advertise the Stag along with a Tagged Offset (TO) and a length for the desired transfer. In untagged buffer transfers the data source sends a message to the "next" untagged buffer on the front of the untagged buffer queue at the data sink. It is the job of the upper layer protocol on both the data sink and the data source to ensure that the data sink has enough queued buffers to be able to receive the untagged transfer.

2.3.3 Remote Direct Memory Access Protocol

The Remote Direct Memory Access Protocol (RDMAP) [11] uses DDP to provide remote direct memory access over TCP/IP networks and other reliable transports. RDMAP is also designed to allow for *kernel bypass*, in which a user-space process bypasses the operating system kernel when performing data transfers. Kernel bypass is possible because the entire network stack (RDMAP, DDP, MPA, TCP, IP and Ethernet in Figure 2-1) is offloaded onto the RDMA hardware. A user-space application, with access to the RDMA hardware is able to make network connections and send data without any interaction with the operating system kernel. Kernel bypass allows a user process to avoid the extra context switches and data copies that are usually involved when communicating through the kernel.

RDMAP defines a set of data transfer operations which are summarized below [11]:

Send uses an untagged DDP data transfer in order to send information from a data source to an untagged data sink buffer on the receiver side. When using a Send operation the upper layer protocol at the data source provides a message and a length to send to the data sink. At the data sink side, after receiving the data from a Send operation, the upper layer protocol is provided with the length and the received message.

Send with Invalidate (SendInv) uses an untagged DDP transfer, just like the Send operation. However, in a SendInv message an additional Stag in the header is transfered to the receiver. After the receiver gets the data from the SendInv message, it invalidates the use of the buffer that is identified by the extra Stag field (which is a separate buffer from the one that is receiving the data transmitted with this send operation).

Send with Solicited Event (SendSE) uses an untagged DDP transfer, just like the Send operation, but after receiving the data in the data sink buffer, it also generates an event in the receiver side.

Send with Solicited Event and Invalidate (SendInvSE) is a combination of SendInv and SendSE. After the receiver gets the data, it invalidates the buffer identified by the extra Stag and generates an event on the receiver side.

RDMA Read uses a tagged DDP transfer in order to perform a zero-copy read from a remote data source buffer. The RDMA Read operation uses a two message exchange in order to perform the data transfer. First, the data sink sends, to the data source, an RDMA ReadRequest message which contains the <Stag,TO,len> tuple that describes the data source buffer, and the <Stag,TO,len> tuple that describes the data sink buffer. After receiving the RDMA ReadRequest message, the data source will send a RDMA ReadResponse message to the data sink which performs the zero-copy transfer from the data source buffer to the data sink buffer. On the data sink side, the upper layer protocol provides the RDMAP layer with the data source buffer's Stag, Tagged Offset and length and the data sink buffer's Stag, Tagged Offset and length. The upper layer protocol on the data source is not notified of an RDMA Read operation performed by the remote peer.

RDMA Write uses a tagged DDP transfer to write data from the data source buffer directly into the data sink buffer on the remote peer. On the data source side, the upper layer protocol provides the RDMA layer with the data sink buffer's Stag, Tagged Offset and with a message and message length. The upper layer protocol on the data sink is not notified of a RDMA Write operation performed by the remote peer.

Terminate uses a DDP untagged transfer to send error information to the remote peer.

These data operations are provided to users to allow them to access the functionality of RDMA hardware. The exact means used to provide these seven operations is left open in order to facilitate compatibility with a large range of systems.

2.4 iSER

In order for iSCSI to compete with other SAN technologies, which have the ability to perform at speeds that are greater than 1 Gigabit/sec, it must be able to make efficient use of 10GigE. To this ends, the iSCSI Extensions for RDMA (iSER) [8] protocol was created. iSER is an implementation of the Datamover Architecture (DA) for iSCSI [3]. DA specifies operations that a small protocol layer, residing below iSCSI on the protocol stack (see Figure 2-1), needs in order provide a standardized interface for iSCSI to transfer commands and data over a network. The iSER extensions specify an implementation of DA that encapsulate and translate iSCSI PDUs in order to send them over an *RDMA Capable Protocol* (RCaP), such as iWARP, while retaining a minimal amount of extra state information.

2.4.1 iSER Operational Primitives

The Datamover Architecture for iSCSI defines a set of operational primitives that must be provided to the iSCSI layer. iSER describes an implementation of these operational primitives that allow for using an RCaP layer as the transfer method. There are nine operational primitives that an iSER layer will provide for iSCSI.

Send_Control operational primitive is used by an iSCSI initiator and target device to request the transfer of *iSCSI Control type PDUs* which are iSCSI PDUs that contain either iSCSI/SCSI commands or unsolicited Data-Out PDUs. Depending on the type of iSCSI Control PDU that is being sent, a different RDMA Send-type operation is used:

iSCSI Command PDUs sent by the initiator are transferred using RDMA Send with Solicited Event operations. If the Command PDU will require an optional data transfer phase, the iSER layer at the initiator will advertise an Stag to the target for the data sink or data source buffer (depending on the direction of the transfer) to be used.

iSCSI Response PDUs sent by the target are transferred using an RDMA Send with Solicited Event operation unless the Response is responding to an iSCSI Command PDU that involved a data transfer where an Stag was advertised; in this case a Send with Solicited Event and Invalidate operation is used to automatically invalidate the Stag used for the completed data transfer.

iSCSI Data-Out PDUs sent by the initiator for unsolicited data transfers are transferred using a Send operation with the exception of the final Data-Out PDU in a burst which is transmitted with the SendSE operation instead.

Put_Data is an operational primitive that is used only by the target. This operational primitive is used to send an iSCSI Data-In PDU. The iSER layer at the target translates the Data-In PDU, received from the local iSCSI layer, into an RDMA Write operation using the Stag that was advertised by the initiator in the iSCSI Command PDU that requested the data transfer. The data is written using a zero-copy transfer directly into the initiator's data sink buffer.

Get_Data is an operational primitive that is used only by the target. This operational primitive is used to send an iSCSI Ready2Transfer (R2T) PDU. The iSER layer at the target translates the R2T PDU, received from the local iSCSI layer, into an RDMA Read operation using the Stag advertised by the initiator in the iSCSI Command PDU that requested the data transfer. The RDMA Read from the target will perform a zero-copy read of the data from the initiator's data source buffer into the data sink buffer on the target.

Allocate_Connection_Resources is used by the initiator and the target iSCSI layer to tell the iSER layer to allocate connection specific resources that it will need for an RDMA session.

Deallocate_Connection_Resources is used by the initiator and the target iSCSI layers to tell the iSER layer to deallocate any resources that it has allocated for a given connection.

Enable_Datamover is used by the target and the initiator in order for the iSCSI layer to tell the iSER layer that it should enable iSER-assisted mode on the given connection.

Connection_Terminate is used by the initiator and the target iSCSI layers to tell the iSER layer to terminate the RDMA connection.

Notice_Key_Values is used by the initiator and the target to allow the iSER layer to see the result of iSCSI Login key=value text negotiations. The iSER layer can then use some of the information from these negotiations to allocate the appropriate amount of resources for this connection.

Deallocate_Task_Resources is used by the initiator and the target to allow the iSER layer to free up resources that it has allocated for a specific iSCSI task.

Along with the operational primitives provided by the iSER layer to iSCSI, there is another set of operational primitives that the iSCSI layer must provide to iSER. These primitives are used as call backs for iSER to signal the local iSCSI layer of various events.

Control_Notify is used by the iSER layer to signal to the iSCSI layer that a new iSCSI Control type PDU has been received. iSCSI Control type PDUs, as described above, are the iSCSI PDUs that contain iSCSI specific commands, SCSI commands or unsolicited Data-Out PDUs.

Data_Completion_Notify is used by the iSER layer on the target to notify the iSCSI layer that a data transfer (either an RDMA Read or an RDMA Write) has completed.

Data_Ack_Notify is used by the iSER layer at the target to notify the iSCSI layer of an arriving data acknowledgment.

Connection_Termination_Notify is used by the iSER layer to notify the iSCSI layer of an unexpected connection termination.

2.4.2 iSER Specific iSCSI Login/Text Keys

The iSER extensions also specify a few additional iSCSI Login/Text keys used to negotiate iSER specific parameters.

RDMAExtensions is a boolean type key that is added by iSER to the iSCSI Login phase. This optional key must be negotiated to "Yes" by both the initiator and target in order to operate in iSER-assisted mode.

TargetRecvDataSegmentLength and InitiatorRecvDataSegmentLength keys are numeric keys that are added by iSER to the iSCSI Login phase. The values of these keys declare the maximum number of bytes that either the target or initiator can receive in a single iSCSI Control type PDU. These two keys are part of the flow control mechanism that iSER uses for RDMA untagged transfers. The values of these keys corresponds to the size of the untagged buffers that are queued with the RCaP layer.

MaxOutstandingUnexpectedPDUs declares the maximum number of outstanding, unexpected PDUs that a device can receive. This key is also used in order to help with flow control of RDMA untagged transfers. This parameter corresponds to the number of untagged buffers that are queued with the RCaP layer.

2.4.3 iSER Inbound/Outbound RDMA Read Queue Depths

In addition to flow control for untagged buffers, the iSER layer is also responsible for flow controlling RDMA ReadRequests. When performing RDMA Read operations, an RCaP layer must have the appropriate amount of resources to handle all of the possible RDMA ReadRequest messages that the remote peer may send. In the case of iSER, only the target will be sending RDMA ReadRequests, therefore the initiator declares the maximum depth of the Inbound RDMA Read Queue (IRD). The iSER layer at the target is then able to select a maximum depth of the Outbound RDMA Read Queue (ORD) which is at most as large as the IRD. If the ORD value proposed by the target is less than the IRD value declared by the initiator, the initiator may choose to free up the extra resources. These parameters are negotiated in an additional Hello message exchange that is performed in iSER-assisted mode as described in Section 2.4.5.

2.4.4 iSER Connection Setup

An iSER-assisted iSCSI session is established by first performing a standard iSCSI login in regular TCP streaming mode. During the login phase, if both of the devices negotiate RDMAExtensions=Yes, then the following full feature phase will use the iSER extensions and RDMA. The RDMAExtensions key must be offered in the first available LoginRequest or LoginResponse PDU in order to allow the devices to negotiate the additional iSER parameters on a successful RDMAExtensions negotiation. A failure to negotiate the RDMAExtensions key to "Yes" will cause the devices to fallback on traditional, unassisted, iSCSI in the proceeding full feature phase.

On the iSCSI initiator, the `Allocate_Connection_Resources` operational primitive is invoked on the iSER layer just before sending a LoginRequest with the T-bit set for transition into full feature phase. The initiator, then has the option of invoking the `Notice_Key_Values` primitive, before invoking they `Allocate_Connection_Resources` primitive, to allow the iSER layer to see the results of the login phase negotiations. After receiving the target's LoginResponse message with the T-bit set for full feature phase, the initiator must invoke the `Enable_Datamover` primitive in order to transition the TCP connection from streaming mode into iSER-assisted mode.

On the iSCSI target, the `Allocate_Connection_Resources` operational primitive must be invoked on the iSER layer just before sending a LoginRepsonse with the T-bit set in, in response to the initiator's request to transition into full feature phase. As with the initiator side, the target can optionally invoke the `Notice_Key_Values` primitive to allow the iSER layer to see the results of the login phase negotiations. The target must invoke the `Enable_Datamover` operational primitive with its final LoginResponse PDU with the T-bit set to complete the transition to full feature phase and to transition the TCP connection into iSER-assisted mode.

2.4.5 iSER Hello Message Exchange

Directly after entering full feature phase of the iSCSI connection the initiator must send an iSER Hello message to the target declaring the iSER-IRD (the maximum number of outstanding Read Requests that the RCaP layer at the initiator is prepared to receive). In response to the Hello message, the target must send the initiator a HelloReply message declaring the iSER-ORD (the maximum number of outstanding Read Requests that the iSER target is prepared to send to the initiator) that will be used for the connection. Upon receiving the HelloReply message from the target, if the iSER-ORD value is lower than the iSER-IRD value, the initiator may free up any extraneous resources. Once the Hello message exchange is completed, the devices are successfully operating in an iSER-assisted iSCSI full feature phase where commands and data are transmitted as described in Sections 2.4.7 and 2.4.8.

2.4.6 iSER Connection Termination

In a normal logout, the iSCSI layer at the initiator uses the `Send_Control` operational primitive to send an iSCSI LogoutRequest PDU to the target. After receiving the iSCSI LogoutRequest, the target responds with an iSCSI LogoutResponse (using the `Send_Control` operational primitive), and then invokes the `Connection_Terminate` primitive to notify the iSER layer that it should close the RCaP stream. After the initiator receives the iSCSI LogoutRespsonse from the target, it also invokes the `Connection_Terminate` operational primitive on its end to notify its iSER layer that it should terminate the RCaP stream.

2.4.7 iSER-Assisted iSCSI Read Operation

Figure 2-9 shows an iSER-assisted iSCSI Read operation. In this figure, the initiator invokes the `Send_Control` operational primitive on its local iSER layer to request the transmission

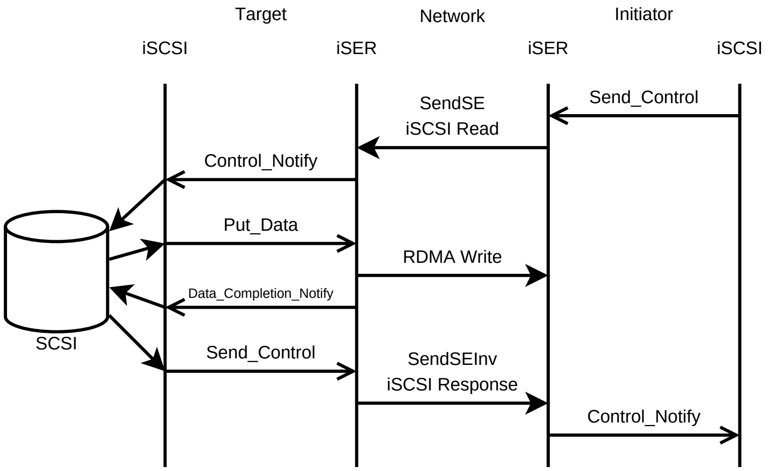

Figure 2-9: An iSER-Assisted iSCSI Read

of an iSCSI Command PDU encapsulating a SCSI Read CDB. When the iSER layer at the target receives the Command PDU, it generates a `Control_Notify` event at the target iSCSI layer to notify it of the received iSCSI Command. After passing the Read CDB off to the SCSI layer and acquiring the read data buffer, the iSCSI target invokes the `Put_Data` operational primitive on its local iSER layer to transfer the read data to the data sink buffer on the initiator using a zero-copy RDMA Write operation. This entire RDMA Write operation happens without intervention of the CPU on either end. Once the RDMA Write operation completes the iSER layer at the target notifies the iSCSI layer by invoking the `Data_Completion_Notify` operational primitive on the target-side iSCSI layer. When the target SCSI layer has been notified that the data transfer is complete, the target iSCSI layer invokes the `Send_Control` operational primitive in order to tell the target-side iSER layer to transmit an iSCSI Response PDU. After receiving the iSCSI Response PDU, the initiator's iSER layer notifies the iSCSI initiator by invoking the `Control_Notify` primitive and the transfer is completed. Notice that a Read at the SCSI/iSCSI layer is implemented by an RDMA Write operation.

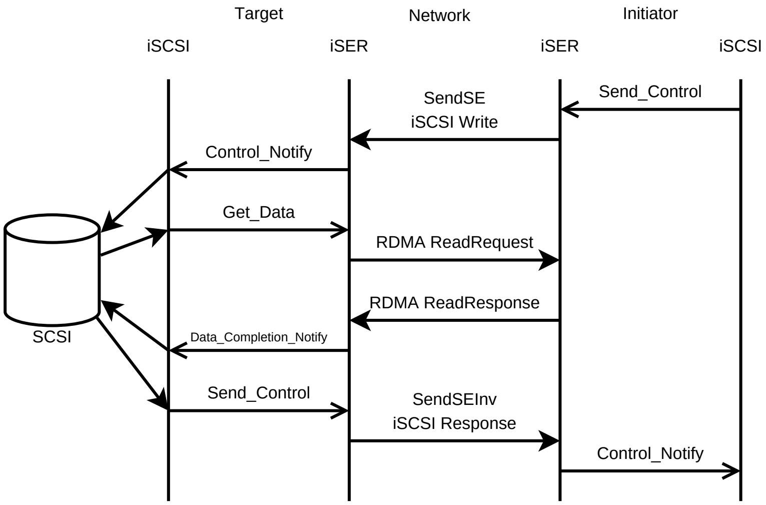

Figure 2-10: An iSER-Assisted iSCSI Write

2.4.8 iSER-Assisted iSCSI Write Operation

Figure 2-10 shows an iSER-assisted iSCSI Write operation. To begin the Write operation, the initiator invokes the `Send_Control` operational primitive on its local iSER layer to transmit an iSCSI Command PDU encapsulating a SCSI Write CDB. Upon receiving the iSCSI Command PDU, the target-side iSER layer invokes the `Control_Notify` operational primitive on the target iSCSI layer to notify it of the received command. The iSCSI layer on the target passes the CDB to the target SCSI level, and then uses the `Get_Data` operational primitive to tell the target-side iSER layer to read the data from the data source buffer on the initiator. The iSER layer at the target transmits an RDMA ReadRequest to the initiator RDMA layer which responds with an RDMA ReadResponse message containing the Read Data. The RDMA ReadRequest and ReadResponse messages are exchanged without the intervention of the CPU on either end and the data is transferred without using any extraneous copies. Once the target side RDMA layer is finished placing the Read data, the iSER layer notifies the iSCSI layer at the target using the `Data_Completion_Notify` operational primitive. The target iSCSI layer notifies the target SCSI layer that the data transfer has completed, and then uses the `Send_Control` primitive to have an iSCSI Response PDU sent to the initiator. When the initiator's iSER layer receives the iSCSI Response PDU it notifies the iSCSI layer using the `Control_Notify` primitive and the transfer is completed. Notice that a Write at the SCSI/iSCSI layer is implemented by an RDMA Read operation.

2.4.9 iSER and Current RDMA Hardware

With current iWARP hardware, the iSER specification is not possible to follow as stated. Specifically, current RDMA hardware does not perform "TCP stream transitioning", "Zero-based virtual addressing" or the Send with Solicited Event and Invalidate operation. These three issues are discussed in detail in Sections 3.6.1 3.6.2 and 3.6.3 respectively. The iSER implementation discussed in this thesis uses a set of *extended operational primitives* in order to aid in the establishment of RDMA connections. Additionally, a non-standardized iSER header is used in order to communicate extra information that is required by current RDMA hardware in order to perform RDMA data transfers. Finally, since the Send with Solicited Event and Invalidate operation is not supported, our implementation uses a Send with Solicited Event operation in order to transmit iSCSI/SCSI response PDUs.

CHAPTER 3

IMPLEMENTATION

In order to perform our comparisons between iSER-assisted and traditional iSCSI, we first needed to build a working iSCSI-iSER-iWARP stack. Due to the complexity of each layer in this stack, we choose to base our work on preexisting implementations instead of creating our own where possible. This process involved selecting from various software projects to use as a base, and the reworking them to fit our needs.

3.0.10 Protocol Layers

The software layers that we choose to use for this project are:

- The University of New Hampshire iSCSI Reference target and initiator (UNH-iSCSI).
- The OpenFabrics Alliance (OFA) [1] RDMA stack (for communication with the RDMA hardware). We also made use of a software-only implementation of the iWARP protocol developed at the Ohio Super-Computer Center (OSC) which provides an OFA-compatible interface [6].
- An old iSER implementation from the OpenIB iSER target project [15].

Figure 3-1 shows how all of the protocol layers fit together with iSCSI and the iSER extensions. The following sections describe how each of these layers work and interact with the pieces that sit above or below it in the protocol stack. Additionally, a description of the modifications that were required and any difficulties that were encountered is given.

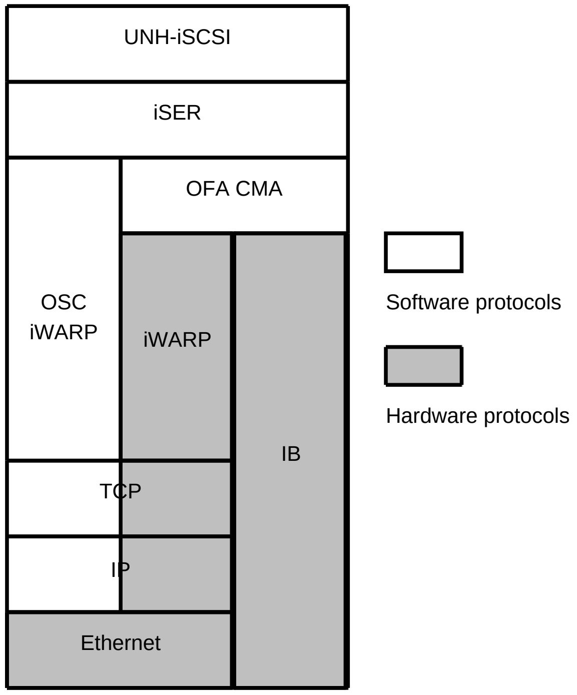

Figure 3-1: iSCSI-iSER-RDMA Stacks.

3.1 iSCSI

In order to create a working iSER solution, we first decided to choose an iSCSI implementation to which we would add iSER support. There are a handful of freely available iSCSI implementations for the Linux operating system. We selected one that was created here at the University of New Hampshire InterOperability Lab (UNH-iSCSI) [9]. There are a number of reasons for this choice:

- The UNH-iSCSI implementation was created and is maintained here at the InterOperability Lab (IOL) at the University of New Hampshire, which makes it very easy to get support for the project. Most other projects would require a lot of support from external organizations.

- UNH-iSCSI contains both an iSCSI initiator and target together in one project. Both of these are well tested and supported by the IOL. Most other projects have chosen to build either an iSCSI initiator or a target, not both. This could lead to interoperability issues between the two separate implementations.

- Previous work has already been done to add iSER support to UNH-iSCSI [10]. While there are other projects available that are developing or are thinking about developing iSER, the UNH-iSCSI project already had some preliminary hooks for iSER that were added during this previous, incomplete attempt.

3.1.1 Three Steps

The UNH-iSCSI project was developed as a Linux kernel module, which means that UNH-iSCSI code runs in kernel-space. Kernel-space code is much more difficult to debug than user-space code because the user-space is provided with memory protection. A mistake made in kernel-space can bring down an entire system, whereas a mistake made in user-space will, at most, abort the process that caused the error. In addition to memory protection, user-space code can be easily run in a debugger which can provide line-by-line stepping and can give stack-traces when errors are encountered.

Since the Open Fabrics Alliance RDMA stack, which we choose to use for communicating with the RDMA hardware, provides a user-space API in addition to a kernel-space API and since user-space programs are much easier to debug, and therefore to develop, than kernel modules we decided to use three steps in our implementation.

1. Move UNH-iSCSI into user-space.

2. Add support for iSER-assisted iSCSI, in user-space.

3. Move everything back into the Linux kernel.

3.1.2 Emulating the Kernel-Space APIs

The first step in our implementation was to move the UNH-iSCSI project into the Linux user-space. As described above projects that run in user-space are much easier to develop than projects in kernel-space. In user-space we are provided with an almost limitless set of tools such as the GNU Debugger `gdb`, the `valgrind` profiler and more. The process that we used to move our project out of the kernel and into the Linux user-space is described below.

In Linux, the kernel-space code is not able to make use of traditional user-space libraries. This is partially because the kernel implements most of the library functionality on behalf of user-space. Another reason that the library functions in the kernel are different is because the kernel does not see the same virtual view of the computer hardware that user-spaces processes do. The kernel has the ability to use some shortcuts that would be impossible in user-space (such as direct access to the hardware without an abstraction layer). The main task that we did in order to move UNH-iSCSI into user-space was to emulate a large set of functions traditionally provided by the Linux kernel using only user-space functionality.

In order to properly mimic the kernel-space libraries, we attempted to use as much code directly from the Linux kernel as possible. Specifically, we were able to use the Linux kernel doubly-linked list and atomic_t (a structure which defines a multiprocessor atomic counter) data types directly from the source code of the kernel. We were also able to make use of the Linux bit-wise operation functions and some other utility routines directly. Other higher level structures, such as threading and semaphores, we were not able to copy directly.

To build the API for pieces that we were not able to pull directly out of the kernel source code, we needed to implement them in terms of user-space functions. To handle threading, we used the POSIX threads library. This library provides an API for creating and manipulate new, concurrent threads of control. Since UNH-iSCSI is mostly event driven, it relies heavily on threads in order to wait for events from multiple sources (such as receiving on the network, transmitting on the network, the SCSI mid-level and more). The use of threads also requires locking primitives to prevent race conditions when multiple threads are attempting to access common data at the same time. The Linux kernel is heavily parallelized and has a very rich set of tools for maintaining concurrent threads and preventing race conditions. To reimplement the Linux locking primitives required us to use more features of the POSIX threads library, such as conditional waits and mutex data structures.

An example of another major structure that the Linux kernel provides that is not typically found in user-space are memory caches. The kernel has a set of functions for creating a cache of similarly sized memory regions that are used for very quickly allocating commonly used structures. In order to provide this functionality in user-space, we were required to implement our own simple memory cache using a list of pre-allocated memory chunks.

Finally, there were a few pieces of the kernel API that, when translated to user-space were "no-ops". An example of this is the translation between virtual and physical addresses. In the kernel-space, certain operations require a translation between a virtual memory address and a physical memory address (such as DMA transfers). In user-space there is never a need to use physical memory addresses, so these routines were translated into routines that, effectively, do nothing.

3.1.3 Moving to User-Space

The UNH-iSCSI target was the first piece we moved to user-space. The target is able to act in three different modes: *DISKIO* mode[1], *FILEIO* mode[2] and *MEMORYIO* mode[3]. Of these three modes, only the DISKIO mode is required to be in kernel-space. In DISKIO mode, the target sends SCSI commands straight into the Linux SCSI mid-level, an operation that can only happen from within the kernel. In the other two modes (FILEIO mode and MEMORYIO mode) the target does not need to be in kernel-space at all. This design allows

[1] The target uses a SCSI disk for storage. DISKIO mode allows the target to provide raw access to a real SCSI disk drive.

[2] The target uses a file on its local file system to emulate a disk drive. This allows the target to be used in a system that does not have a SCSI disk available. It also allows for debugging the iSCSI implementation without causing any harm to SCSI disks in the target system.

[3] The target uses a memory buffer instead of a disk. This mode allows the iSCSI stack to be tested without worrying about the overhead of the Linux file system.

the target piece to be moved to user-space without significant changes as long as it is not compiled with DISKIO mode support.

The UNH-iSCSI initiator required more work to get it to run in user-space. Since the initiator receives commands directly from the Linux SCSI mid-level[4] it required some redesigning to move it out of the kernel. In order for the initiator to act like a real SCSI initiator device, we needed to devise a way to pass it SCSI commands that it could then transfer to the target. To accomplish this, we wrote a simple interpreter for the initiator. This interpreter reads commands from standard input and performs basic actions like: logging into a target with a new iSCSI connection, sending a SCSI command to the target, and perform a SCSI READ or WRITE operation. This simple interpreter allows us to write scripts that emulate a real SCSI session. In addition to emulating a SCSI session, these scripts collect timing information to determine the performance of transfers.

Timing information, in the user-space initiator, is gathered by using a special `repeat` command. The `repeat` command specifies a $RepeatCount$ and an output file to log to. Following a repeat command can be either a SCSI READ or SCSI WRITE operation. The `repeat` command tells the interpreter to perform the following operation $RepeatCount$ times while keeping track of the total number of bytes transferred and the time that the entire transfer takes. Once the repeated operations are completed the command logs the $RepeatCount$, number of bytes transferred and total transfer time (in seconds) to the specified output file. In order to prevent memory exhaustion, the user-space initiator uses a very simple flow control mechanism when building and queuing new SCSI commands with the iSCSI layer. The `repeat` command tracks command completions and makes sure that it never queues more than a certain number of SCSI command with the iSCSI layer. If this mechanism were not in place, our emulated sessions would not be able to use a $RepeatCount$ that would transfer more bytes than the physical memory of the host computer.

Figure 3-2 depicts a possible configuration for the user-space iSCSI system. This diagram

[4] The UNH-iSCSI initiator registers itself with the operating system as a Host Bus Adapter (HBA).

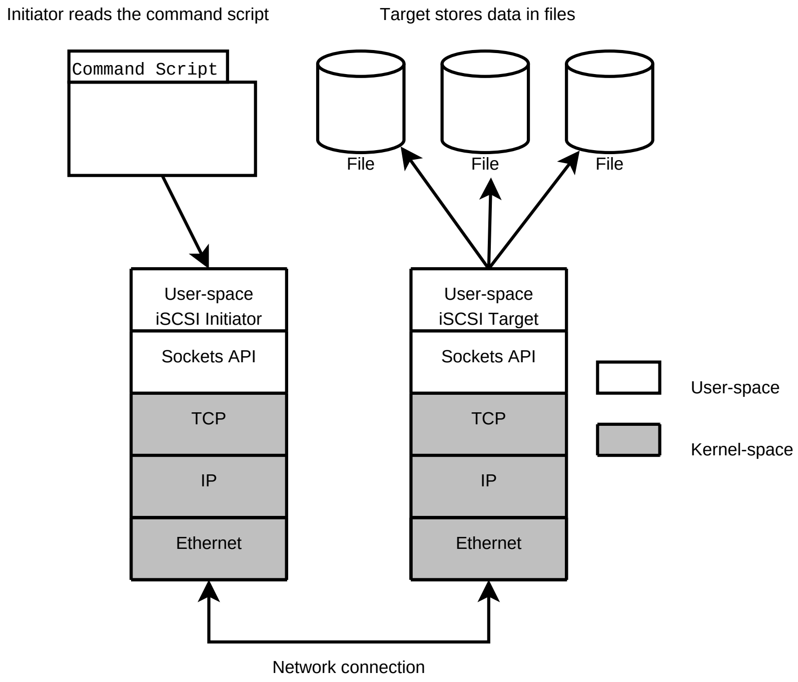

Figure 3-2: User-space iSCSI Implementation

shows the initiator reading from a command script, and a target that is in FILEIO mode using local files as its backing data store. Alternatively the user-space target could use a memory buffer instead of files (MEMORYIO mode), which would allow us to evaluate the performance of the transfer protocols without the overhead of an actual storage system.

3.2 RDMA Interfaces

To have the ability to communicate with RDMA hardware, there are a handful of RDMA interfaces that provide POSIX socket-like abstractions. This type of abstraction is advantageous because most programmers are very familiar the socket communication paradigm. A group called the OpenFabrics Alliance (OFA) provides a free RDMA stack with a socket-like abstraction layer that they call a Communication Manager Abstraction (CMA), along with a verbs layer that is used to perform data transfers (together these pieces are referred to as the *OFA stack* or the *OFA API*). The OFA stack has been growing in popularity due to its active development community and inclusion in the Linux kernel. In addition to kernel support, the OFA stack also provides a user-space library that allows for user-space applications to use its API. We choose to use the OFA API for this project because of its availability and its growing popularity.

Another advantage of the OFA stack is that it can reside on top of either iWARP (which this project focuses on) or Infiniband [5]. There is a growing interest in the computer storage industry to see a comparison between iWARP and Infiniband since they are both competing RDMA technologies. With the ability to use either technology, the OFA API may eventually allow us to make comparisons between iWARP and Infiniband without any significant modifications to our solution.

3.3 Software iWARP

The iWARP protocol suite uses TCP/IP as its method of achieving reliable transport. This feature allows iWARP to be implemented in software using the standard socket interface provided by Linux and other operating systems. Since RDMA hardware is expensive and possibly buggy we choose to use a free, software iWARP stack created by the Ohio Supercomputer Center (OSC) [6] to aid in our development (see Figure 3-1). Version 1.1 of this *software RNIC*[6] provides an OFA-like interface that should be compatible with the true user-space OFA interface. Using this software RNIC, we were able to begin developing with a fully-software, user-space system. This gave us the ability to view the entire project in a software debugger (such as the GNU Debugger, GDB) to aid in finding and fixing errors. Also, since the OSC RNIC provides the user-space OFA like interface, it is interchangeable with the true OFA interface and a real hardware RNIC. The OSC software RNIC gave us the ability to do development in a fully software environment, then switch to hardware

[5] Infiniband is another high performance fabric that offers RDMA support.

[6] RNIC stands for RDMA Network Interface Controller, and refers to the hardware used to perform the iWARP protocol. OSC has called their software iWARP implementation a software RNIC.

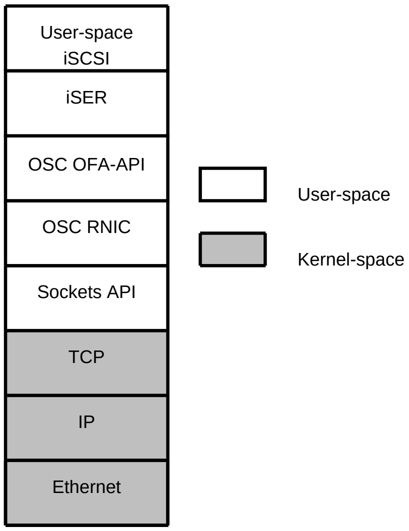

Figure 3-3: iSER Over OSC RNIC

RDMA once we had the system debugged and working. The OFA software RNIC also gave us the ability to develop pieces of our solution when we did not have direct access to our RDMA hardware[7]. Figure 3-3 shows the layout of our user-space implementation when using the OSC software-RNIC.

Another interesting feature of the OSC software RNIC is its ability to provide a cost effective transition mechanism for the deployment of iWARP. The OSC RNIC, on one end of the transmission wire, is able to be used in conjunction with a hardware RNIC on the other end; this combination can allow systems to transition to RDMA without requiring the currently expensive iWARP hardware in every computer. This setup can also drastically reduce the CPU usage on a server with a hardware RNIC performing protocol offloading while clients are using the freely available software RNIC. There are no speed advantages on the client with this setup, but it allows applications to begin adding support for RDMA without requiring clients to purchase any expensive hardware. Figure 3-4 shows a possible topology where client machines with software RNICs are connected to an iSCSI target using

[7] This was very useful to have the ability to continue development while away from the lab

a real hardware RNIC.

We did encounter some difficulties while using the OSC RNIC. It turns out that the OSC OFA interface does not match up exactly with the true OFA interface. When performing large I/O operations, a data type called a *scatter-gather list* (scatterlist) is often used to tie together a list of discontiguous memory buffers in order for them to be used as if they were one single contiguous buffer. The first issue that we encountered with the OSC RNIC was that it does not allow for scatter-gather lists with more than a single discontiguous buffer entry. We were able to resolve this issue, however, by copying scatter-gather lists into single, contiguous, buffers before passing them to OSC. This solution, while easier than spending time to fix the OSC RNIC to properly use scatterlists, adds a lot of extra copy-overhead in both directions. We expect that this extra overhead will create more CPU load on endpoints using the OSC RNIC. Since these endpoints were expected to show a performance decrease anyway (from the extra overhead involved in implementing software iWARP) this turned out to be acceptable for development purposes.

Another issue with the OSC OFA interface is that it was not designed to be used by a multi-threaded application. Originally the OSC RNIC's OFA interface used a simple state machine to emulate the order in which events are reported by the OFA API during connection establishment. The problem with this state machine is that, in a multi-threaded application, events could be missed or reported incorrectly. Additionally, the state machine would never report more than one connection request event. Once the first connection request event was reported the state machine would never return to the state where it listened for another connection. Since our iSER solution is multi-threaded and since we wanted to support multiple connections, we needed to modify the OSC OFA interface to use a queue and locking primitives for incoming events. Using an event queue enables the RNIC to listen for more than a single connection in a separate thread. When a new connection request is made the event is added to the end of the queue so that it can be communicated to the upper layer protocol. This behavior more accurately emulates the behavior of the real OFA API which provides its own event queue to the upper layer protocol.

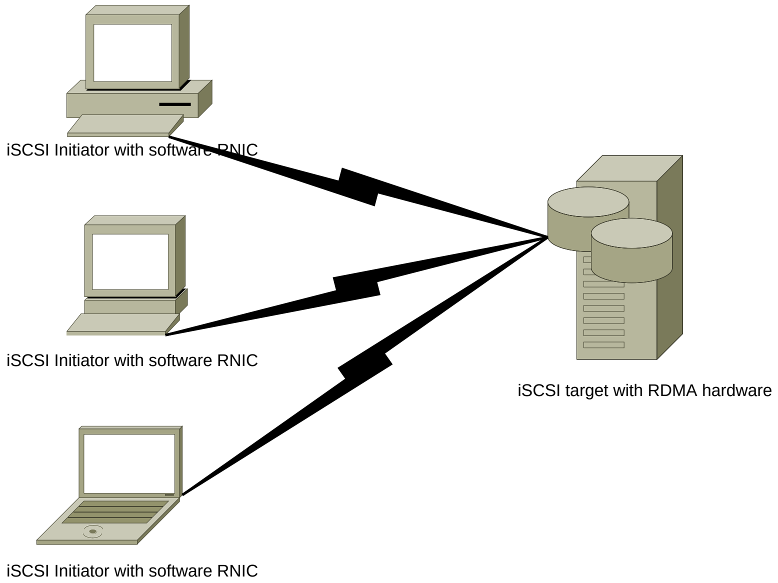
iSCSI Initiator with software RNIC
iSCSI Initiator with software RNIC
iSCSI target with RDMA hardware
iSCSI Initiator with software RNIC

Figure 3-4: Hybrid Software/Hardware RDMA

We also found that the OSC OFA interface does not support completion queues. The OFA stack uses completion queues for user-space applications to get notifications about completed data transfer events. Without completion queues an application is required to poll for completion events in a tight loop. Due to time constraints, we choose not to implement a completion queue in the OSC RNIC. Instead, our solution attempts to alleviate the CPU requirements of a tight polling loop by yielding the processor after a poll returns no events. The extra overhead caused by this tight loop did not prove to be an issue during the initial stages of our development where we made heavy use of the OSC RNIC.

3.3.1 Difficulties With the OFA Stack in User-Space

One of the difficulties that we have encountered using the OFA API is that there is not a lot of documentation for it. We were required to learn the interface based on the source code[8] and example programs that are distributed with it. It took a long time to become

[8] OFA software is all open-source and the code is freely available.

comfortable with using the API. Also, since the API is constantly under development, some of the example programs did not address pieces of the API that have changed or were newly available. There was a lot of trial and error while learning to use the OFA stack.

Another problem that we encountered is that, if not used carefully, there is an inherent race condition in the functions used to retrieve data transfer completion events from RDMA. In user-space, the upper layer protocol (in our case iSER) is notified of completion events by polling a *completion queue* (CQ) structure by using the `ibv_poll_cq` function provided by the OFA API. Since it would be a waste of processor cycles to sit in a busy loop polling this queue, the API also provides a set of three functions to: request notification of events (`ibv_req_notify_cq`), wait for an event notification (`ibv_get_cq_events`) (which merely blocks until an event arrives, at which time the upper layer protocol must poll the CQ to get the event) and to acknowledge the reception of an event notification (`ibv_ack_cq_event`). A naive implementation may use a process similar to the one shown in Figure 3-5 in order to handle events arriving on a CQ. The problem is that if an event arrives after polling, but before re-requesting notifications, the event will be missed until another event arrives and unblocks the call to `ibv_get_cq_events`. In a normal situation, this can cause a deadlock where the remote end is waiting for a response to a request that the local end missed due to this race condition. To fix this issue, an implementation must make sure that there is always a call to `ibv_poll_cq` in between requesting notification events and blocking on a call to `ibv_get_cq_event`.

3.4 Adding iSER Support

The iSER layer that we choose to modify is one taken from the OpenIB iSER target project [15]. This code was given to the OpenIB iSER group under a dual BSD/GPLv2 license, and therefore was freely available for us to use. In addition to being free, an older version of this same iSER implementation was used in a previous, unsuccessful, attempt at adding iSER to UNH-iSCSI [10] and UNH-iSCSI already had some preliminary support for this

```
do:
  ibv_req_notify_cq
  ibv_get_cq_event
  ibv_poll_cq
  ibv_ack_cq_events
  <process the received events>
loop
```

Figure 3-5: Pseudo Algorithm for Processing CQ Events

iSER interface[9].

3.4.1 iSER Modifications

The OpenIB iSER implementation was written to run in the Linux kernel. Since we began our development in user-space it was necessary for us to move this code out of the kernel for it to be able to work with the user-space version of UNH-iSCSI. The approach that we used to move this project into user-space was the same as the one used to move UNH-iSCSI; we wrote user-space implementations of necessary kernel services. Since many of the basic kernel services were already written for UNH-iSCSI we were able to share a lot of code between the two projects, however, some additional functionality was necessary to support features of the kernel that OpenIB iSER used and UNH-iSCSI did not.

Since we choose to use a newer version of iSER than the one used in the previous attempt [10] it was necessary to make modifications to UNH-iSCSI to reflect some changes that were made to the iSER interface. These changes were made so that the iSCSI implementation does not need to maintain extra state information and actually simplified some of the iSCSI code. Most of these updates were minor but required some time to find and fix.

[9] The iSER version that was used by the previous attempt was not be used in this project because of licensing issues.

The previous iSER implementation attempt also used a single per-connection variable in the iSCSI layer to store incoming PDUs from iSER. This approach is not sufficient since there is no guarantee that the iSCSI layer will finish processing a PDU before another PDU has been received. The problem that we found is that PDUs would get lost because the next arriving PDU would overwrite the previous PDU in this single variable before iSCSI was able to process it. In order to fix this issue a simple thread-safe queue was added to the iSCSI interface for receiving PDUs from iSER. In this approach the iSER layer adds received PDUs to the tail of the iSCSI receive queue. When iSCSI is ready to "receive" a PDU, it checks this queue:

- if the queue is empty, iSCSI blocks until a PDU is added to the queue from iSER
- if the queue is not empty, the iSCSI layer processes the front PDU on the queue and removes it when complete.

This approach is used in both the target and initiator.

Another change that was required in the iSER layer is because the OpenIB iSER implementation was, originally, developed using kDAPL [5] as its interface to the RDMA layer. We choose to use the OFA stack in this project, so it was necessary for us to convert the iSER layer to interface with OFA instead of kDAPL. Both kDAPL and OFA have implemented their abstraction layers to be similar to a POSIX socket interface, so it was possible to make a change from one interface to the other without a large impact on the structure of the iSER implementation. Most of the changes that we needed to make, here, were renaming functions, types and constants. There were some issues, however, since OFA uses different terms than DAPL does to refer to various functionalities.

3.5 Moving to Kernel-Space

The advantage of having our project running in kernel-space is that we can send and receive SCSI commands to and from the Linux SCSI mid-level. This means that our iSCSI target can use a real disk drive and the iSCSI initiator can receive SCSI commands passed down from the Linux virtual file system. Unlike the user-space version, the kernel-space iSCSI initiator registers itself with the operating system as a Host Bus Adaptor and can provide the user with real disk access to a connected target. This setup allows us to explore our implementation in a real iSCSI environment (with no emulation). This also gives us a real working product that people can freely make use of for high-performance storage networking.

In our user-space application we implemented Linux-kernel services to use instead of the user-space run-time environment calls, therefore we didn't need to make many large changes in order for our code to compile in the kernel. There are, however, some differences between the OFA user-space API and the OFA kernel-space API. To solve this problem we created an abstraction between the two interfaces so that our iSER layer can use most of the same code for both user-space and kernel-space CMAs. Although this small layer is mainly need to abstract function and type names, there is some more complexity added for event notification and memory registration (which are handled differently in the kernel).

We encountered a large number of difficulties with some of the differences between these two OFA interfaces when building our user-space/kernel-space abstraction layer. The kernel-space interface uses call-back functions to notify the user of events where the user-space interface uses completion queues and polling. The kernel-space interface also handles memory registration a little differently because the kernel does not use virtual user-space addresses for memory buffers. Since the kernel does not have the same protections as user-space applications these differences were difficult to find and fix.

3.5.1 Deferred Event Handling

One substantial difference between user-space and kernel-space is that, in certain kernel-space contexts (such as interrupt or atomic contexts), *sleeping functions* can not be called. Many operations, such as memory allocation, buffer registration/de-registration and waiting on a semaphore can cause the current process to be removed from the scheduler run-queue and effectively be put to sleep for a period of time. In kernel-space there are places where the current control path is not tied to a process or is holding a special locking primitive, that prevent it from begin *safely* put to sleep. Some of these places include: control paths that are locked with a *spin-lock*[10] and atomic contexts such as event notification call-back functions that are called from an interrupt. We run into issues with memory de-registration which can happen with a spin-lock held and with data transfer completion notifications that can happen within an interrupt handler. In order to avoid errors that are introduced by sleeping in one of these contexts, we use a *work queue* structure that is provided by the kernel to defer execution of certain code until it can be executed in a safe context.

3.5.2 Call-back Functions v.s. Queues

The OFA user-space API uses various queue data structures to communicate events arriving from the RDMA hardware to the upper layer protocol which is using the API. There are two main queues which are used in the user-space API: an *event channel* is used to signal the upper layer protocol of connection oriented events (such as, receiving a connection request, the completion of address or route resolution or a connection termination), and a *completion queue/completion channel* pair is used to notify the upper layer protocol of data transfer completion events.

In user-space RDMA event channels are used to signal an upper layer protocol of connection oriented events. In the kernel-space version of the OFA API, a call-back function is used instead of a queue. In order for us to, cleanly, abstract between these two APIs we choose to implement the functionality of an RDMA event channel in our kernel-space abstraction. The reason for this decision is because it encourages a great deal of code sharing since the event handler thread can use the same interface for both user-space and kernel-space. The downside to this approach is that it is less efficient, in kernel-space, to add an event channel than to simply handle events from the call-back function. There are two reasons why this is not an issue:

[10] A mutex locking primitive that waits for a critical section with a busy wait.

1. Connection events do not happen often. The only times that we receive connection events are during connection setup and teardown. These events will not occur in the "fast-path" where speed is critical.

2. Connection event handlers can not be used in atomic context. In our experience, the call-back functions from the OFA API may happen from an atomic context, one in which we do not have the ability to call function which may cause a reschedule of the current thread. Some of the operations that we perform during our connection event handlers (such as memory allocation) may cause the current thread to sleep. If a thread sleeps in an atomic context the system may deadlock, and this is strictly prohibited by the Linux kernel. In order to handle our events outside of an atomic context, we would need to queue them up for a separate thread to quickly process, which is exactly what we do with our event channel implementation.

In user-space data transfer completion events are handled by using two different queues: a *completion channel* and a *completion queue*. The completion queue (CQ) holds completion event structures. This CQ can be polled in order to get the completion events structures that contain information about completed data transfers. The completion channel is used to signal the upper layer protocol of the availability of events on the CQ by allowing the user to block while no events are available. In the kernel the upper layer protocol (such as our iSER implementation) is notified of completion events by a call-back from the RDMA device's interrupt handler routine. Our user/kernel space abstraction layer handles both of these interfaces while sharing as much code as possible to avoid duplication and to avoid the need to make changes in two places for a common bug fix. The way in which we handled this difference between the two interfaces is by making use of a common function which polls a completion queue for events. In the user-space a separate thread is created for each connection whose only job is to sit blocked on a call to `ibv_get_cq_event` and to call the shared poll function when it is awakened by an event. In kernel-space, our event notification call-back function makes use of a deferred work queue to schedule a call to the shared completion queue polling function once a safe context is available. The reason that we choose not to implement our own completion channel in kernel-space is because data completion events need to be handled quickly. We rely on the kernel's deferred work handling to allow us to process these events in a safe context, as a tradeoff we lose a little bit of code sharing.

3.5.3 Memory Registration

With the kernel-space OFA API, memory must be explicitly mapped for DMA transfers. In order for us to ignore these details throughout most of our iSER code, we choose to register memory for DMA operations in the kernel-space abstraction code while registering the memory with the RDMA subsystem. While this sounds obvious it was actually difficult due to very complicated memory buffer handling inherited from the original iSER layer. Since our iSER layer was originally developed to run in Linux 2.4 kernels there was a lot of logic for dealing with conversion between virtual memory regions and physical memory pages. In more current versions of the Linux kernel these details are handled by functions provided by the Infiniband subsystem, which contains the kernel-space OFA API. In order to make sure that we are not converting or mapping memory regions multiple times it was necessary to track down the old procedures and remove them, or modify them in such a way that they do not conflict with the new ones.

Our implementation deals with two different general classes of registered memory:

1. Header and immediate data buffers.

2. Non-immediate data buffers (used for the bulk of data transfers).

Figure 3-6 shows these two registered memory types for both the target and the initiator.

The header and immediate data buffers are fairly small. Each SCSI command is encapsulated in an iSCSI header with an iSER header placed before it. These commands are transmitted using an RDMA Send type message, and are received into pre-posted RDMA Recv buffers in the receiver's untagged buffer queue. Since memory registration can be

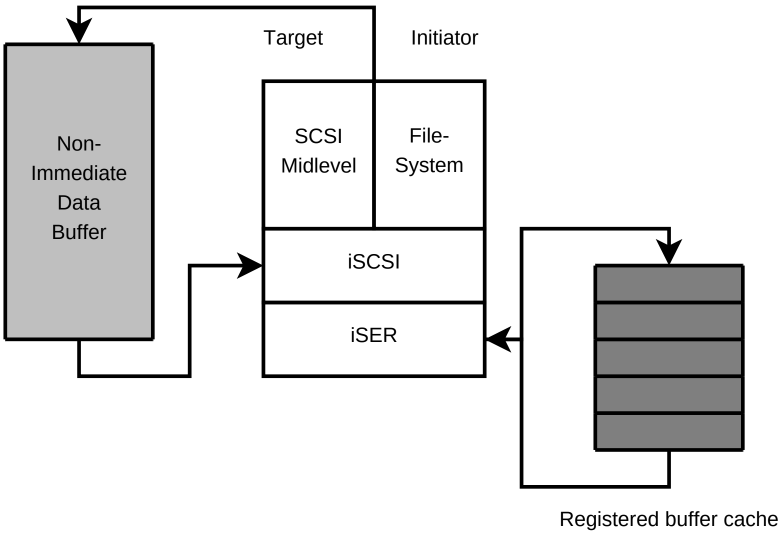

Figure 3-6: Registered Memory Regions

quite costly, we choose to cache these registered buffers to avoid extra register/unregister operations for small transfers. When a header buffer is required we can pull one off of the cache that is already registered instead of allocating and registering a new one. After each Send or Recv operation is complete, we can push the buffer back onto the cache instead of unregistering and deallocating it.

The Non-immediate type of data buffers are passed to iSER from the iSCSI layer and are registered and mapped before each data transfer. In the case of the target, these buffers are allocated by the SCSI mid-level and are passed to iSCSI. On the initiator side these buffers are passed down from the filesystem layer. In both of these cases, we have a specific memory region that we are required to use for the data transfer. We register these regions before each transfer and we unregister them when a transfer completes. It is not helpful to cache these data buffers since they are not allocated by the iSER layer and there is no guarantee that we will use one of these buffers for more than a single transfer. Another approach, that could be used to alleviate buffer registration overhead, is to cache these non-immediate buffers with the assumption that the SCSI mid-level or file system layer will use them for subsequent transfers. A *least-recently-used* policy could be adopted to handle removing stale buffers from the cache when memory is low.

In SCSI, scatter/gather lists (also called scatterlists, or sg lists) are often used to describe non-immediate data buffers[11]. A scatterlist is an array of address/length pairs that map a discontiguous set of memory pages to a contiguous buffer. Figure 3-7 shows how a scatter/gather list is used to perform this mapping. In this figure, main memory is shown as an array of pages on the left and the virtually-contiguous buffer that the scatterlist maps these pages to is shown on the right. Pages in the buffer are labeled A-F in order to help demonstrate this mapping. Also, note that the third element in the scatterlist which refers to the page labeled "E" has a length of two so it also maps the page labeled "F", which directly follows "E", onto the buffer. Scatterlists are heavily used for large data transfers since memory fragmentation often prevents the allocation of very large contiguous buffers. Scatterlists are often given to iSCSI (and eventually to iSER) from the Linux SCSI mid-level and from the file system.

Using the OFA API, we are able to register an entire scatterlist as a single registered memory region. This means that an Stag can be associated with an entire scatter/gather buffer. Each RDMA Read or Write operation may then consist of a vector of multiple, registered, scatter/gather buffers depending on the RDMA hardware that is being used.

To be able to support the OSC software RNIC (which only supports using a single scatterlist per RDMA operation) and RNICs with more strict memory buffer limitations we were required to add routines to merge and re-expand scatterlist entries. These routines take a scatterlist and copy it into a single contiguous buffer before registering and transferring it. Upon completing a data transfer where a scatterlist was merged, we copy the contents of the contiguous buffer back into the scatterlist. While this copy-overhead kills performance, it is necessary in cases where it is impossible to perform RDMA operations

[11] As of Linux 2.6.25, it appears that scatter/gather lists are *always* used with the SCSI mid-level to describe these buffers.

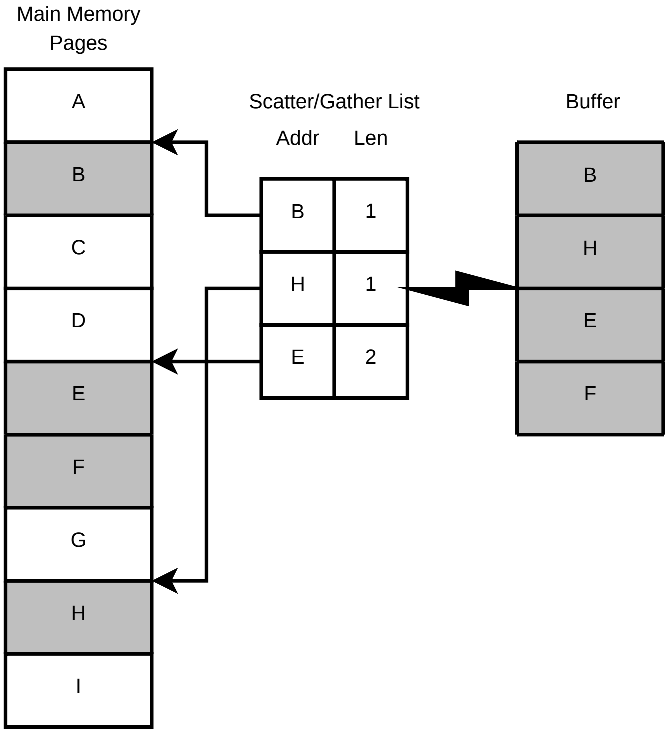

Figure 3-7: Scatter/Gather Lists

with larger scatterlists. In the case of the OSC software RNIC this performance cost is not an issue since the advantages of the software RNIC do not include performance. We have not encountered a hardware RNIC that requires these merge/expand operations of scatter/gather lists, however, we feel that a slower transfer is still better than no transfer at all.

Another issue that we ran into throughout our development is that the OFA stack does not give a lot of information about errors that may occur. The kernel-space interface returns a single error number from most function calls, however, even with verbose debugging enabled there is often no extra information reported beyond the very general error number. This makes it very difficult to track down the cause of an error. In many cases the CMA or RNIC driver code needed to be edited to add additional logging for error paths. Another approach that we used to get better error reporting was to duplicate the parameter checking performed by the device drivers so that we could report errors that would occur when calling certain device driver functions. These issues tended to be very time consuming (since they often required re-compilation of the kernel). Similar difficulties were found in the user-space interface, however, these were not nearly as difficult to track down since we were able to use an interactive debugger.

We also found that it was difficult to determine whether or not issues were in our code or in the driver/hardware code. Since RDMA is a new technology, even the core Linux support for it is not extremely well tested. We found that changes to the kernel, driver code and firmware often surfaced new bugs and issues in our implementation. Another interesting thing to note on this issue is that the user-space implementation seemed to be much less effected by kernel-code or driver changes than the kernel-space implementation was. Due to the fact that we were originally attempting to track some of the most up to date Linux kernels, we tended to encounter these bugs fairly often. During the time that we were doing our development the Linux kernel seemed to make some fairly significant changes to the SCSI mid-level and the scatter/gather list systems. These changes required us to be constantly fixing glitches and adding macros to abstract functionality that is different between different kernel versions.

3.6 iSER Difficulties

One of the problems that we encountered with our iSER implementation is that there are large differences between the way that the iSER standard [8] expects RDMA devices to behave and the way that they actually do behave. We found that there are three notable places where current hardware is unable to conform to the the iSER standard.

3.6.1 TCP Stream Transitioning

While the iSER standard was being written it was believed that iWARP devices would have the ability to perform *TCP stream transitioning*. The idea behind TCP stream transitioning is that a TCP socket could be opened in the regular streaming mode and eventually transitioned into an RDMA Capable TCP session using iWARP (hence the iSER `Enable_Datamover` operational primitive). This ability would let applications negotiate the use of RDMA mode and, if selected, they could "turn on" RDMA mode on-the-fly. In other words RDMA mode would be an option that could be enabled on an already established TCP connection.

The iSER specification takes advantage of the fact that an iSCSI login consists, mainly, of the negotiation of various parameters; iSER states that an RDMAExtensions parameter must be negotiated to "Yes" by both the initiator and the target in order to use RDMA mode. The login phase of an iSCSI connection is expected to happen in regular, streaming TCP mode and, if RDMAExtensions was negotiated to "Yes", the devices will transition the TCP connection to RDMA mode after the login is complete. Once in RDMA mode, before beginning the full feature phase of the iSCSI session, the devices must also perform a simple Hello message exchange to establish some RDMA specific parameters (namely the *Inbound RDMA Read Queue Depth* and the *Outbound RDMA Read Queue Depth*, which is the maximum number of outstanding, unexpected read requests that the device can handle for a given connection).

The problem is that current RDMA hardware does not perform TCP stream transitioning. A connection between two RDMA devices starts either in TCP streaming mode or in RDMA mode using iWARP, and there is no way to switch between the two modes without establishing a completely new connection. This makes it impossible, with current RDMA hardware, to adhere to a portion of the iSER specification.

Since RDMA mode must be enabled from the start, our iSER target either only listens for RDMA connections or only listens for traditional iSCSI connections on the iSCSI *well known port number*. Our implementation does not support the run-time negotiation of the use of iSER-assisted mode as stated by the iSER standard. This approach requires the administrator of a storage network to choose if a target will communicate with iSER-assisted devices only or with traditional iSCSI devices only, never both. An alternative approach, that is feasible with current hardware, is to allocate a new well known port number for iSER. This would allow a target to distinguish between iSER and traditional iSCSI connections based on the port number that an initiator device is connecting to.

Additionally, since we determine whether or not we are using RDMA mode at compile time[12], there is no real need to negotiate the RDMAExtensions key in the iSCSI login. Our target implementation negotiates this key anyway, however, if the key is negotiated to "No" while compiled for RDMA mode the behavior is undefined[13]. Likewise if the key is negotiated to "Yes" while compiled for traditional iSCSI mode the behavior is also undefined.

Another complication caused by this is that we had to choose where to perform the hello exchange. The IRD value needs to be negotiated very early in an RDMA connection

[12] This could be done with run-time configuration, however, this feature is not yet implemented

[13] The behavior in this situation is undefined because some pieces of the implementation check the value of this key to determine which mode they are operating in. If the key's value does not match the compile-time value some control paths break.

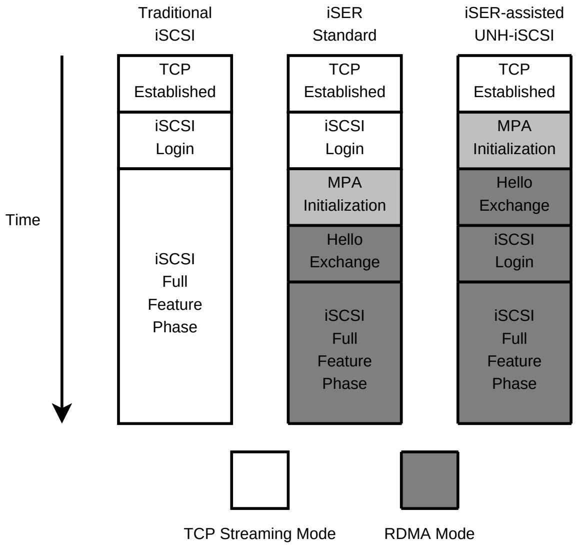

Figure 3-8: TCP Stream Transitioning, Theoretical V.S Actual

in order for the RDMA layer to allocate the proper amount of resources to handle all of the possible RDMA ReadRequest messages that it may receive. For our implementation, we choose to perform the hello message exchange *before* the iSCSI login phase instead after it. This way, our RDMA session is aware of all of the proper parameters before data is transferred.

Figure 3-8 shows three iSCSI connection modes. The left most diagram shows a traditional iSCSI connection with no RDMA. The center diagram shows the iSER specification's vision of how TCP stream transitioning should work with an iSER-assisted connection. Finally the right most diagram shows how iSER-assisted mode works with current hardware and UNH-iSCSI. Notice that the SCSI login phase on the traditional iSCSI and on the iSER standard iSCSI line up side-by-side. It is easy to see that, if stream transitioning were possible, a negotiation in the iSCSI login phase could allow a connection to choose to be transitioned into RDMA mode, or to choose to remain in traditional TCP mode.

Because of our requirement to start a TCP connection in RDMA mode some of the iSER standard's descriptions of the iSER/iSCSI interface no longer make sense. Various operational primitives are suppose to be used at specific times during an iSER-assisted session, for example the `Allocate_Connection_Resources` primitive (which allocates RDMA specific resources for a connection) must now be used by iSCSI before the login phase, instead of before the iSCSI full feature phase as stated by the iSER standard. Other operational primitives, such as `Enable_Datamover` (which is suppose to perform the TCP stream transition), are completely unusable with current implementations. Our target and initiator make use of these operational primitives in places where we felt they made the most sense, however this does not necessarily follow the iSER specification.

Finally, an extended set of operational primitives needed to be added in order for the iSCSI layer to establish an RDMA capable connection. If TCP stream transitioning truly existed, a connection would be established by the traditional means, using the operating system socket API. Since stream transitioning does not exist in current RDMA hardware, operational primitives were added to the iSER layer to provide the following services:

Connection_Establish is invoked by the iSCSI initiator in order to tell the iSER layer to establish a new RDMA connection to an iSER-enabled target.

Connect_Accept is invoked by the iSCSI target in order to tell the iSER layer to accept an RDMA connection request made by an iSER-enabled initiator.

Additionally, the iSCSI layer must provides the following operational primitives so that the iSER layer can notify it of new event types:

Connection_Establish_Notify is invoked by the iSER layer on both the initiator and target in order to notify the iSCSI layer that an RDMA connection is established and ready for an iSER-enabled iSCSI login.

Connection_Request_Notify is invoked by the iSER layer on the target in order to notify the iSCSI layer of a new iSER-enabled, RDMA, connection request by an initiator device.

3.6.2 Zero-based Virtual Addressing

While setting up for an RDMA data transfer, the upper layer protocol is required to make an advertisement for the memory buffers that data will be transferred to and from. In the case of an RDMA Read operation, the iSCSI initiator must advertise the memory region that it would like the target to read from. In the case of an RDMA Write operation, the iSCSI initiator must advertise which memory region it would like the target to write into. According to the iWARP protocol suite a buffer advertisement consists of three pieces of information:

1. The *Steering Tag* (STag) for the buffer.[14]

2. The length of the data that will be transferred.

3. The *tagged offset* (TO) for the transfer.

These three pieces of information describe everything needed for the RNICs to perform the data transfer from the memory of one machine into the memory of the other[15].

The problem that we encountered is that there are two possible interpretations of the tagged offset field. The first interpretation is referred to as *Zero-based Virtual Addressing.* In this interpretation a tagged offset value of zero refers to the base address of the memory region specified by the STag. Figure 3-9 shows how the three fields of a buffer advertisement are used to find the specified memory region using Zero-Based Virtual Addressing. Using this interpretation it is not necessary to advertise the tagged offset for a buffer if all transfers are to start at the base of the buffer (offset zero).

The other interpretation of the tagged offset field is that it contains the address of the tagged buffer. Since RDMA requires that a buffer advertisement contains the Stag, Tagged Offset and length of a tagged buffer, a remote peer will still have enough information to refer

[14] The steering tag is an opaque handle that uniquely describes a registered memory region.

[15] Actually these three pieces of information must be given for both the source and destination buffers, however, this information for the local buffer does not need to be transferred over the wire.

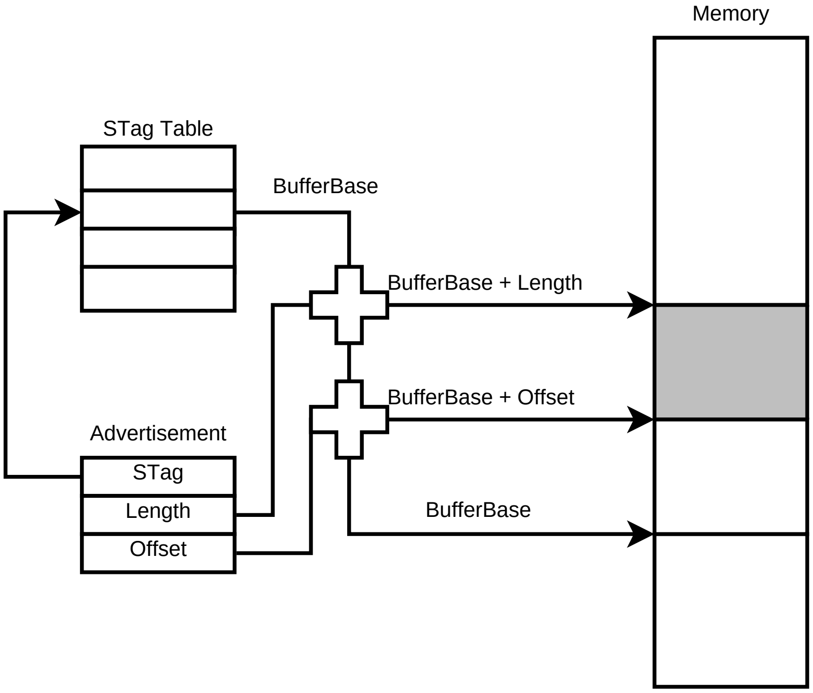

Figure 3-9: Zero-based Virtual Addressing

to the correct buffer for RDMA operations. The difference is that the tagged offset value advertised refers to an address instead of an actual offset. In this interpretation an offset of zero does not refer to the base of a registered memory region, and will most likely be invalid. Using this interpretation the advertisement of the tagged offset is required, since it refers to an address and will be used by the RNIC to find the base of a buffer. This approach is advantageous because the tagged offset field is 64-bits (as opposed to the 32-bit STag field) and is large enough to contain an entire memory address on a 64-bit architecture. RNICs that use this approach have the ability to store the buffer address directly in the tagged offset, and can circumvent the need to do a lookup on the STag for each transfer request. Figure 3-10 shows how an advertisement that does not use Zero-based virtual addressing may be used to locate a memory region.

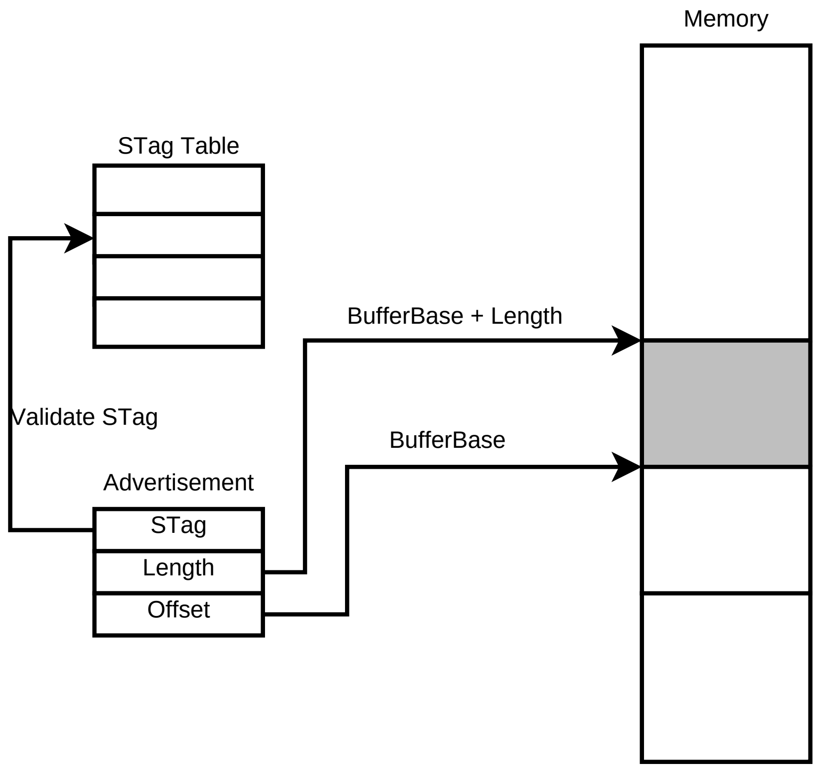

Figure 3-10: Non-Zero-Based Virtual Addressing

The specific problem we found is that the iSER specification does not advertise tagged offset values. The iSER buffer advertisement only advertises STags[16] with the assumption that the tagged offset is implicitly zero. If iWARP hardware used Zero-Based Virtual Addressing this would not be a problem, however, current iWARP hardware does not use Zero-Based Virtual Addressing, instead it takes advantage of the size of the tagged offset field to hold the memory address of an advertised buffer. This means that, in order to fully specify a memory region, an extra set of fields needs to be added to the iSER header to advertise tagged offset values for a tagged buffer. Figure 3-11 shows the two different iSER headers: the one specified by the iSER standard and the one use by UNH-iSCSI.

In this figure it is apparent that a conformant iSER implementation would not be interoperable with the UNH-iSCSI implementation since it would not understand the two extra fields that we have added. With current iWARP hardware, however, a conformant iSER implementation can not exist.

3.6.3 Send with Solicited Event and Invalidate

The iSER specification requires an iSER-assisted target to use the RDMAP Send with Solicited Event and Invalidate operation when transmitting iSCSI/SCSI response PDUs. Currently, the OFA API does not support Send with Invalidate type messages, so our implementation uses a standard Send type message in order to transmit these PDUs. The Open Fabrics Alliance does have plans to add support for Send with Invalidate type messages in the next release of the Linux kernel. Since Send with Solicited Event and Invalidate operations only provide a minor increase in efficiency and security, this does not cause any issues with our implementation.

[16] The transfer length is available through the iSCSI header that is transferred along with the iSER advertisement.

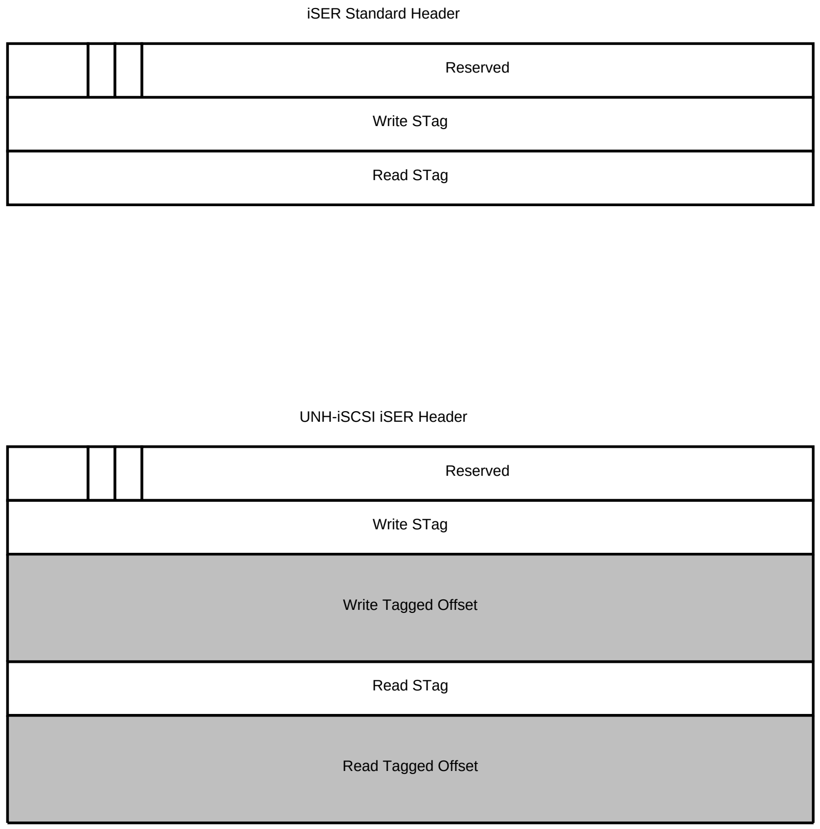

Figure 3-11: iSER Headers

3.7 The Completed Implementation

3.7.1 iSER Layer Layout

The iSER layer resides between UNH-iSCSI and the OFA API, as shown in Figure 3-1. Our iSER layer provides one of two different sets of operational primitives to the iSCSI layer above it: one for an iSCSI target and one for an iSCSI initiator. These primitives use a common set of functions in order to maintain the state information required for iSER and to perform command and data transfers.

Establishing an iSER connection is performed by a simple state machine, as shown in Figure 3-12. The passive connection on the iSCSI target is established as follows: The iSER layer, upon the registration of a new iSCSI target, uses the OFA API in order to tell the RDMA hardware to listen for new connection requests. When a new connection request event arrives at the target's iSER layer the extended operational primitive `Connection_Request_Notify` is invoked to signal the iSCSI target of a new connection request. The iSCSI target will then invoke the `Connection_Accept` primitive on the iSER layer, and the target's iSER layer will tell the OFA layer to accept the connection. Once the connection has been accepted, the target's iSER layer waits for the arrival of a Hello message from the initiator. After the Hello exchange, the connection is ready for the iSCSI login phase, and the target's iSCSI layer is notified with the `Connection_Establish_Notify` operational primitive. Figure 3-12 shows the process described above. A summary of the Hello message exchange is shown in this figure in order to simplify it. In the actual process, the Hello messages are exchanged by performing RDMA Send and Recv operations, which would add many more steps to the diagram.

The iSER initiator has a more complicated state machine for making an active connection to the iSER-assisted iSCSI target, as shown in Figure 3-13. When the iSCSI initiator layer invokes the `Connection_Establish` primitive on its local iSER layer, iSER begins the connection by asking the OFA layer to perform address resolution. The address resolution attempts to find which network interface card to use for this connection. If the OFA layer

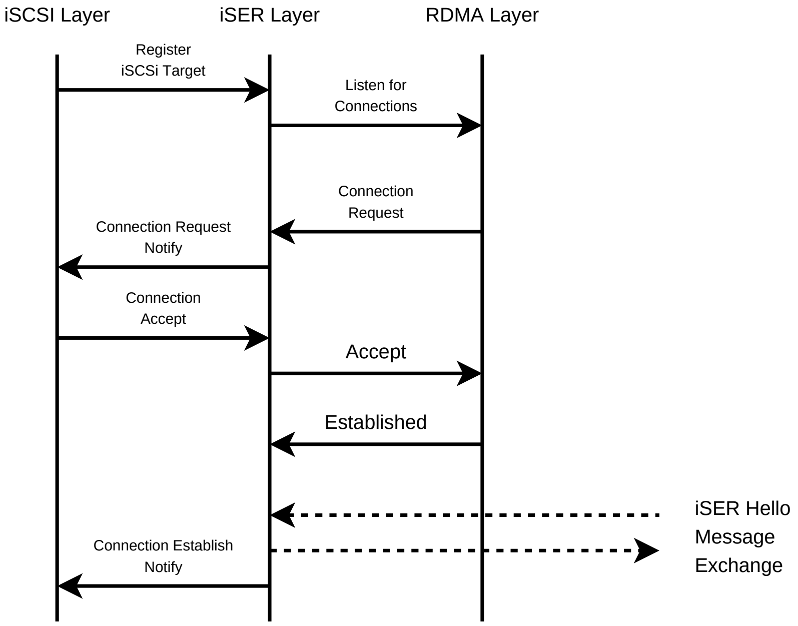

Figure 3-12: iSER Target-Side Connection Establishment

was able to resolve the address, it signals the initiator's iSER layer, which then requests that the OFA layer perform a route resolution. When the route has been resolved, the initiator's iSER layer tells the OFA layer to make a connection to the iSCSI target. Once the connection has been established the initiator's iSER layer enters the Hello message exchange. At the completion of the Hello message exchange the iSER layer at the initiator invokes the `Connection_Establish_Notify` primitive on the iSCSI layer in order to notify the initiator's iSCSI layer that the connection is established. This procedure is shown in Figure 3-13. Note that, as with Figure 3-12 the Hello message exchange has only been shown abstractly in order to simplify the diagram.

The iSER layer on both the initiator and target is notified of network events from the RDMA hardware by the OFA stack in two ways. Connection events, such as those described above, are sent to the iSER layer by an event channel[17]. The iSER layer contains a per-

[17] In kernel-space these events use call back functions instead of a channel, however, our approach implements a channel with these call back function in order to promote code sharing.

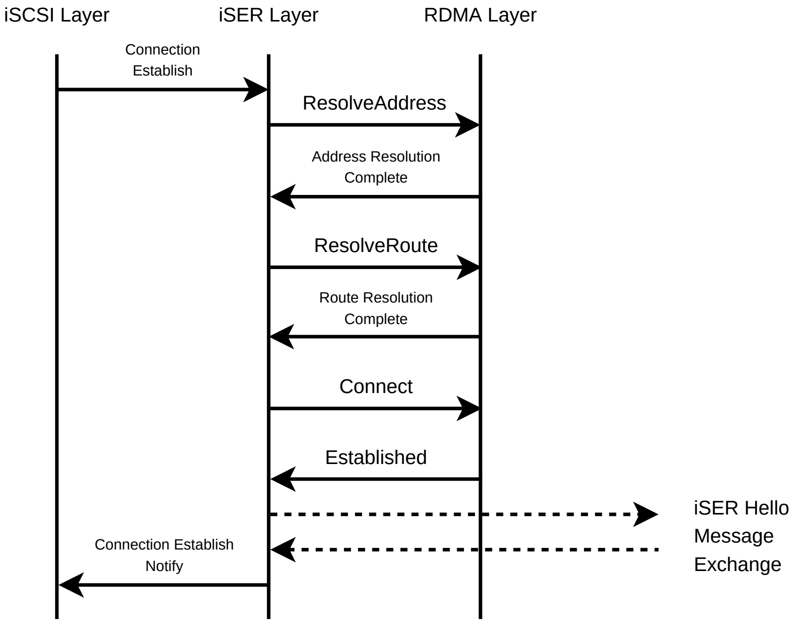
iSCSI Layer
iSER Layer
RDMA Layer
Connection Establish
ResolveAddress
Address Resolution Complete
ResolveRoute
Route Resolution Complete
Connect
Established
iSER Hello Message Exchange
Connection Establish Notify

Figure 3-13: iSER Initiator-Side Connection Establishment

adaptor thread which waits for new connection events on the OFA event channel. Each time an event arrives, this per-adaptor thread does the processing of the event and then returns to wait for another one. The other type of event that the OFA stack will generate is called a *work completion.* Work completion events signal the completion of *work requests* that were submitted by the iSER layer. Work requests are generated by iSER for all of the types of data transfer operations (Sends, Receives, RDMA Writes and RDMA Reads). Work completions are added to the end of a completion queue by the OFA stack, and the iSER layer, in user-space uses a per-connection thread to poll this queue in order to receive work completion events. In kernel-space, work completion events are notified through call back functions, in which the iSER layer will create a deferred work structure that will poll the queue in a shared kernel worker thread.

3.7.2 iSCSI Target

The UNH-iSCSI target consists of four main threads of control:

1. The server thread.

2. The SCSI target process thread (for all modes except DISKIO mode).

3. The (per-connection) transmit thread.

4. The (per-connection) receive thread.

The UNH-iSCSI target's *server thread* listens for and accepts new iSCSI connections. In the case of an iSER-assisted session, this thread blocks until it is signaled by a call back from the iSER layer that a new connection request has been made.[18] Upon receiving a new connection request, the iSCSI target accepts the connection by invoking an extended iSER operational primitive called `Connection_Accept`. This extended, operational primitive tells the iSER layer to accept the connection and to perform the Hello message exchange. Once the Hello message exchange has been completed the iSER layer signals the target server thread that the connection setup is completed. When the connection setup has completed, the target server thread spawns a transmit and receive thread which effectively begins the new connection.

The *target process thread* exists only for MEMORYIO and FILEIO modes. This thread is used to emulate the Linux SCSI mid-level. In MEMORYIO and FILEIO modes, functions are provided in order to notify the target process thread of new SCSI commands. Upon receiving a new command, the target process thread will perform the given service using either a memory buffer (for MEMORYIO mode) or a file on the local file system (in FILEIO mode) as the backing data store. When services have been completed, this thread notifies the iSCSI transmit thread so that an appropriate action can take place. In the case of

[18] This call back function is part of a non-standardized, extended iSER operational primitive set that we inherited from the OpenIB iSER target project's code base. Without the ability to transition a traditional TCP connection into an RDMA capable connection, the iSER operational primitives do not provide the functionality to listen for and accept new connections. These extended operational primitives have been added in order to support current RDMA hardware which does not perform TCP stream transitioning (see Sections 2.4.9 and 3.6.1).

DISKIO, these actions are performed by the true Linux SCSI mid-level, using a real SCSI disk device as the backing data store.

The UNH-iSCSI target's per-connection *transmit thread* performs one half of the work for an iSCSI connection. This thread is woken up by events that occur in the *receive thread* and the Linux SCSI mid-level. Upon waking up, the transmit thread will look through all of the pending iSCSI commands for the connection for which it is responsible. If any of these commands are in a state which requires an action to be performed, the transmit thread will perform that action. An example of an action that the transmit thread may perform is transmitting an iSCSI PDU (hence the name "transmit thread"). The iSER `Send_Control`, `Put_Data` and `Get_Data` operational primitives are used, in iSER-assisted mode, by the target transmit thread in order to transmit iSCSI PDUs on an RDMA capable connection. The iSER layer will use the `Data_Completion_Notify` operational primitive to signal the transmit thread when a data transfer has completed. The transmit thread can then signal the SCSI mid-level of the transfer completion and get ready to send an iSCSI/SCSI response.

The UNH-iSCSI target's per-connection *receive thread* performs the other half of the work for an iSCSI connection. The receive thread waits for iSCSI PDUs to arrive on the network. When an iSCSI PDU arrives, the receive thread performs the initial actions for this PDU. Some initial actions include: progressing the login phase, storing data, or notifying the SCSI mid-level of a new command. In an iSER-assisted session the iSER layer uses the `Control_Notify` operational primitive in order to enqueue new PDUs for the target receive thread. Each time the iSER layer invokes the `Control_Notify` primitive on the iSCSI layer, the new PDU is added to the tail of a receive queue and the receive thread is woken up in order to process it.

Figure 3-14 shows the interaction between the three iSCSI target threads in UNH-iSCSI along with the interaction with the iSER layer and the SCSI mid-level. The interaction between the iSCSI target threads and the iSER layer happens on behalf of the thread of control that is invoking an operational primitive. This means, for example, if the target transmit thread invokes an operational primitive on the local iSER layer, the target transmit

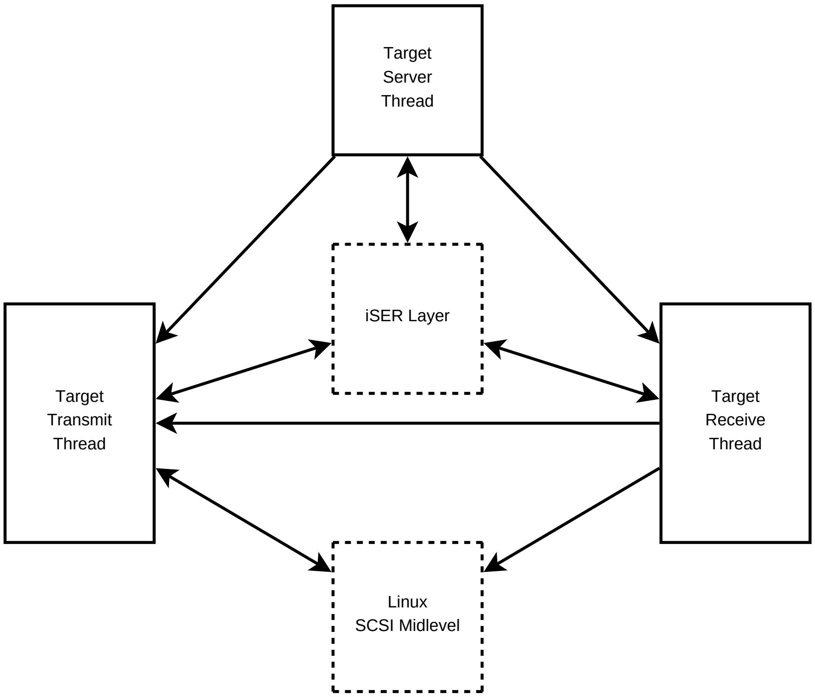

Figure 3-14: iSCSI Target Thread Interactions

thread drops down to the iSER layer (via a function call) in order to begin the desired service. Likewise, when the iSER layer's event processing thread invokes an operational primitive on the iSCSI layer (again, via a function call), it raises up into the iSCSI layer to begin the notification. Figure 3-15 shows this interaction between the iSCSI and iSER layers. The dashed arrows between threads represent communication through shared structures and semaphores and the solid arrows represent the function calls that move between the iSCSI and iSER layers[19].

3.7.3 iSCSI Initiator

The UNH-iSCSI initiator consists of two main threads of control.

1. The initiator (per-connection) transmit thread.

[19] A similar interaction also occurs between the iSCSI and the SCSI layer, however this document does not cover that interaction in great detail.

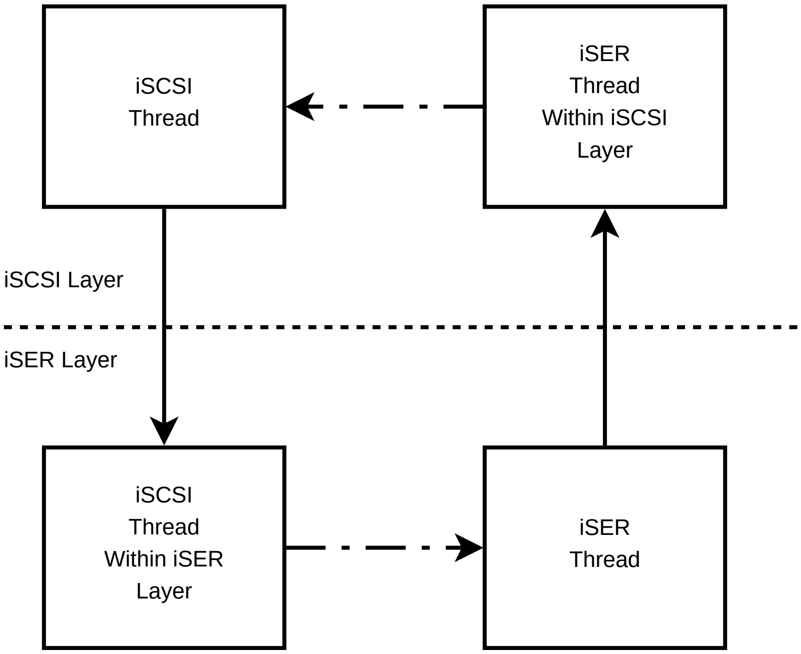
iSCSI Thread
iSER Thread Within iSCSI Layer
iSCSI Layer
iSER Layer
iSCSI Thread Within iSER Layer
iSER Thread

Figure 3-15: iSCSI/iSER Layer Interaction

2. The initiator (per-connection) receive thread.

The initiator *transmit thread* handles the transmission of iSCSI PDUs. Since the initiator registers itself as a Host Bus Adaptor with the operating system, it is required to implement a `queue_command` function that allow the SCSI mid-level to pass it new SCSI commands. Each new command is added to the initiator's queue by the `queue_command` function, and the transmit thread is woken up in order to transmit the new command to the iSCSI target. The `Send_Control` operational primitive is used by the iSCSI initiator's transmit thread in order to give the local iSER layer each new iSCSI PDUs to transmit to the target.

The initiator *receive thread* waits for new incoming iSCSI PDUs on the network. The initiator provides the `Control_Notify` and `Data_Completion_Notify` primitives to the iSER layer so that it can receive notification of new incoming iSCSI PDUs or data from the RDMA capable connection. Upon receiving an iSCSI response PDU indicating that the

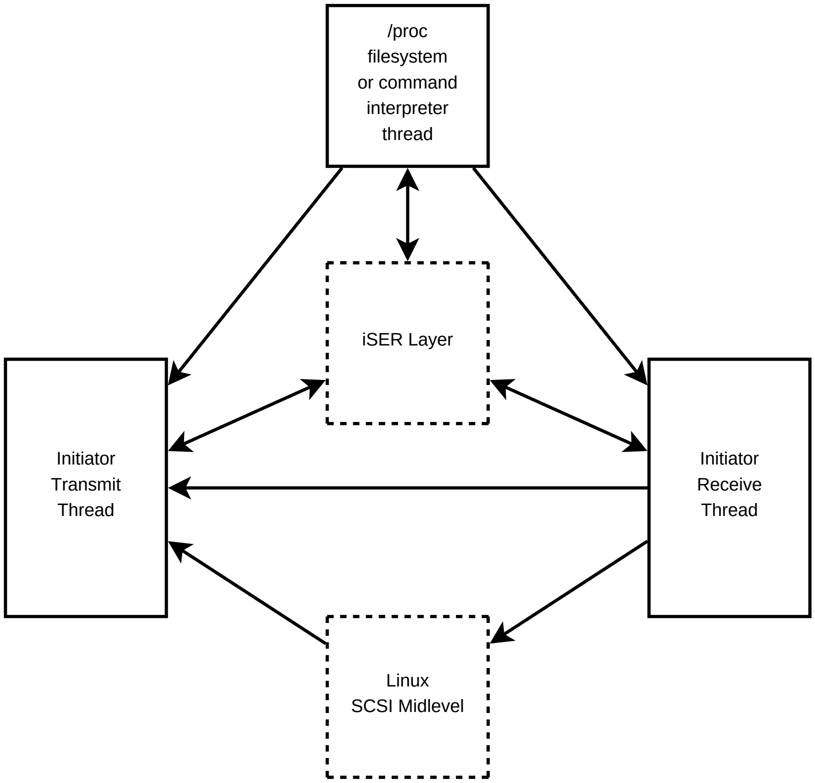

Figure 3-16: iSCSI Initiator Thread Interactions

iSCSI target has completed servicing the request for a previous iSCSI Command PDU, the initiator receive thread will signal the SCSI mid-level that the command has completed.

In order to create the initiator's transmit and receive threads for a new connection, the `/proc` file-system is used in kernel-space mode and a command interpreter thread is used in user-space mode. Writing certain commands to the `/proc` file system entries that correspond to the iSCSI initiator (or to the command interpreter, in the case of user-space) will cause a new connection to be created. A new connection, in iSER-assisted mode, is created by invoking some of the extended operational primitives similar to the ones used in the target. Once the local iSER layer has completed making the connection to the target, the initiator receive and transmit threads are created and the new iSCSI connection begins.

Figure 3-16 shows the interaction between the iSCSI initiator threads. As with the iSCSI target, the interaction between the iSCSI and iSER layer happens on behalf of the thread that is invoking an operational primitive (refer to Figure 3-15).

CHAPTER 4

RESULTS

Our implementation is designed to support all of the protocols shown in Figure 3-1. It is also designed to run in user-space and kernel-space and to use the three UNH-iSCSI IO modes (DISKIO, MEMORYIO and FILEIO). This design should be able to perform a large variety of test combinations. Table 4.1 gives a table of possible test combinations. FILEIO mode, while useful in development, is not something that we consider interesting with respect to performance results since it does not tell us any useful information pertaining to the real world use of our implementation. DISKIO mode is interesting since it demonstrates the system while using a real disk drive, and MEMORYIO is useful to demonstrate the performance of just the protocol stack (without the disk overhead). Also, one should note that evaluation runs with the target environment of user-space can not use DISKIO mode because of previously mentioned restrictions on using the Linux SCSI mid-level.

4.1 Tests Performed

Due to time constraints, all of the tests shown in Table 4.1 were not performed. The experiments performed, for this thesis, used the following subset of the possible configurations:

- iSER-assisted kernel-space target and initiator over iWARP with MEMORYIO mode.
- Traditional iSCSI kernel-space target and initiator with MEMORYIO mode.
- iSER-assisted user-space target and initiator over iWARP with MEMORYIO mode.
- Traditional iSCSI user-space target and initiator with MEMORYIO mode.

Initiator Transport	Target Transport	Initiator Environment	Target Environment	IO Mode
TCP	TCP	User-space	User-space	MEMORYIO
TCP	TCP	Kernel-space	User-space	MEMORYIO
TCP	TCP	User-space	Kernel-space	MEMORYIO
TCP	TCP	User-space	Kernel-space	DISKIO
TCP	TCP	Kernel-space	Kernel-space	MEMORYIO
TCP	TCP	Kernel-space	Kernel-space	DISKIO
OSC	OSC	User-space	User-space	MEMORYIO
iWARP	OSC	User-space	User-space	MEMORYIO
iWARP	OSC	Kernel-space	User-space	MEMORYIO
OSC	iWARP	User-space	User-space	MEMORYIO
OSC	iWARP	User-space	Kernel-space	MEMORYIO
OSC	iWARP	User-space	Kernel-space	DISKIO
iWARP	iWARP	User-space	User-space	MEMORYIO
iWARP	iWARP	User-space	Kernel-space	MEMORYIO
iWARP	iWARP	User-space	Kernel-space	DISKIO
iWARP	iWARP	Kernel-space	Kernel-space	MEMORYIO
iWARP	iWARP	Kernel-space	Kernel-space	DISKIO
Infiniband	Infiniband	User-space	User-space	MEMORYIO
Infiniband	Infiniband	User-space	Kernel-space	MEMORYIO
Infiniband	Infiniband	User-space	Kernel-space	DISKIO
Infiniband	Infiniband	Kernel-space	Kernel-space	MEMORYIO
Infiniband	Infiniband	Kernel-space	Kernel-space	DISKIO

TCP:	Traditional iSCSI without iSER with software TCP.
OSC:	iSCSI with iSER and software OSC iWARP/TCP.
iWARP:	iSCSI with iSER, OFA CMA and hardware iWARP/TCP.
Infiniband:	iSCSI with iSER, OFA CMA and hardware Infiniband.

Table 4.1: Performance Test Combinations

The MEMORYIO mode was chosen in order to demonstrate the throughput of the iSER protocol without the overhead of a disk drive. For this project, the main interest was in demonstrating the benefits of using RDMA hardware to assist an iSCSI connection. The overhead of a disk drive was ignored, for these evaluations, since it is independent of the performance of the iSCSI protocol itself.

While these configurations do not nearly cover the entire set of possible test configurations, they do succeed in demonstrating the benefits of iSER over traditional iSCSI. Using these results, a comparison of the throughput of data transfers in both the read and write directions, with and without RDMA, over a 10GigE network connection was possible.

4.2 Throughput

This section shows the results gathered for the throughput of our implementation in the configurations mentioned above. All of the graphs shown in this section use the following iSCSI parameters:

- MaxRecvDataSegmentLength=512KB
- MaxBurstLength=512KB
- InitiatorRecvDataSegmentLength=512KB (only in iSER-assisted mode)
- TargetRecvDataSegmentLength=512KB (only in iSER-assisted mode)
- InitialR2T=Yes
- ImmediateData=No

The latter two parameter values were chosen in order to prevent the use of immediate and unsolicited data. The current iSER implementation does not properly handle immediate and unsolicited data. These two negotiations were necessary for our iSER-assisted implementation to work properly. The first four parameter values were chosen because

512KB is the largest SCSI data transfer size that is generated by the Linux SCSI mid-level. In the Linux SCSI mid-level, large data transfers are segmented into multiple SCSI Read or Write CDBs that are at most 512KB long. The largest data transfer size possible was chosen in order for the kernel-space iSER-assisted implementation to overcome some of the RDMA memory registration and transfer-setup overhead by performing larger transfers. Using a larger value here would be a waste of resources, since no more than the maximum of 512KB per command will ever be used within kernel-space.

The computers used to perform these evaluations contained four Intel 2.6GHz, 64-bit processor cores with a total of four gigabytes of main memory. These computers were running the Red Hat Enterprise Linux operating system version 5 with version 1.3 of the OpenFabrics Enterprise Distribution. The kernel version of the systems, as reported by the `uname -a` command, was "Linux 2.6.18-8.el5" with Symmetric Multiprocessing support enabled.

4.2.1 Maximum Data Throughput

The maximum throughput of a 10GigE network is 10 Gigabits/sec. This value describes the maximum throughput of the total number of bytes transmitted on the network. In the following graphs, the throughput values, shown along the vertical axis, represent the data throughput carried within TCP/IP and, in the case of the iSER-assisted graphs, iWARP. Each of the protocols in the network stack (see Figure 2-1) use a portion of the available 10 Gigabit/sec bandwidth in order to add extra header information to the data.

Table 4.2 shows the header information added by the various network layers involved in an RDMA tagged buffer transfer. The section labeled *RDMA Data Payload*, which contains 1440 bytes, is where the data is placed in an iSER-assisted iSCSI data transfer. Since the entire Ethernet frame on the wire is 1538 bytes in size, only $\frac{1440}{1538} \approx 93.6\% \Rightarrow$ 9.36 Gigabits/sec of the entire 10 Gigabit/sec bandwidth is available for payload data in an iSER-assisted iSCSI data transfer using the iWARP protocol.

Table 4.3 shows the header information added by the various network layers involved in

Bytes	Layer
12	Ethernet Inter-Frame Gap
8	Ethernet Preamble
14	Ethernet Header
20	IP Header
20	TCP Header
16	Tagged RDMA Header
1440	RDMA Data Payload
4	MPA CRC
4	Ethernet CRC
1538	Total Ethernet Frame

Table 4.2: Header Bytes With iWARP RDMA

Bytes	Layer
12	Ethernet Inter-Frame Gap
8	Ethernet Preamble
14	Ethernet Header
20	IP Header
20	TCP Header
1460	TCP Data Payload
4	Ethernet CRC
1538	Total Ethernet Frame

Table 4.3: Header Bytes With With TCP/IP

a TCP data transfer. The section labeled *TCP Data Payload*, which contains 1460 bytes, is where the data is placed in a traditional iSCSI data transfer. Since the entire Ethernet frame on the wire is 1538 bytes in size, only $\frac{1460}{1538} \approx 94.9\% \Rightarrow 9.49$ Gigabits/sec of the entire 10 Gigabit/sec bandwidth is available for payload data in a traditional iSCSI transfer.

In the following figures, the lines labeled *Theoretical Max RDMA Data Payload Throughput (9360 Megabits/sec)* represents the theoretical maximum data payload throughput available for iSER-assisted iSCSI over the iWARP protocol while using tagged buffer transfers.

4.2.2 Kernel-Space Throughput

In the kernel-space implementation of iSCSI the initiator registers itself with the operating system as a Host Bus Adaptor. Upon the completion of a successful connection to an iSCSI target, a disk device file is created in the `/dev` directory on the initiator that allows for block level access to the target device. The `read` or `write` system calls can be used on this special file in order to cause the operating system to generate a set of SCSI Read or SCSI Write CDBs. These CDBs are then passed to the iSCSI initiator in order to perform the appropriate data transfers to or from the connected target. Once the target finishes the data transfer and sends the initiator an iSCSI response PDU, the iSCSI initiator will inform the operating system and the `read` or `write` system call will complete.

To test the throughput in the kernel-space implementation, two programs, `blastin2` and `blastout2`, were used to read from or write to the iSCSI target device file on the initiator. In the case of `blastin2`, the `read` system call is used to read data from the iSCSI target. Likewise, the `blastout2` program uses the `write` system call to write blocks of data to the iSCSI target. Both of these programs accept two command-line arguments. The first argument specifies an iteration count, the second argument specifies the size in bytes to use for each `read` or `write` system call. The iteration count argument is used in order to perform operations multiple times to get an average throughput value.

In the `blastin2` program the `O_DIRECT` flag is passed to the `open` system call when opening the target device file. This flag is a non-standardized extension added by the Linux

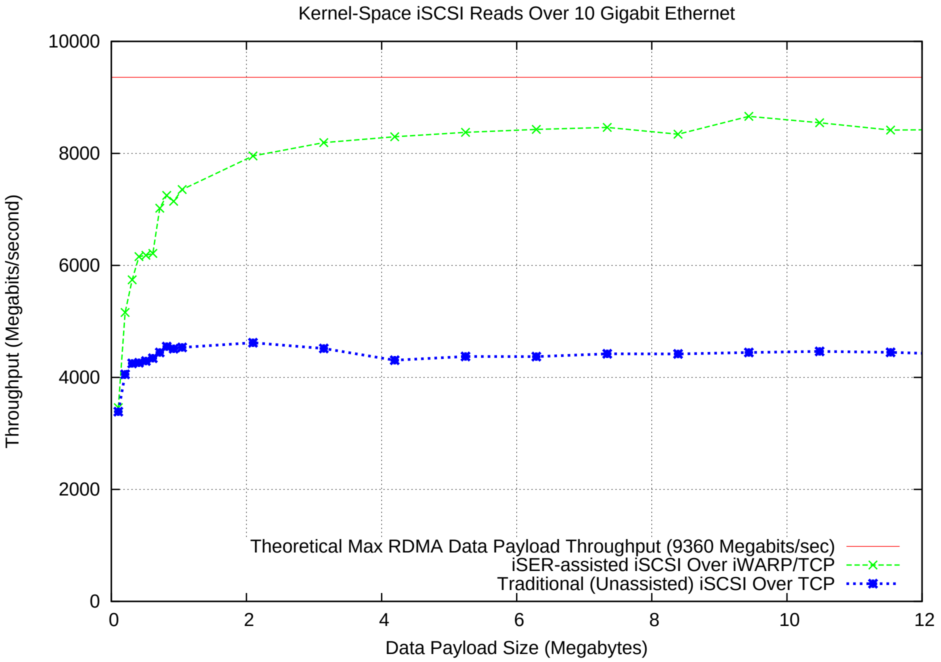

Figure 4-1: Kernel-space iSCSI Read Throughput

kernel that attempts to prevent the operating system from caching accesses to the opened file. Without the O_DIRECT flag the operating system was shown to give impossibly-high performance. This is because after performing the first read command the file system layer in the operating system, assuming that the operation had been performed on a block device, would cache the read data from the target in main memory. Subsequent read calls would then return this data straight from main memory instead of transmitting SCSI Read CDBs to the iSCSI target. With the O_DIRECT flag, however, operations on the target device file are not cached in main memory and a new iSCSI CDB set is generated for each read.

Figure 4-1 shows the throughput of both iSER-assisted iSCSI and traditional iSCSI performing SCSI Reads in kernel-space. The horizontal axis of the graph shown in this figure represents the size of the read system calls performed on the target device file. Each read operation was performed for 10000 or 1000 iterations depending on the size of the operation. For read system calls that were less than 1MB in size, 10000 iterations were used in order to reduce the effects of timing inaccuracies. For `read` system calls that were larger than 1MB in size, 1000 iterations were used since timer accuracy was less of an issue and reducing the iteration count, here, reduced the amount of time required to run the experiment. The throughput values, on the vertical axis, shows the average value over all of the iterations performed for each specific transfer size.

In Figure 4-1 we see that the iSER-assisted iSCSI throughput surpasses the traditional iSCSI implementation almost immediately. The throughput of iSER-assisted iSCSI climbs steeply for transfer sizes below 2MB. This is because, as the transfer sizes increase, the overhead involved in setting up an RDMA transfer becomes less significant than the benefits of offloading and zero-copy transfers. Transfer sizes greater than 2MB show a more steady throughput as the maximum speed of the network is approached. The throughput does fall short of the maximum throughput of a 10GigE network. This extra overhead comes from a few places:

- Ethernet, TCP/IP and RDMA header bytes sent on the wire use a portion of the 10GigE bandwidth. This reduces the amount of bandwidth that is available for data.
- In kernel-space iSCSI Command and Response PDUs that are transmitted for each 512KB of data also use a portion of the bandwidth. Again, this reduces the amount of bandwidth available for TCP/IP data.
- Processing of each iSCSI Command and Response PDU on the target and initiator increases the latency of the transfer and therefore also reduces the throughput.

Figure 4-2 shows the throughput of both iSER-assisted iSCSI and traditional iSCSI performing SCSI Writes in kernel-space. The horizontal axis of the graph shown in this figure represents the size of the `write` system calls performed on the target device file. Each `write` call was performed for 10000 or 1000 iterations with the same procedure as used for the read operations. The vertical axis of this graph shows the average throughput over all of the iterations performed for each specific transfer size.

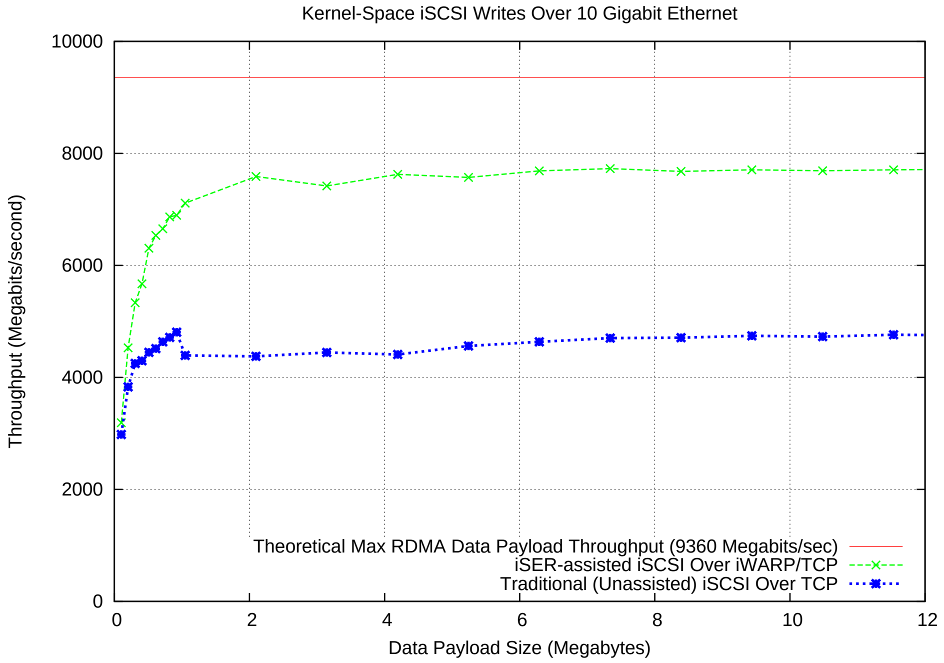

Figure 4-2: Kernel-space iSCSI Write Throughput

In Figure 4-2 we see that the iSER-assisted iSCSI throughput, again, surpasses traditional iSCSI immediately. In this figure, we also see a steep climb in throughput until about 2MB. This steep climb in throughput happens as the overhead of setting up an RDMA transfer becomes less significant with respect to the size of the data transfer. For transfer sizes greater than 2MB the throughput flattens out. In this graph, iSER-assisted iSCSI falls even further short of the maximum throughput of the 10GigE connection. This is because, in addition to suffering from all of the sources of extra overhead that were mentioned above, iSER-assisted Writes involve an extra RDMA ReadRequest message to be sent from the target to the initiator in order to begin the data transfer. This extra exchange increases the latency of SCSI/iSCSI Write commands and reduces the data throughput, as shown in this graph.

Figure 4-3 shows the iSER-assisted iSCSI Read and Write throughput values plotted on the same graph. This figure shows that for transfers that are greater than about one

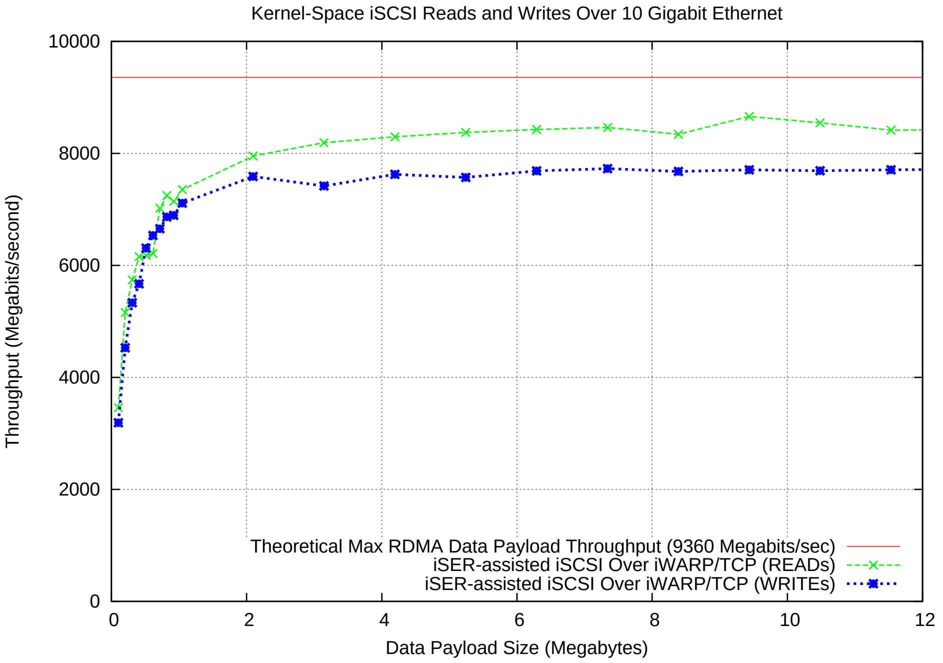

Figure 4-3: Kernel-space iSER Read and Write Throughputs

megabyte in size the performance for Read operations is about 500 Megabits/sec greater than that of Write operations. The reason that iSCSI/SCSI Read operations perform better than iSCSI/SCSI Write operations, as described earlier, is because of the extra RDMA ReadRequest message that is required for the RDMA Read operations that implement iSCSI/SCSI Write data transfers.

4.2.3 User-Space Throughput

Since the user-space implementation does not register itself with the operating system, we are unable to make use of the `blastin2` and `blastout2` programs used for the kernel-space evaluations. Instead, scripts were written for the user-space initiator's interpreter that would exhibit behavior similar to that of the blast programs. The graphs shown next were generated using these blast-like scripts in user-space.

Figure 4-4 shows the throughput of user-space Read operations. The same iteration

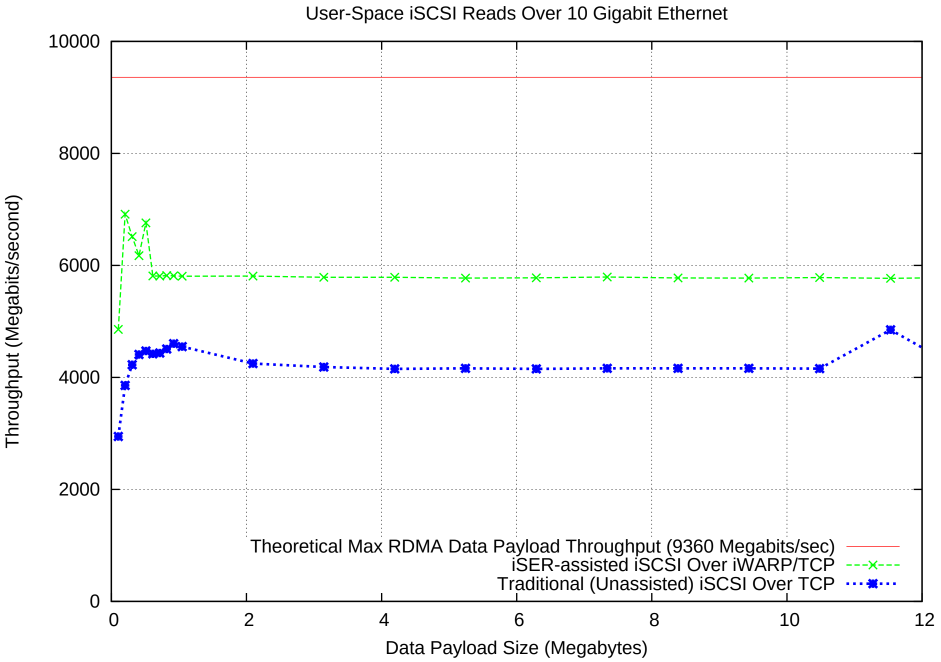

Figure 4-4: User-space iSCSI Read Throughput

scheme was used in order to generate this graph as was used for the kernel-space graphs. These iteration counts were implemented using the `repeat` command in the iSCSI initiator's command interpreter. We are unable to explain why the throughput for user-space iSER-assisted Read operations flattens out at just over 5500 Megabits/sec. In user-space there is less processing being performed on each Read operation since there is no interaction involved with the Linux kernel SCSI mid-level. It is because of this that we would expect the user-space Read operations to have a slightly greater throughput than kernel-space.

Figure 4-5 shows the throughput of user-space Write operations. Again, the standard iteration scheme was performed by using the `repeat` command in the iSCSI initiator's command interpreter. This graph shows that the user-space iSER-assisted implementation is able to achieve a much higher throughput compared to the traditional iSCSI implementation. In this figure, we see that the throughput of user-space, iSER-assisted iSCSI Writes ramps up very steeply for transfer sizes of under 1MB. After about 1MB the throughput

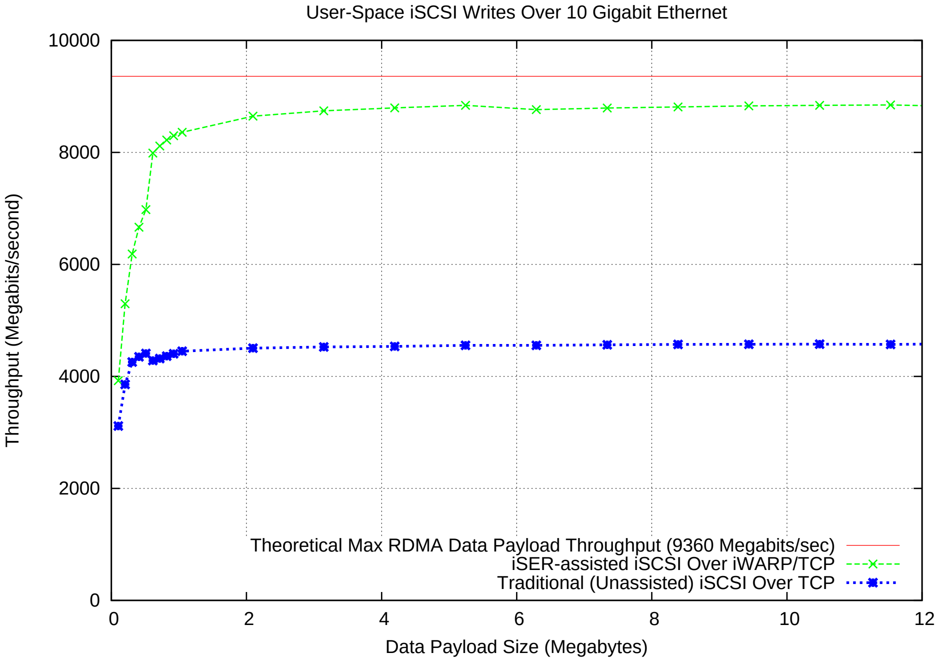

Figure 4-5: User-space iSCSI Write Throughput

flattens out at just over 8500 Megabits/sec. These operations, again, do not reach the maximum throughput of the 10GigE network.

Figure 4-6 shows the user-space Read and Write throughput values plotted on the same graph. As stated earlier, we were unable to determine why the iSER-assisted Read operations do not give higher throughput values.

4.2.4 User-Space and Kernel-Space

The comparisons shown in this section are between the user-space and kernel-space iSER-assisted iSCSI implementations. Figure 4-7 shows the throughput values for iSER-assisted iSCSI Read operations and Figure 4-8 shows the throughput values for iSER-assisted iSCSI Write operations. Although we were unable to explain the behavior of the user-space Read operations, the results of the SCSI Write operations in Figure 4-8 shows that the implementation is able to achieve a greater throughput in user-space than in kernel-space.

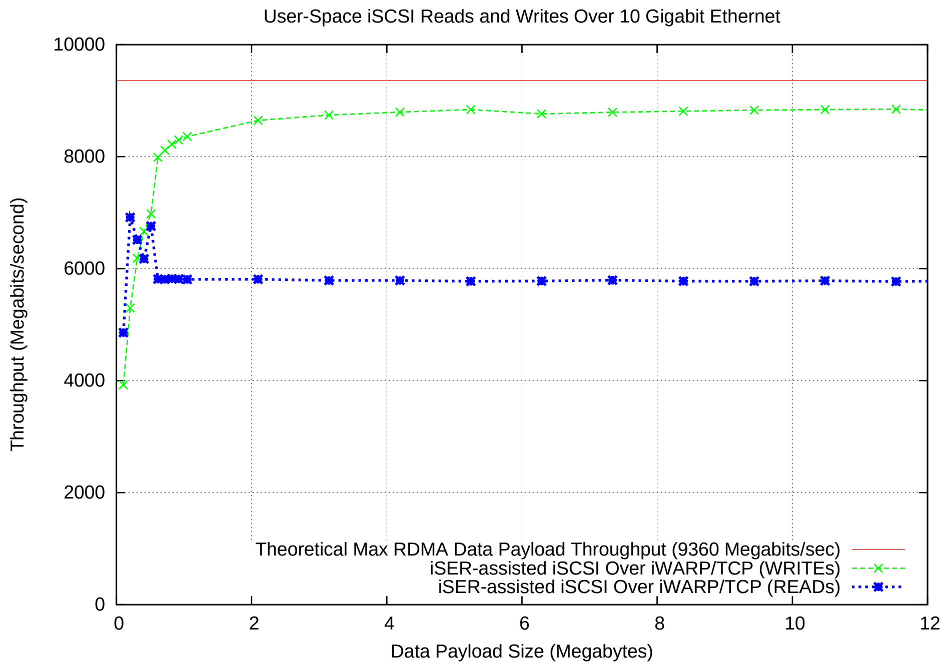

Figure 4-6: User-space iSER Read and Write Throughputs

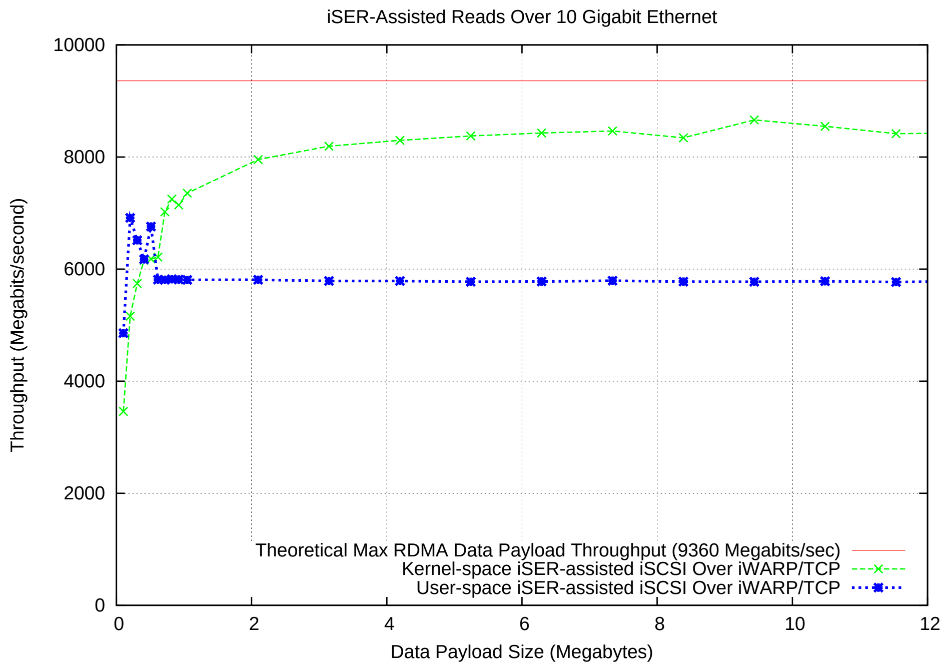

Figure 4-7: iSER Read Operation Throughput

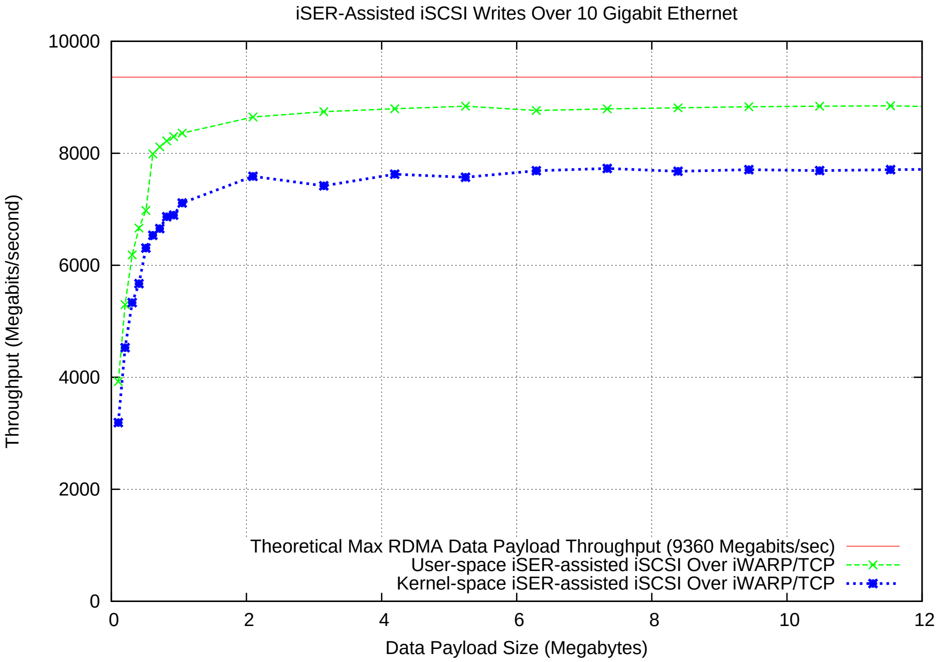

Figure 4-8: iSER Write Operation Throughput

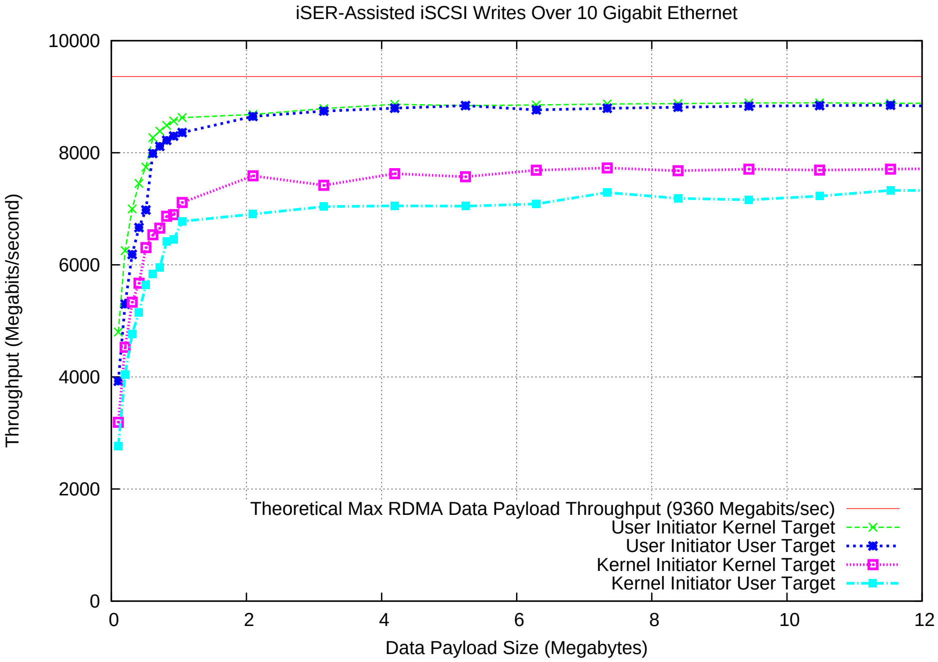

Figure 4-9: Hybrid User/Kernel-Space iSER Write Operation Throughput

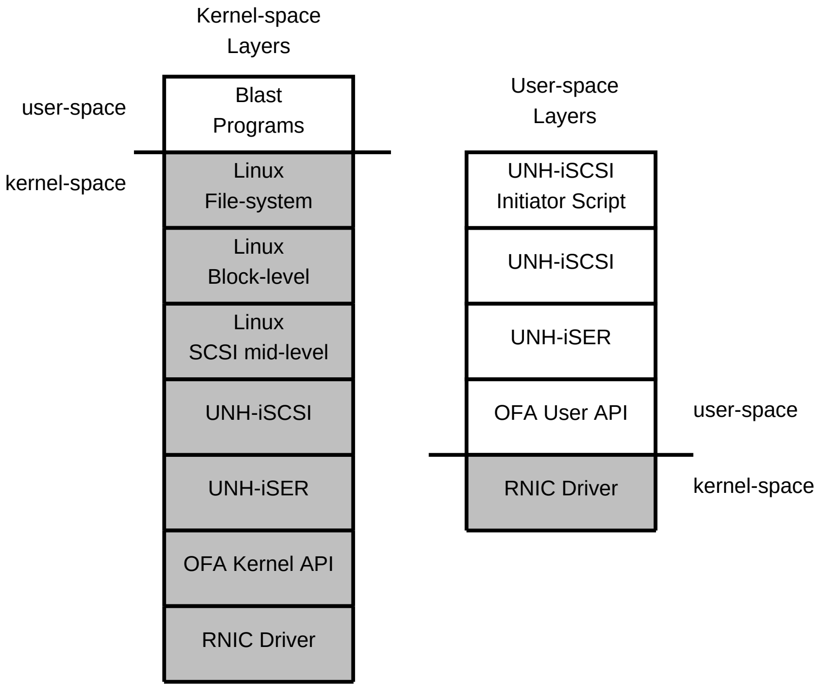

Figure 4-10: Kernel-space and User-space iSER-Assisted Initiator Layering

In order to explore this phenomenon, evaluations were performed on the SCSI Write throughput in a full set of combinations between the user-space and kernel-space, target and initiator. These results are shown in Figure 4-9. This figure shows that the most limiting agent is the kernel-space iSCSI initiator. Both of the runs in which the kernel-space iSCSI initiator was used show significantly lower throughput values than the runs in which the user-space initiator was used.

The reason that the user-space initiator performs significantly better than the kernel-space initiator is because it does not use multiple SCSI commands for each data transfer. Figure 4-10 shows the layering of the stages that data transfer operations pass through on their way through the kernel-space and user-space iSER-assisted iSCSI initiator. In the kernel-space implementation, the `read` and `write` system calls are passed down from the Linux file-system layer and eventually make their way into the Linux SCSI mid-level. In the SCSI mid-level, Linux builds a set of SCSI CDBs that it queues up with the UNH-iSCSI initiator to perform the requested data transfer operations. The maximum size SCSI CDB that the Linux SCSI mid-level will generate is 512KB. This means that, for each megabyte transferred, the initiator will generate and transfer two SCSI CDBs, two iSCSI Command PDUs (one for each CDB), while the target performs two transfer operations and sends two iSCSI Response PDUs. It is also important to note that the iSCSI initiator only sends a single command at a time to the iSCSI target, so none of these commands are being handled in parallel[1]. In user-space, the UNH-iSCSI initiator script performs data transfers by building SCSI CDBs and queuing them with the UNH-iSCSI initiator. The CDBs generated by the UNH-iSCSI initiator script, in these experiments, are all the exact size of the requested transfer. This means that for each data transfer, regardless of its size, there is only a single SCSI CDB, iSCSI Command PDU and iSCSI Response PDU.

Figure 4-11 shows both the kernel-space and the user-space implementations performing a 1MB Write operations. An important thing to remember is that, even for larger transfers, the user-space implementation will never use more than a single SCSI CDB to perform the transfers. In the kernel-space implementation, a new SCSI CDB will be transferred for each 512KB of data. This means that, for a 12MB transfer, the kernel-space implementation will transmit 24 SCSI CDBs, and the user-space implementation will only transmit one.

Figure 4-12 shows the throughput of kernel-space iSER-assisted iSCSI Write commands and of user-space iSER-assisted iSCSI Write commands which are restricted to 512KB in size. In order to build the user-space portion of this graph, data transfers using sets of SCSI Write CDBs of size 512KB were generated using the initiator's command interpreter. This comparison demonstrates the difference in throughput between user-space and kernel-space while both implementations use the same maximum SCSI CDB size. This comparison shows that the kernel-space implementation achieves a greater maximum throughput than the user space implementation does. This is, most likely, because the kernel-space does not suffer

[1] UNH-iSCSI allows this to be changed by reconfiguration, however, at this time the multiple outstanding command feature does not appear to work properly with iSER enabled.

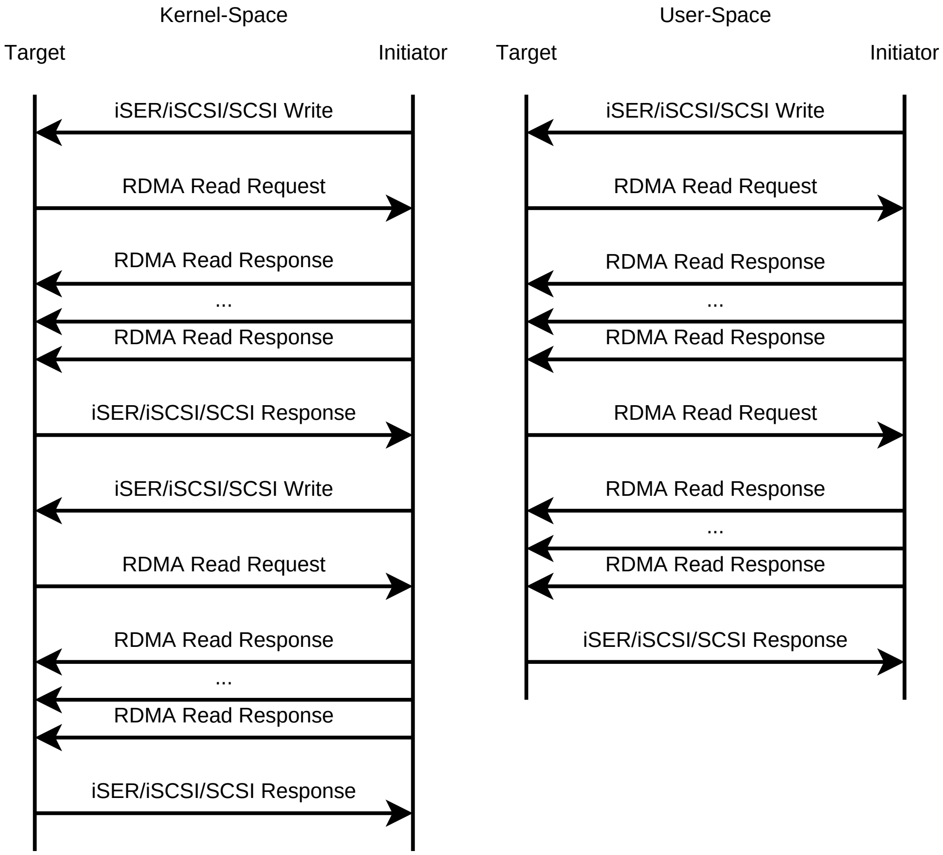

Figure 4-11: Kernel-space and User-space 1MB Write Operation

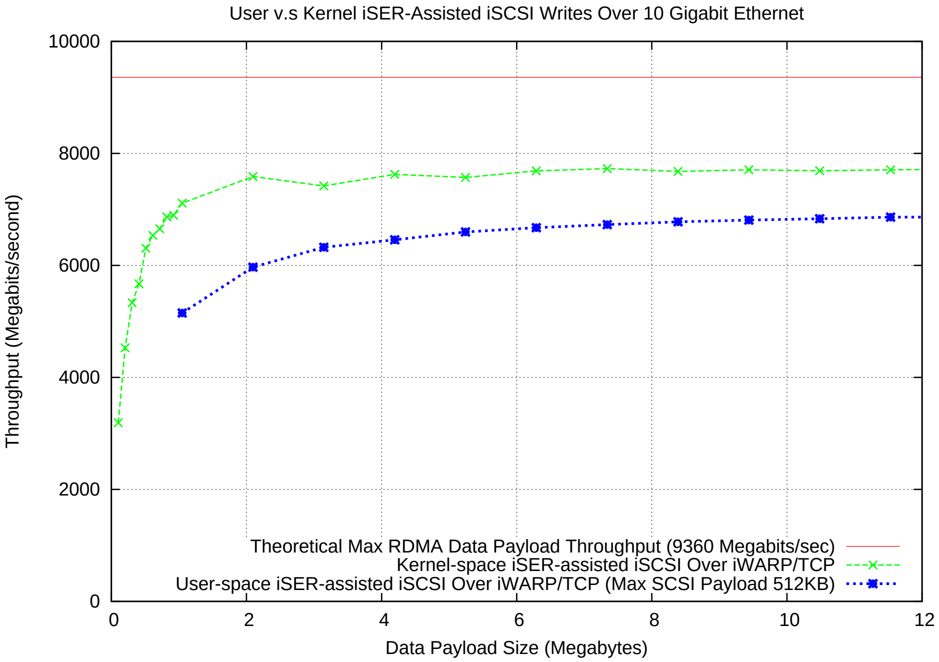

Figure 4-12: Hybrid User/Kernel-Space iSER Write Operation Throughput

from context switch overhead and the extra overhead involved in the user-space/kernel-space API abstraction.

Figure 4-13 shows the difference in performance between the user-space implementation when using exact size SCSI Write commands and when using SCSI Write commands no larger than 512KB. This figure shows that, without the extra overhead of building, registering, transmitting and processing the extra SCSI CDBs, the user-space implementation is able to achieve a significant performance increase.

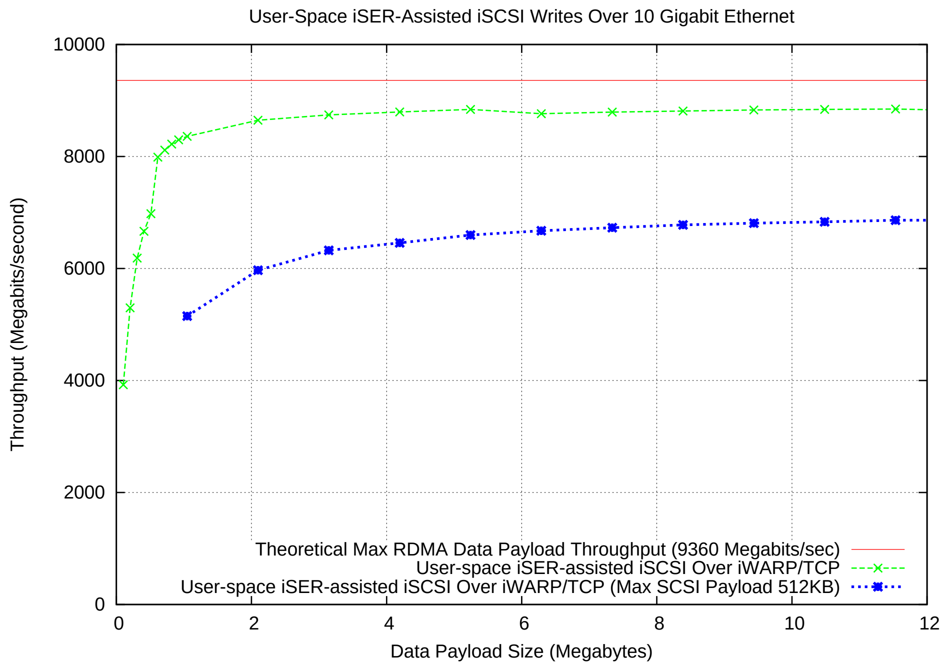

Figure 4-13: Hybrid User-Space iSER Write Operation Throughput

CHAPTER 5

CONCLUSIONS

5.1 Conclusions

The goal of this thesis was to implement an iSER-assisted iSCSI solution and to evaluate its performance. In order to accomplish this goal, support was added to the UNH-iSCSI reference implementation for the iSER extensions to the iSCSI protocol. This paper has described the approach which was used to perform this task and some of the difficulties that were encountered on the way. Additionally, the results of a set of throughput evaluations using this new implementation were presented. Table 5.1 shows the maximum throughput values in Megabits/sec that were achieved by UNH-iSCSI in iSER-assisted mode and in traditional iSCSI mode. These results show that the iSER extensions enable an iSCSI solution to make much more efficient use of a 10GigE network.

	iSER Throughput (Megabits/sec)	iSCSI Throughput (Megabits/sec)
Max Data Payload Throughput	9360	9490
User-Space iSCSI Writes	8847 (94.5%)	4580 (48.3%)
Kernel-Space iSCSI Reads	8661 (92.5%)	4463 (47.0%)
Kernel-Space iSCSI Writes	7727 (82.6%)	4761 (50.2%)

Table 5.1: Throughput Performance Results

5.2 Future Work

Due to time constraints imposed by the unexpected difficulties in getting our implementation to work properly, we were unable to perform all of the possible evaluations that are enabled from our iSCSI solution's design. Table 4.1 shown in the previous chapter presents all of the possible configurations that our iSCSI implementation is designed to support. It would be beneficial for a future project to evaluate the performance of all of these different configurations. Specifically, it would be interesting to see a comparison between our iSER-assisted implementation running over iWARP and Infiniband. Currently, Infiniband is the leading RDMA technology, and it would be useful to see how the newer iWARP protocol performs with respect to Infiniband. It would also be useful to do more work with the UNH-iSCSI target's DISKIO mode in order to evaluate the performance of our solution when it is interacting with a real disk device. This configuration would show the benefits of the real-world usage of our solution. Finally, a comparison between iSER-assisted iSCSI using RDMA offload hardware and a fully offloaded iSCSI solution would be beneficial. If iSER-assisted iSCSI is able to achieve similar speeds to fully offloaded iSCSI, then general purpose RDMA hardware can be used in practice instead of very special purpose iSCSI offload hardware.

In addition to the comparisons mentioned above, a future project could explore the usage of immediate and unsolicited data with iSER-assisted iSCSI. Since RDMA transfers require a significant amount of overhead to setup, it may prove beneficial to make use of iSCSI immediate and unsolicited data transfers to circumvent the RDMA setup overhead for small transfers. Our current implementation does not properly handle immediate and unsolicited data in iSER-assisted mode. A future project could add support for this and then attempt to find an optimal immediate/unsolicited data size to use in order to get the greatest performance on an iSER-assisted connection.

The OpenFabrics Alliance API is planning on adding support for the RDMA Send with Solicited Event and Invalidate operation (SendSEInv) in an upcoming release of the Linux kernel. A future project may want to explore the usage of the SendSEInv operation with iSCSI Response PDUs. Theoretically the SendSEInv operation will allow for re-using some of the memory registration information on the RDMA hardware, which will reduce RDMA transfer setup overhead. It would be interesting to see how much of a reduction in overhead occurs when using the SendSEInv operation as specified in the iSER standard.

Another benefit of Remote Direct Memory Access hardware is that network protocol offloading, should greatly reduce the amount of work that is needed by a CPU. Future work should attempt to measure the CPU load of iSER-assisted iSCSI in various situations and then compare it with the CPU utilization of traditional iSCSI. This evaluation will be useful to demonstrate one of the additional benefits of the iSER extensions to the iSCSI protocol.

BIBLIOGRAPHY

[1] Open Fabrics Alliance. http://www.openfabrics.org.

[2] Infiniband Trade Association. Infiniband Architecture Specification version 1.2.1.

[3] M. Chadalapaka, J. Hufferd, J. Satran, and H. Shah. Datamover Architecture for Internet Small Computer System Interface (iSCSI). RFC 5047 (Informational), October 2007.

[4] P. Culley, U. Elzur, R. Recio, S. Bailey, and J. Carrier. Marker pdu aligned framing for tcp specification. RFC 5044 (Standards Track), October 2007.

[5] DAT-Collaborative. *kDAPL: Kernel Direct Access Programming Library*, June 2002.

[6] D. Delessandro, A. Devulapalli, and P. Wyckoff. Design and Implementation of the iWarp Protocol in Software. PDCS, November 2005.

[7] J. L. Hufferd. *iSCSI The Universal Storage Connection*. Addison Wesley, 2003.

[8] M. Ko, M. Chadalapaka, J. Hufferd, U. Elzur, H. Shah, and P. Thaler. Internet Small Computer System Interface (iSCSI) Extensions for Remote Direct Memory Access (RDMA). RFC 5046 (Standards Track), October 2007.

[9] A. Palekar, A. Chadda, N. Ganapathy, and R. Russell. Design and implementation of a software prototype for storage area network protocol evaluation. In *Proceedings of the 13th International Conference on Parallel and Distributed Computing and Systems*, pages 21–24, August 2001.

[10] M. Patel and R. Russell. Design and Implementation of iSCSI Extensions for RDMA, September 2005.

[11] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A remote direct memory access protocol specification. RFC 5040 (Standards Track), October 2007.

[12] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet Small Computer Systems Interface (iSCSI). RFC 3720 (Proposed Standard), April 2004. Updated by RFCs 3980, 4850.

[13] H. Shah, J. Pinkerton, R. Recio, and P. Culley. Direct data placement over reliable transports. RFC 5041 (Standards Track), October 2007.

[14] Yamini Shastry, Steve Klotz, and Robert D. Russell. Evaluating the Effect of iSCSI Protocol Parameters on Performance.

[15] Voltaire. Open source iSER code at https://svn.openfabrics.org/svn/openib/gen2/.

[16] F. Xu and R.D. Russell. An architecture for public internet disks. In *Proceedings of the 3rd International Workshop on Storage Network Architecture and Parallel I/O (SNAPI05)*, pages 73–80, September 2005.