Long-range RMDA over PCI Express

Henrik Nårstad
Master’s Thesis Autumn 2015
Long-range RMDA over PCI Express

Henrik Nårstad
Abstract

Many computing related tasks today require a lot of hardware infrastructure to fulfill requirements and expectations of its users. Physical infrastructure used to serve systems are often organized in several geographically separate computer clusters.

In this thesis, we have investigated and developed a working prototype which enables nodes in a PCI Express based computer cluster to connect with, and transfer data a node in remote PCI Express based cluster. Central to our design is the cluster gateway, or proxy node. Each cluster consists of endpoint nodes and one proxy node. The proxy acts as a gateway for incoming and outgoing data traffic to and from nodes in the local cluster. Every data transfer is relayed via the proxies which carries the responsibility of forwarding outgoing data to a destination remote cluster, and forwarding incoming data to the recipient node.

The system is implemented on PCI Express based clusters using Ethernet as the medium connecting remote clusters together. The cluster interconnect technology enables nodes to connect to memory segments in a node within the cluster and perform read and write operations on it using either programmed I/O or Remote Direct Memory Access. We have implemented functionality intended to supplement an already existing API, that can be used to execute inter-cluster data transmissions.
Long-range RDMA over PCIe
3.3.2 Transaction ring ........................................... 28
3.3.3 Control Segment ........................................ 29
3.3.4 Connection establishment ................................ 29
3.3.5 Proxy to proxy communication .......................... 30
3.4 Summary ...................................................... 32

4 Implementation ........................................... 33
  4.1 Slab Allocator ............................................ 33
  4.2 Addressing ................................................ 34
  4.3 Distributed mutexes ...................................... 34
  4.4 Proxy to proxy communication ................................ 36
  4.5 Optimizations and Improvements – Endpoint .................. 38
    4.5.1 Reliability .......................................... 38
    4.5.2 Interrupts .......................................... 39
    4.5.3 Vectored DMA ....................................... 40
    4.5.4 Smarter TX-ring interrupt ............................ 40
    4.5.5 Reducing mutex usage ................................ 41
    4.5.6 Communication channel protocol optimization .......... 42
    4.5.7 Control channel optimizations ........................ 43
    4.5.8 Endpoint transfer optimizations ........................ 45
    4.5.9 Optimistic transfers .................................. 45
    4.5.10 Programmed I/O for small transfers .................. 45
    4.5.11 Parallelization ....................................... 46
    4.5.12 Transparent handling of receiving data ................. 47
  4.6 Optimizations and Improvements – Proxy .................... 48
    4.6.1 Multiple remote cluster connections .................... 48
    4.6.2 Multiple local endpoints ................................ 48
    4.6.3 Buffer segment optimizations .......................... 49
    4.6.4 Slab allocation ....................................... 50
    4.6.5 Added support for connecting to individual segments .... 50
    4.6.6 Proxy to proxy channel optimizations ................ 51
  4.7 API for long-range RDMA .................................. 51
    4.7.1 Data structures ....................................... 51
    4.7.2 Functions ............................................. 52
  4.8 Summary ...................................................... 54

5 Evaluation and Results .................................. 55
  5.1 Testing environment ....................................... 55
    5.1.1 Setup ............................................... 55
    5.1.2 Tests ............................................... 56
  5.2 Bandwidth benchmarks ..................................... 57
  5.3 Latency benchmarks ....................................... 66
  5.4 Evaluation .................................................. 69
  5.5 Summary ...................................................... 71
6 Conclusion

6.1 Summary .................................................. 73
6.2 Main Contributions ........................................ 74
6.3 Future work .................................................. 74
Long-range RDMA over PCIe
List of Figures

1.1 A basic overview of two connected computer clusters .................................. 5
2.1 PCI Express packet format ................................................................................. 8
2.2 PCI Express protocol stack ................................................................................. 8
2.3 Data Link Layer Packet structure ................................................................. 10
2.4 Dolphin IXH610 Host Adapter ........................................................................... 14
2.5 Dolphin IXS600 PCI Express switch ............................................................... 15
2.6 Dolphin Software stack ...................................................................................... 16
2.7 A state diagram showing the different states of a SISCI segment [1] ............... 17
2.8 A state diagram showing the different states of a SISCI DMA Queue .......... 18
2.9 SISCI DMA latency and bandwidth ............................................................... 19
3.1 An overview illustrating that all inter-cluster communication happens through a proxy ............................................................................................................. 22
3.2 Control packet layout ....................................................................................... 24
3.3 An example illustrating the message exchanges on the control channel in a successful transmission ................................................................. 25
3.4 An illustration of the proxy buffer segment .................................................. 27
3.5 Each element in the ring buffer segment refers to a buffer .......................... 28
3.6 Control segment ............................................................................................. 29
3.7 Initial proxy-buffer layout ................................................................................. 31
3.8 Finalized proxy-buffer layout .......................................................................... 32
4.1 Slab header ................................................................................................... 34
4.2 Slab operation .................................................................................................. 35
4.3 An example showing the relative address (bottom) to the absolute address (top) .... 36
4.4 Locking and unlocking of distributed mutexes ............................................. 36
4.5 Locking and unlocking of distributed mutexes ............................................. 37
4.6 Pseudocode showing the process of transmitting proxy buffers ............. 38
4.7 Pseudo code showing how the proxy handles incoming packets ............. 39
4.8 Transaction ring split into two separate rings .............................................. 42
4.9 An example illustrating the message exchanges on the control channel in a successful transmission after optimizing the control channel protocol. The gray arrows are the messages that was stripped away .................................. 44
4.10 The buffer cache layout after being processed by the proxy and with the added segment ID field ................................................................. 47
5.1 The testing setup used in the benchmark tests. The example shows two virtual clusters which a proxy and an endpoint, spread across two machines. The arrows illustrate the transmission steps in a transfer from the endpoint in cluster 1, to the endpoint in cluster 2. 56
5.2 Endpoint to Proxy DMA bandwidth benchmark 58
5.3 Endpoint to proxy Vectored DMA bandwidth benchmark 60
5.4 Endpoint to proxy PIO bandwidth benchmark 61
5.5 Proxy to proxy PIO bandwidth benchmark 62
5.6 Endpoint to endpoint DMA bandwidth benchmark 63
5.7 Endpoint to endpoint vectored DMA bandwidth benchmark 64
5.8 Endpoint to endpoint PIO bandwidth benchmark 65
5.9 Endpoint to endpoint DMA latency benchmark 67
5.10 Endpoint to endpoint vectored DMA latency benchmark 68
5.11 Endpoint to endpoint PIO latency benchmark 69
List of Tables

2.1 Transaction Layer Packet types .............................................. 9
5.1 Specifications of the machines used in the testing environment .......... 56
Acknowledgements

I would like to express my gratitude to my supervisors Håkon Kvale Stensland, Carsten Griwodz and Andreas Petlund for their invaluable help, feedback and support. I would also like to thank Roy Nordstrøm and Hugo Kohmann at Dolphin Interconnect Solutions for the support, insightful comments and feedback.

Finally I would like to thank my friends and family for supporting me throughout writing this thesis and my life in general.
Long-range RDMA over PCIe
Chapter 1

Introduction

1.1 Background

Today there is an ever increasing need for computing power and systems demanding high availability. For large and complex systems providing users with services, such as cloud based storage, mail services or social media platforms, users expect the service to be accessible at all times, from their laptop, smartphone, tablet or other device. Such systems typically requires a lot of infrastructure for keeping the availability high. Tasks such as computer graphics rendering or weather prediction algorithms typically requires huge amounts of computing power. Both high availability and computing intensive tasks are often carried out, not on single system machines, but on many cooperating computers commonly referred to as computer clusters.

Computer clusters are a popular alternative to traditional architectures. A computer cluster is formed when multiple individual computers, referred to as nodes, are interconnected to each other via some medium that enables each node to communicate and exchange data with each other, to form a single computer system. An important advantage of computer clusters over traditional architectures are that they are easily scalable; adding or removing nodes as the need may arise is simple with the interconnect mediums such as Infiniband [2], Ethernet or an PCI Express based interconnect. Computers as we know them are always subject to failure, and should an individual node within a cluster fail other nodes can acquire its workload until the failing node is replaced. The manner in which work is redistributed depends on the cluster scheduler in place. While this is still an area of research many different cluster scheduler designs and architectures exists, such as Mesos [3], Apache Hadoop On Demand [4] and Omega which is used in Google’s next-generation cluster management system [5].

For services that provide a lot of data to a lot of different users, such as Youtube, Google’s search engine or Dropbox, it is a challenge to continuously scale their infrastructure to handle the requirements that comes with a great number of users. This is where content delivery systems (CDNs) offers a solution. CDNs are large distributed systems used to deliver content with improved experience for the end user (high throughput, and low latency) between the content delivery network and the end user. A Content delivery network consists of one or more data centers each of which hosts a multiple of nodes. These data centers can be geographically diverse, sometimes in different cities. Geo-replication is the process of replicating content across geographically diverse data centers. Replication across data centers is often done in order to increase the availability. Googles e-mail service replicates across five data centers to tolerate one planned outage, and one planned [6]. Replicating data internally in a data center
and replicating data across geographically separate data centers is different because of the extra delay caused by the distance between them. For that reason, there are systems designed with this in mind. Paxos is an algorithm for reaching consensus among a group of replicas and was first described in [7]. Many systems is based on this algorithm, such as Megastore which has been deployed within Google for several years [8]. Megastore uses a modified version of Paxos, optimized for fast reads and fast writes, to replicate primary user data across data centers [9]. Another system using Paxos is Multi-data center consistency [6].

The usage scenarios of a computer cluster can be many. Clusters are commonly used in many super computers are distributed computing, where the interconnect medium connects computing nodes to form one super computer. The worlds most powerful super computer today, in terms of Floating-point Operations Per Second (FLOPS), is the Tianhe-2 featuring 3120000 cores and 1024 Terabytes of memory [10]. Another example is distributed storage systems such as Cassandra [11] initially developed by Facebook, Apache HBase [12], or Bigtable [13] developed by Google.

With these large distributed systems (like databases, file systems, shared memory, etc.), the workload caused by a large amount of requests can be spread across machines within the cluster. By spreading the traffic across many nodes, the system appears as one, but is in reality a number of nodes cooperating and sharing the overall load. This has the advantage of being very scalable and fail-proof; if one node fails, one or more functional nodes can acquire acquire the workload of the failing computer, thus increasing the availability of the system. As the requirements for a system grows more nodes can be dynamically added to the network so that the workload can be distributed across more nodes. Some examples of content delivery network providers are Akamai [14], Amazon CloudFront [15], Fastly [16] and Microsoft Azure CDN [17].

The process of transferring or replicating data across nodes can be simplified by using Direct Memory Access transfers. This is already supported in some hardware using Peripheral Component Interconnect Express (PCI Express). However, there are no solutions, to our knowledge, that can perform PCI Express Direct Memory Access (DMA) over distances longer than a few meters. In this thesis we discuss the viability of connecting PCI Express based clusters over longer distances than what current technologies support, so that the process of replicating data across distances can be simplified by using Direct Memory Access transfers.

We have implemented a prototype solution that connects two PCI Express clusters over Ethernet. Our implementation enables any two nodes in geographically diverse clusters to perform DMA for transferring or replicating data.

### 1.2 Problem Definition

The process of transferring data between two machines can be done with relative ease by utilizing a computer network, a computer with a network interface card (NIC), the OSI network model, a programming language and a socket API. Even though dealing with network sockets does not require much hassle, it still has some disadvantages over other lower level data data transfer mechanisms such as remote DMA. One of the biggest disadvantages is the fact that it requires more attention from the programmer; he, or she, has to design a protocol to make sure the data arrives correctly at the intended location on the receiving end, there are considerations regarding performance that has to be taken into account, etc. In this thesis we will explore the possibility of making this process simpler, from a programmers perspective, by enabling the
machines to transfer data with Direct Memory Accesses. Even between machines which are not geographically close. Copying data using RDMA introduces several advantages over traditional copying mechanisms. Most importantly is the fact that the copying itself is not handled by the Central Processing Unit (CPU). Instead it is handled by a separate dedicated DMA engine. Having the DMA engine handle the copying leaves the CPU to do other tasks while the copy is taking place, which in the end, may lead to a more efficient system. DMA operations are also very simple by nature, so there is also potential for improving bandwidth and latency performance.

We have used PCI Express equipment produced by a company called Dolphin ICS which enables interconnected machines within a local cluster to perform DMA transfers. The goal for this thesis is to implement a prototype system, using the PCI Express equipment, which is able to perform DMA transactions between connected PCI Express clusters over a long distance. By long distances, in this case, we are talking about distances up to several hundred kilometers, or the distance between neighboring cities.

We will first design and implement a prototype for performing DMA transfers over PCI Express using Ethernet and the OSI model as the medium connecting the remote machines. Next, we discuss the drawbacks and advantages with the design pattern we used, as well as alternative solutions. We also explore and discuss the measures we took in order to reach the performance of our last version of our prototype.

1.3 Limitations

While parts of this system could be implemented at a hardware and/or kernel level for a boost in performance, we will limit ourselves to a software implementation in user space. The focus is primarily on the design and implementation of an efficient method of receiving and transmitting data from one end node to a remote end node. Possibilities for further optimizations and improvements is presented in section 6.3. To be able to complete this thesis in a timely fashion we have also limited us to terminating the DMA transfers at the cluster gateway node (see chapter 3), thus ending the PCI Express session before sending data over Ethernet to the remote cluster. If a PCI Express session were to exist between remote clusters the PCI Express protocol would have to be extensively modified in order to cope with the extra challenges caused by the increased distances.
1.4 Research method

In this thesis we have have designed and developed a prototype for connecting computer clusters over Ethernet and performing DMA transfers. We started by specifying the requirements and specifications before implementing a first version of our prototype. After the implementation step, we performed tests in order to verify that the system performed in accordance to the initial requirements and specifications. This research method corresponds to the design paradigm specified in Computing as a Discipline by the ACM Task Force. [18]

1.5 Main Contributions

In this thesis we have presented and demonstrated the performance of a prototype for long distance DMA transactions and that it can be done with relatively good results. Several details of the design and implementation of our prototype is included. Also included is a presentation of the technology we used.

We first designed and implemented a protocol for transferring data between two connected clusters. In each cluster there is a common gateway, or proxy; proxies communicate with other cluster-proxies and must follow this protocol in order to correctly transfer data. This is discussed in chapters 3 and 5.

Next, we designed and implemented an end-to-end protocol that works on a separate control channel. An endpoint machine that wishes to transfer data to another endpoint in a remote cluster must follow this protocol. This is discussed in chapters 3 and 5.

After we had tested and verified that we had a working implementation of the cluster gateway and the endpoint, we started the process of identifying where the bottlenecks were and how to minimize them so that better performance could be achieved.

The result of this thesis is a working prototype of a system, capable of performing inter-cluster data transfers using proposed extensions to an existing API used for local cluster programming.

1.6 Outline

In chapter 2, we introduce PCI Express technology and the layer in its protocol stack. This chapter also includes an introduction to Dolphin Interconnect Solutions, the SISCI developers kit [19] and the hardware we used in our system. In chapter 3, we explain and discuss the various design aspects of our prototype. In chapter 4, we present and discuss implementation details as well as optimization steps. In chapter 5, we evaluate the results of our tests, different designs or solutions, possible improvements to further increase performance. In chapter 6, we give a summary and conclusion of the thesis.
Chapter 2

PCI Express

In the last chapter we gave a short introduction to the thesis along with the problem statement. In this chapter we will give an introduction to the main technology used in this thesis, namely Peripheral Component Interconnect Express (PCI Express). PCI Express is a standard specifying details from low level hardware up to flow control mechanisms. It defines a protocol stack which consists of three layers. The first section introduces some higher level details of the standard while the next three sections gives an introduction to the individual protocol stack layers. In the next section we give an introduction to Dolphin and their PCI Express based solutions which is essential to the thesis. Dolphin provides both PCI Express based hardware and software. Finally we introduce the SISCI API, which is a low level API used to program and control Dolphins PCI Express based hardware.

2.1 PCI Express protocol

PCI Express is a high performance serial I/O interconnect technology which is meant to replace the older parallel bus standards, PCI and PCI-X. PCI Express has many improvements over its predecessors, among which bandwidth is the most important. Almost all modern desktop computers today comes built-in with PCI Express. A traditional usage scenario for PCI Express is to connect secondary devices, such as sound, video and network cards, to the CPU and memory within one machine. However, there exists PCI Express technologies which is used to interconnect two or more machines into a network of computers, this is known as a cluster. This technology allows for computers within a local cluster to share their memory, typically to improve computing performance.

PCI Express is based on PCI and PCI-X, but have moved away from the parallel bus model of its predecessors. Instead of a parallel bus, PCI Express implements a serial bus model but still remains backwards compatible with PCI in software, in the sense that PCI systems can detect and configure PCI Express devices without explicitly supporting PCI Express.

PCI Express devices communicate with each other by sending packets over a path. This path is called a link and consists of one or more lanes. A link can be made up of 1, 2, 4, 8, 12, 16 or 32 lanes [20]. The number of lanes a link consists of is commonly referred to as the link width. The larger the link width is the greater the bandwidth becomes, but power consumption and cost also is increased. Each lane consists of a receive and transmit pair, which means that a lane is capable of both sending and receiving data simultaneously.

Generation 1 and 2 of PCI Express uses an encoding scheme called 8b/10b, this basically
means that each byte (8 bit) is encoded as 10-bit. Generation 3 uses a similar encoding, but it leaves less overhead: 128b/130b. As the name suggests, each 128-bit symbol is encoded as a 130 bit symbol. The main reason for employing this encoding scheme is that it makes error detection easier, the encoding is covered in greater detail in section 2.1.3. When calculating Gen3 bandwidth we do not take into account the overhead introduced by the encoding scheme (2 bits per 128 bit) because its so little it is not large enough to matter.

As mentioned earlier PCI Express sends packets over a link to communicate with other PCI Express devices. These packets are built incrementally as the data to be sent traverses through the layers in the protocol stack. The layout of a PCI Express packet can be seen in figure 2.1. Depending on the device the maximum payload size of a PCI Express packet is 4KB per packet.

![Figure 2.1: PCI Express packet format](image)

The PCI Express protocol stack consists of three layers (not counting core/software). See figure 2.2. The transaction layer either receives or sends data to the device core. If it receives data from the device core it will add its own header and an optional ECRC (End-to-End Cyclic Redundancy Code), then forwards the TLP packet down the stack to the data link layer. The data link layer will add a sequence number and a 16-bit LCRC. The DLL packet is forwarded down to the physical layer which adds start and end headers which encapsulates the entire packet. The packet is then transmitted out on the link.

### 2.1.1 Transaction Layer

The transaction layer receives Transaction Layer Packets (TLP) from the data link layer and forwards them to the software layer. The transaction layer also generates transaction layer
packets based on a request from the software layer. There are four different transaction types, each with its intended purpose. Table 2.1 lists the different types.

Routing Mechanisms

PCI Express devices communicate with each other over buses. In a typical system there are a number of devices, each of which can reside on a different bus, so there must be a way to connect these buses to facilitate communication over multiple buses. This is where bridges and switches comes into play. A bridge is an interface to other buses, for example a PCI, PCI-X or a PCI Express bus. A switch is built up by a number of ports, where a port is an interface to a single bus that multiple PCI express devices can share. A switch is basically a packet router that routes PCI express packets.

There exists several routing mechanisms: ID routing, address routing and implicit routing. ID routing routes packets based on unique IDs assigned to each device. This ID is called BDF (Bus, Device, Function). When an ID routed packet reaches a switch port, the port will first compare its own BDF to the target BDF of the packet, if the port is not the destination it will check if the target bus is below the port in the topology, if this is the case the packet is forwarded to the bus directly below the port. If not, the packet is forwarded to the other ports within the switch, which will then perform the same check. ID routing is used for configuration requests, ID routed messages and completions. This routing method is commonly used when configuring a PCI Express based device. Address routing is used with memory and IO requests. If an endpoint receives an address routed packet it checks the address against its own addresses. The packet is dropped if the address doesn’t match or it is accepted if it does match. Address routing can be used when the device is configured, and is makes for a convenient and way to communicate with the device for driver software or application level software. In Implicit routing the packet is routed based on a code in the packet header which indicates a known location in the topology.

<table>
<thead>
<tr>
<th>Request type</th>
<th>Transaction type</th>
<th>(Non-)Posted</th>
<th>Basic usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Read</td>
<td>Non-Posted</td>
<td>Transfer data to/from a memory-mapped location</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>Posted</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read lock</td>
<td>Non-posted</td>
<td></td>
</tr>
<tr>
<td>IO</td>
<td>Read</td>
<td>Non-posted</td>
<td>Transfer data to/from an IO-mapped location</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>Non-posted</td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>Read</td>
<td>Non-posted</td>
<td>Device function configuration/setup</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>Non-posted</td>
<td></td>
</tr>
<tr>
<td>Messages</td>
<td></td>
<td>Posted</td>
<td>From event signaling to general purpose messaging</td>
</tr>
</tbody>
</table>

Table 2.1: Transaction Layer Packet types

 Posted vs. Non-Posted

In a case where a requester sends a non-posted TLP the completer must respond with a completion packet. The completion packet can contain data, for example in response to a memory, IO or configuration read request. The completion packet can also contain no data at all, but simply report the transaction status. Posted transactions does not expect a completion packet.
The downside of posted transaction is that the requester has no way of knowing if the completer encounters an error. The upside is, of course, that the requester doesn’t have to wait for the completion packet and thus improves performance.

Flow control

Every port at a PCI Express link implements flow control. Before a transmitter can transmit a packet he must first make sure that the receiver is able to receive it. A device implements one or more (up to 8) virtual channels (VCs). A VC is simply a buffer acting as a queue for packets, each channel features fully independent flow control. This is useful, for example, in a case where a single traffic flow causes a bottleneck for all other traffic. The flow control mechanism in PCI Express is credit based, where a credit specifies a size unit in the VC buffer. The data link layer of the receiver will send packets back to the transmitter during a transmission containing information about the amount of credits in the VC being used. While the data link layer conveys flow control information, the transaction layer performs flow control operations and sends TLP packets based on credits. This means that flow control is in fact a cooperative feature between the transaction layer and the data link layer.

Quality of Service

PCI Express implements Quality of Service (QoS) in order to support different traffic flows. This basically means that PCI Express packets are treated differently based on what requirements with regard to performance. It is the transaction layer that is responsible for managing this feature, and in order to differentiate packets to support QoS, the Transaction Layer Packet (TLP) header includes a 3-bit Traffic Class (TC) field. This TC TLP header field is used to map traffic into different virtual channels (VCs). VC mapping is specific to a link and can change from one link to the next. The mapping of TCs to VCs is implemented in software an can change from one system to another. However there are some rules that must be obeyed;

2.1.2 Data Link Layer

The data link layer lies between the transaction layer and the physical layer. The data link layer communicates with the data link layer of a neighboring device with Data Link Layer Packets (DLLPs). This layer receives TLPs from the transaction layer, prepares them for sending, then forwards the packets down to the physical layer. It also receives packets from the physical layer then forwards them up to the transaction layer. It is the data link layers responsibility to assure the integrity of TLPs. It is also responsible for transmitting flow control information between a transmitter and a receiver, link initialization and power management. A DLLP is always 8 bytes, including the two framing bytes. The general layout of the Data Link Layer Packet is shown in figure 2.3.

| (8-bit) DLLP type | (24-bit) DLLP type dependant | (16-bit) CRC |

Figure 2.3: Data Link Layer Packet structure
Flow Control

PCI Express uses a credit based flow control scheme. Before flow control can be started it needs to be enabled and must be enabled for each VC (Virtual Channel) that is set up. While the transaction layer is responsible for doing flow control operations. The link layer is responsible for conveying flow control information.

The Ack/Nak Protocol

The main purpose of the Ack/Nak protocol is to ensure reliable delivery of TLPs (Transaction Layer Packets). To ensure the integrity of the packet an LCRC (Link Cyclic Redundancy Code) is added to the TLP. The Cyclic Redundancy Code is used for error detection. The transmitter calculates the CRC of the TLP then adds it to the outgoing packet. The receiver of the packet then calculates the CRC of the incoming TLP then compares its results to the LCRC. If this CRC check succeeds at the receiver side, an Ack (Acknowledgment) DLLP is sent back to the receiver. If the CRC check failed, a Nak (Not Acknowledged) DLLP is sent, so that the transmitter can resend it.

Sequence numbers

Each outgoing TLP is assigned a unique 12-bit sequence number. The transmitter keeps an internal counter which is incremented by one continuously with each TLP. The sequence number is used by the receiver in Ack or Nak DLLPs as a unique identifier for TLPs which is to be sent back to the transmitter.

Replay buffer

The transmitter will occasionally receive a Nak DLLP containing the sequence number of the packet which the transmitter must resend. The transmitter keeps all transmitted packets in a buffer called the Replay Buffer. The Nak packet contains the sequence number of the packet which was not correctly received. The transmitter must then resend all non-acknowledged packets up to this sequence number. The same principle is true for Ack packets; The Ack contains the sequence number of the last correctly received packet, so once an Ack is received all packets up to that sequence number is discarded from the replay buffer since it is no longer needed.

Replay Timer

This timer counts the time until the next replay must happen. A replay simply means to resend all packets in the replay buffer. If the timer expires at some point it means that the transmitter has sent one or more packets that was not successfully received or an Ack was not received within the specified time frame. The transmitter must then resend the entire replay buffer. After a replay the timer is reset. To prevent the timer from continuously expiring it is reset on each received Ack/Nak. The timer value is calculated by multiplying the AckNak_LATENCY_TIMER value by 3.
2.1.3 Physical layer

The physical layer is the lowest level in the PCI Express protocol stack. It communicates directly with the data link layer above it, and indirectly with the transaction layer. Two framing bytes are added to the DLLP received from the layer above. These framing bytes are added so that the receiver can detect packet boundaries. The start byte is added at the start of the packet and the end byte is added at the end. The physical layer is divided into two logical sub-blocks: The logical block and the electrical block. The logical block mainly concerns itself with things like preparing packets for serial transmission and preparing received packets and sending it up the protocol stack. Only aspects of the logical block will be discussed here.

Bandwidth

Each generation of PCI Express improves on its earlier version in terms of bit rate; Gen1 has a bit rate of 2.5 GT/s, Gen2 twice as high (5.0 GT/s) and Gen3 8.0 GT/s. GT/s (Gigatransfers per second) is a unit of measurement to describe the number of transfer operations performed each second. For example, a bus 32 bits wide will transfer 32 bits on each transfer operation. With a transfer rate of 1 GT/s the bit rate is $32 \times 10^9$ transfers per second. To calculate the bandwidth we have to consider several things; the bit rate, the number of lanes, the fact that each lane has a transmit and receive pair, and the encoding scheme (8b/10b or 128b/130b). Gen1 bandwidth (GB/s): $(2.5 \text{ Gb/s} \times 2 \text{ directions}) / 10 \text{ bits per symbol} \times \text{lanes}$. Gen2 bandwidth (GB/s): $(5.0 \text{ Gb/s} \times 2 \text{ directions}) / 10 \text{ bits per symbol} \times \text{lanes}$. Gen3 bandwidth (GB/s): $(8.0 \text{ Gb/s} \times 2 \text{ directions}) / 8 \text{ bits per byte} \times \text{lanes}$.

Byte encoding

PCI Express uses two different kinds of encoding schemes. For 2.5 GT/s (Gen 1) and 5 GT/s (Gen 2) uses 8b/10b encoding while 8 GT/s (Gen 3) transfers uses 128b/130b encoding. Byte encoding is performed after the byte striping process and is done independently on each lane. The main reason for doing this encoding is to maintain DC balance, enhance error detection and to allow clock recovery. The 8b/10b encoding scheme converts bytes (8 bits) into 10-bit symbols. While this achieves the advantages stated above it also introduces a 2-bit overhead (20%). However it is still considered to be acceptable to achieve these goals.

8b/10b encoding divides the byte into a 5-bit section and a 3-bit section. The 5 bits are mapped to 6 bits and the 3 bits are mapped to 4 bits. Instead of converting 8 to 10 bits 128b/130b encoding converts a 128 bit block to 130 bits, thus reducing the overhead, and improving performance.

Clock recovery

In PCI Express the transmitter (Tx) supplies the clock to the receiver (Rx) to use for latching incoming data. The transmitter will embed the clock into the data stream and the receiver recovers it from the stream. This accomplished by using a Phase-Locked Loop (PLL). The receiver takes the input stream as a reference clock and compares it to an output clock with a specified frequency. Based on the comparison, the frequency of the output clock is either increased or decreased until the Tx and Rx clock matches.
2.2  Dolphin Interconnect Solutions

So far we have touched on some lower level mechanics of the PCI Express standard, but nothing about how, from a programmers or end-users perspective, to utilize it. While there are many different implementations out there the remainder of this chapter concerns itself with the technology we used in our implementation, developed by Dolphin.

Dolphin Interconnect Solutions [21] is a company headquartered in Oslo. They make high bandwidth, low latency, interconnect products that are used to connect computers together to create high performance computing platforms. They have a long line of products used for interconnecting computers, but for this thesis we used the IXH610 Host adapter (see section 2.2.1). Some applications in which their products are being used are financial trading applications, real time simulators, video information distribution and network file systems. To make software for their hardware it is most common to make use either of their Supersockets API or the SISCI API.

2.2.1  Hardware

Dolphin has a long line of interconnect hardware, but for developing our prototype we used their IXH610 Host adapter and the IXS600 PCI Express switch. The IXH610 is an interface which is based on a PCI Express Gen. 2, non-transparent bridging architecture [22]. It provides 40 Gbits/s performance and can be configured to perform both transparent and non-transparent bridging.

The IXS600 is a PCI Express switch which enables switching for Dolphin IX adapters. It is PCI Express Gen. 3 compliant and can scale up to 20 nodes [23].

2.2.2  Software

In addition to hardware, Dolphin also produces software to use with their hardware. While they provide drivers for multiple operating systems, they also provide application layer software which makes it easy for existing applications to make use of their solutions without making any changes to the application itself. This section gives a short introduction to Supersockets and their TCP/IP driver.

Supersockets

Supersockets is perhaps the easiest way of writing programs for Dolphins products. In essence, it is a high level API implementation of the well known Berkeley sockets API [24], made to run on the PCI Express architecture. Supersockets are unique in contrast with the traditional socket interfaces because it bypasses the networking stack in the operating system (OS). The OS networking stack suffers from multiple performance bottlenecks such as copying data several times, protocol overhead, etc. All this is bypassed in Supersockets which instead combines the use of programmed Input/Output (PIO) and Remote Direct Memory access (RDMA). In short, RDMA is used for larger data transfers, whereas PIO is used for shorter transfers.

Supersockets also have the advantage that there is no need for changes in already existing programs that wishes to use Supersockets. On Linux it is enough to run the program through
a wrapper program which intercepts the `socket` function call in libc. On Windows a proxy application enables specific programs to use the PCI Express path through the Layered Service Provider. If, for some reason, the PCI Express link should go down Supersockets automatically switches to use the regular OS network stack.

**TCP/IP Network driver**

Dolphin also has their own networking stack which is able to perform TCP/IP over PCI Express (IPoPCIe). It is specially optimized for the PCI Express architecture. This means that any networking application, both in user space and in kernel space, can communicate using Dolphins PCI Express technology in a transparent way. Some applications for this technology are Networking/cluster file systems not supported by Supersockets and Microsoft Hyper-V live migration.
2.3 The SISCI API

SISCI (Software Infrastructure for Shared-Memory Cluster Interconnects) is, compared to Supersockets, a low level API. It was developed in a European research project [25], and provides an interface to user-space programs that can be used to program PCI Express adapter cards configured to operate in non-transparent bridging (NTB) mode.

With the SISCI API programs can allocate and share access to local memory via remote memory mapping or Remote Direct Memory Access (RDMA). It is also possible to set up custom interrupts and trigger them from a remotely connected machine. In the following sections we will explain the essentials of the SISCI API.

2.3.1 Memory Segments

Perhaps the most important concept in the SISCI API is the memory segment. Before any communication and data transfers can be performed between interconnected nodes in a cluster the running process must allocate a memory segment. Memory segments intended to be shared to other machines must be allocated through a special SISCI function, SCICreateSegment. This function ensures that the segment is associated with the underlying driver software and adapter hardware connecting the cluster members. A machine can allocate as many segments he/she wishes, as long as there is enough available memory. Because of this, and that remote
nodes must be able to connect to each and every segment on a node, the programmer must give his/hers segments an ID number. Local segments can only have one ID and not two local segments can have the same ID. However, different nodes across the cluster can create segments with the same ID as long as it is unique to that machine.

Remote nodes must use the ID of the node and the and the ID of the segment when connecting to a specific segment. In SISCI there is also the option to not allocate any memory with the segment. This is useful when used together with the function SCIAttachPhysicalMemory, which is privileged an can be used to attach any physical memory of the local machine to the segment. This is often used for mapping memory of a physical device.

Once a segment has been created it exists in its default state, called "not prepared". In this state it can not yet be accessed by remote nodes because it is not ready to be used by the host adapter. It can however be used locally, by mapping the segment into the process’ virtual memory space (see section 2.3.2). From the default state the programmer may choose to call the function SCIPrepareSegment; this function will prepare the segment to be used by the host adapter. For any node to be able to connect to a segment, it must be in the "available" state. The programmer may toggle the availability of a segment with the functions SCISetSegmentAvailable and SCISetSegmentUnavailable. Figure 2.7 illustrates the different states in which a SISCI segment can be. The memory a segment refers to can be used in two ways, either through Programmed Input/Output (PIO) or RDMA.
2.3.2 Programmed Input/Output

Programmed I/O is a mechanism of transferring data between the local computer and a remote segment. In contrast to DMA, PIO transfers must be performed with the help of the CPU. PIO is triggered when the running process uses CPU instructions to access a memory segment of a remote machine. Some examples of such instructions for Intel x86 are lods, stos and push [26]. SISCI offers two functions for mapping a segment into a process’s virtual memory space – SCIMapLocalSegment and SCIMapRemoteSegment. The former function maps a local segment while the latter maps a segment in a remote machine into a local process’ memory space. Once a remote segment has been mapped, it can be used just like local memory, e.g. memory can be accessed by the use of traditional C functions such as memcpy or bcopy. PIO is generally considered to be inferior to DMA, but for smaller transfer sizes (2-4 kilobytes in our tests) PIO can be faster. This is mostly because with DMA there is an initial setup time before the actual transfer can start.

2.3.3 Remote Direct Memory Access

Direct Memory Access (DMA) refers to the process of reading or writing to system memory without the involvement of the CPU. RDMA refers to the same concept as DMA, but the important difference being that the transfers are performed between separate machines. To do a Remote DMA (RDMA) transfer, the functions SCIStartDmaTransfer and SCIStartDmaTransferVec can be used. When DMA transfers are done with the SISCI API, a DMA engine residing on the PCI Express host adapter is initialized and programmed. This is the only step in which the CPU is involved with the transfer. This step provides a significant enough overhead that, as mentioned in section 2.3.2, PIO transfers are usually faster for smaller message sizes.

A major advantage with DMA, besides it being very fast, is that the data transfers are performed asynchronously. This means that the CPU is free to perform other tasks while the data transfer is in process. DMA transfers are commonly performed between devices internally in a machine. This type of DMA includes, but are not limited to, transfers between host memory and disc drives, network interface cards, or a PCI Express host adapter like the Dolphin IXH610 we used in our experiments. It is also possible to do DMA transfers directly between devices. Shown in figure 2.9 is the performance of RDMA transfers between two machines equipped with IXH610 cards.

Essential to DMA transfers is the DMA queue, sci_dma_queue. A DMA queue must
be created before a DMA transfer can be performed and is used internally by SISCI to program the DMA engine residing on the host adapter. Figure 2.8 illustrates the different states of a dma queue, where IDLE means that the queue is created and is empty, POSTED means that the queue has been sent to the DMA engine and is being processed, DONE means that all data transfers associated with the queue is done, ERROR means that one or more of the transfers in the queue failed, ABORTED means the program aborted the queue while being processed by the DMA engine.

![Figure 2.8: A state diagram showing the different states of a SISCI DMA Queue](image)

### 2.3.4 Interrupts

Interrupts is a feature in SISCI which allows remote machines to tell a node that a predetermined condition has occurred. After connecting to a remote interrupt it can then be triggered, this can result in two different behaviors in the remote machine, depending on how the interrupt was configured: 1) The function `SCIWaitForInterrupt` can be used to block on the interrupt. The thread can then only be woken up when the interrupt is triggered (`SCITriggerInterrupt`) by one of the remote nodes connected to it. 2) The interrupt can also be configured to asynchronously call a callback function whenever the interrupt is triggered. This is configured when the interrupt is created, using the function `SCICreateInterrupt`. Like SISCI segments, interrupts are identified by a unique number chosen by the programmer. This number must be used, in addition to the node id, when connecting to a remote interrupt (`SCIConnectInterrupt`).

### 2.3.5 Events

SISCI also allows for a program to catch events. There are many ways in which an event can occur and what actually causes it. An event can be generated when a network component breaks, a cable can be unplugged, a segment can be made unavailable by an unexpected crash, completion of a DMA transfer etc.

Not all events are made visible through the SISCI API because they are handled transparently at a lower level, but some are forwarded to an application. There is no requirement for the programmer to deal with events; they can simply be ignored, or they can be caught and dealt
Figure 2.9: SISCI DMA latency and bandwidth

with. Events can be caught by either blocking until the event occurs, or asynchronously through a user-defined callback function which is called whenever the interrupt occurs.

For example, there are a number of events associated with a memory segment; SCI_CB_CONNECT / SCI_CB_DISCONNECT meaning that a node has connected/disconnected from a segment, SCI_CB_OPERATIONAL / SCI_CB_NOT_OPERATIONAL meaning that the route to the connected node is available or temporarily unavailable or SCI_CB_LOST meaning that the connection is permanently lost due to an unrecoverable event happened at the connected node. These events can be caught asynchronously by passing a callback function when creating a segment, or by calling SCIWaitForLocalSegmentEvent which will block until an event occurs.

### 2.3.6 Reflective memory

Dolphin has extended the SISCI API to support multicast/reflective memory. This allows for one node to perform a multicast, e.i one node transmits a packet and multiple nodes receive it. This is multicast in the true sense; only one packet is sent from the sender, and the switch replicates and distributes the packet to the other receivers. This is done by allocating a multicast segment which must be identified with a unique ID. Once set up, it can be read and written to in the same way normal SISCI segments are used.
2.4 Summary

In this chapter we have given an introduction to PCI Express and Dolphin’s PCI Express based products we have used. We have learned about the different generations of PCI Express standards and some differences between them. We have also learned that a PCI Express link can consist of up to 32 lanes in Gen 3. and that the number of lanes in a link affects the bit-rate. PCI Express consists of three main protocol layers; 1) The transaction layer which describes the different request and transaction types. The transaction layer is also responsible for handling flow control operations and sends TLP packets based on credits. 2) The data link layer which is responsible for flow control initialization and conveying flow control information. It is also responsible for sending Ack/Nak packets and detect errors in incoming packets by the help of a CRC code. 3) The physical layer which is responsible for spreading data across available lanes in the link as well as byte encoding and clock recovery. We have also presented Dolphin and the SISCI API which enables us to program the PCI Express enabled hardware we utilized. With knowledge of the essential concepts of this API we will present extensions to it that enables can be used to perform remote DMA transfers between nodes in separate clusters.

In the next chapter we will present the design and implementation details of our prototype as well as the proposed SISCI API extensions to perform inter-cluster RDMA transfers.
Chapter 3

Design

In the previous chapter we introduced the main technology we used in our prototype along with an introduction to the SISCI API. In this chapter we present the design aspects of our prototype for performing long range Direct Memory Access (DMA) transfers between two separate PCI Express clusters. As stated in the limitations section in chapter 1, there is no PCI Express session between the clusters, this would require the PCIe protocol to be modified, which is outside the scope of this thesis. We have instead created a solution that emulates an existing PCI Express session between the clusters.

Our design consists primarily of two major components; The *proxy* and the *endpoint* node. Each cluster has one node which works like a proxy. This node is different from other nodes as it works like the interface to remote clusters. Every data transfer, incoming or outgoing, is relayed through this proxy. Its main responsibilities are to parse incoming packets and forward them to the intended local receiver node. It is also responsible for processing and forwarding outgoing packets received from local endpoint nodes to the destination cluster.

The endpoint is a node in a cluster. The endpoint establishes a connection with the local proxy upon initialization. This connection is vital, as it is used whenever a transfer to a node in a remote cluster is made. To ensure reliable delivery, the endpoint also maintains a separate direct connection to endpoint machines. This connection is used for control messages to synchronize and detect errors that might occur.

With our design, the data flow from a local endpoint node to a destination endpoint in a remote cluster travels through a minimum of four machines, including the two endpoint nodes. The transmitting endpoint forwards data to its local proxy, which in turn forwards it to the proxy in the remote cluster, which then in turn forwards it to the destination endpoint. Before the transfer itself takes place, the control channel is used to agree on the transfer. When the receiving endpoint has received everything, the control channel is used again to notify the transmitting endpoint that the transfer was completed.

3.1 Requirements

While we specified many requirements for our system, the main goal was to create a system which could connect separate clusters together in such a way that nodes in the separate clusters easily can perform transfers between themselves. The link which makes the connection between clusters are an Ethernet link.
Another requirement is that the system must support reliable delivery. This means, in short that each transfer must arrive at the receiving end without any bit-errors and without parts of it dropped. In a reliable delivery system, the transmitter is dependent on some kind of acknowledgment from the receiving end, that confirms that the transfer was a success. For example, UDP provides an unreliable service because a transmitter has no assurance whether the package arrives at the destination, nor does it provide any message if there was an error during transmission. TCP, for example, is a reliable delivery system. It is a connection oriented protocol which provides the sender notification whether the transmission was a success or not.

A central property of cluster networks is that they provide high bandwidth and low latency. This is also key for a system that establishes inter-cluster connections.

Another important requirement is any node in cluster A should be able to connect to any other node in a remote cluster B. If more than one remote cluster is present, the system should also manage and maintain connections to them. Nodes must also not be limited to connecting with nodes in only one other remote cluster.

As this thesis focuses on implementing a workable prototype, we do not concern ourselves with security measures which might be necessary in a deployment scenario. Data being replicated between remote data centers might be sensitive in one form or the other. It might be sensitive details about employees in a company or it might be user e-mails. To prevent data being leaked, we would need to implement some measure to prevent this, for example a stream encryption mechanism such as SSL/TLS [27, 28]
3.2 Endpoint

The endpoints machines are responsible for initializing and starting remote data transfers. They have to synchronize and negotiate with the remote node how the transfer executed, and make use of the proxy to perform the actual transfer. The endpoint implements an API for performing the remote transfers. This API is presented in section 4.7. In this section we will explain the concepts and inner workings of our endpoint.

3.2.1 Initialization

The endpoint maintains two shared SISCI segments. The data segment and the control segment. The data segment is used as the main data storage for receiving and transmitting data to a remote node, whereas the control segment is used when receiving data to keep track of how many bytes of the total transmission is transferred. These two segments are allocated and initialized upon startup of the endpoint and live on for as long as the endpoint stays operational.

The second thing the endpoint does is to connect to the proxy. The endpoint will first map the control segment of the proxy. This segment contains information such as the number of buffers in the proxy buffer segment and their size. Using this information the endpoint connects and maps the transaction ring and the buffer segment data structures of the proxy. At this point a connection between the endpoint and the proxy has been established, but the proxy is not yet aware of the presence of the endpoint, so the endpoint will update the proxy control segment with its node id, segment size and segment ID, finally the "connect" field is set so let the proxy know that there is a new endpoint. This field is periodically checked by the proxy and a connection to the endpoint is made if it is set.

The initial implementation as described here did not have support for multiple local endpoints or remote clusters which is why there was no need for a more sophisticated connection routine at that point. As described in section 4.6, that was changed in a later version.

3.2.2 Connection establishment

Before a remote transmission can start, the transmitting endpoint must first establish a connection with the receiving endpoint. This phase consists of establishing a direct TCP connection between the transmitter and the remote endpoint, and exchanging information that must be used in a transmission. As explained in section 3.3.5 the buffers processed by the proxy consists of a header, containing the remote proxy IP address and the remote endpoints node ID. This information (remote proxy IP and the endpoints node ID), is relayed to the transmitting node upon the connection establishment. After the transmitter has received this information, he replies with an OK message to indicate to the remote endpoint that a transmission might start any time. After the OK message is transmitted, the connection establishment phase is concluded.

This connection is not shut down after this phase is finished however. This same connection is also used during transmission as a control/synchronization channel.

3.2.3 Control channel

From our requirements we remember that reliable delivery is a requirement. It is with this in mind that we introduced a control channel that is used to control and synchronize the transmis-
sions. This control channel is set up during the connection establishment phase as described in the previous section. As we will see, the control channel enables the transmitter to know whether his transmission was completed or not. It also enables the receiving end to handle and control the transmission it is about to receive.

The type of traffic over this channel is very different compared to the channel between proxies. Where the proxy-to-proxy channel primarily consists of large amounts of data, our control channel consists of small payload sizes and is more prone to latency than bandwidth.

The control channel follows a protocol which is used before and after a remote transmission can take place. The sender needs to know whether the endpoint is alive and ready to receive data. The transmitter is also dependent on knowing when the transmission is completed. The protocol consists of a series of message exchanges that is performed on a per-transmission basis, meaning the same series of message exchanges are performed for each transmission. The messages are of a fixed size and consists of three fields: 1) type, which describes what type of message this is; 2) offset, which depending on the type of message contains either an offset into the remote buffer or a magic number used in "hello" messages; and 3) size, which depending on the type of message contains either the size of the transmission to come, or a magic number used in "hello" messages. In figure 3.2 the layout of the control packets can be seen.

![Figure 3.2: Control packet layout](image)

Before starting a transmission the sender sends a HELLO message to the remote endpoint. This type of message does not contain any useful data, but is instead sent to make sure the remote node is alive. If everything is fine the receiving end replies with the same HELLO message. Second the transmitter sends a DMA_START message which contains the offset into the remote buffer, it is starting at this offset, the data will be placed. The DMA_START message also contains the total size of the transmission. If everything is fine, the receiver will respond with a DMA_START_OK message, the receiver can also deny the transmission to start in the first place. In this case he will respond with a DMA_START_DENY message.

After the sender has transmitted all the data through the proxy, he waits for the receiver endpoint to send a DMA_COMPLETED message, which means everything was received. Finally the transmitter sends an OK message to conclude the transfer. An example of a successful transaction is illustrated in figure 3.3. The next section explains how the transaction phase is done.

### 3.2.4 Transferring data

After two endpoints are agreed, the data transfer can start. In Section 3.3 we introduced the proxy and the role it plays in the system. All data going out of a local cluster and to a node in a remote cluster are transferred through the proxy of each their respective cluster. We also
introduced two important data structures; the buffer segment and the transaction ring. These data structures are not only central to the proxy but also to the endpoint, as it also makes use of them during a transfer.

Before the endpoint can DMA any data to the proxy, it must allocate space. The endpoint uses the slab allocator to allocate buffers from the buffer segment in the proxy. This is where the data will end up. After a buffer has been allocated, local data is DMAed into it using the SISCI function `SCIStartDmaTransfer`, then the header is prepended to the buffer. The header fields are essential and are used by the proxy to further forward the buffer to its destination. The header fields are; remote proxy IP (the address of the proxy in the remote cluster) and remote node id (the node id of the remote endpoint). The packet format is represented in figure 3.7.

As the proxy buffer sizes are subject to change, they will not always be big enough to fit all the data in one operation. When this is the case, the steps described above are repeated until everything has been transferred.

When all the data has been transferred the endpoint must insert nodes into the transaction
ring for the proxy to process and forward them. This process goes as follows: 1) Take the transaction ring lock, to ensure only one accesses it at once; 2) Allocate a transaction ring node using the slab allocator; 3) Initialize it with a buffer number corresponding to the one(s) used in the DMA transfers, and set the number of bytes of the buffer actually used; 4) Insert the node into the transaction ring at the end of the ring; 5) Repeat steps 2-4 until all nodes are inserted; 6) Release the transaction ring lock.

When the transaction ring lock is released, the proxy will detect that there are new nodes in the ring, and start processing and forwarding the buffers to its destination cluster.

### 3.2.5 Receiving data

From the remote endpoints perspective the process of receiving data is much less complex, but it still has some responsibilities. It must first decide whether or not to accept the transfer, and assuming a transfer has been started it must report back to the sender node when it has received everything. The DMA\_START control channel message contains the offset into the buffer in which the incoming data will end up, and the size of the transfer. The offset is copied into the control segment of the endpoint. This value is continuously updated by the proxy as the data is received. Using the size received from the endpoint and the offset updated by the proxy allows the endpoint to monitor the progress of the transfer, and send the DMA\_COMPLETED control message when everything has been received.

### 3.2.6 Summary

In this section we have introduced the first design of the endpoint. We know how the transfers are synchronized and what the process looks like from starting a transfer to completing it. This first version was a good version but had some missing features and had much room for improvement both in performance and functionality. In chapter 4 we explain the most important optimizations and improvements that was done with the initial implementation as a starting point.

In the next section we introduce the design of our proxy and the role it plays in our system.

### 3.3 Proxy

To facilitate communication between two clusters we chose a solution where all data transfers travels through a proxy. The proxy is central to every cluster and acts as a connection point, for both local nodes and other remote proxies. The proxy can be seen as an application layer network bridge where its primary purpose is to connect multiple networks. Figure 3.1 illustrates how a typical transfer happens. The data first travels through a PCIe switch before it arrives at the proxy. The proxy then parses the packets and sends them to the remote proxy where the packets are parsed and finally transmitted to the receiving node.

It is highly desirable that links between clusters are as powerful, in terms of high bandwidth and low latency, such hardware can be very expensive. Because equipping each node with such hardware can quickly become expensive, we chose our proxy design where only one node, the proxy, needs this. This has the potential to drastically reduce deployment costs. An alternative to a proxy solution where each node performs transfers directly between themselves will not only quickly become expensive, but it also becomes a scalability problem.
There are downsides with a proxy based solution too, one of which is that the proxy becomes the bottleneck of the system. Nodes in the cluster will have to share the same inter-cluster link. In a worst case scenario where all nodes are transmitting at the same time the theoretical bandwidth is reduced to \( \frac{\text{total capacity}}{\text{number of nodes}} \). However as the cluster and demand for more bandwidth grows, more proxies could be added. This would increase the total capacity for inter-cluster transfers per proxy/gateway added. If this is done the need for a scheduling algorithm arises, so that the available proxies are evenly distributed across nodes in the cluster so that the total capacity is utilized in an efficient manner.

With a central connection point system such as this, the bottleneck also becomes the weakest point in the system. Should a proxy unexpectedly fail and become inaccessible, every connection, incoming and outgoing, would be broken until the proxy is repaired and running again.

With our proxy based solution that must be able to handle multiple clients, we need to manage everything from local and remote connections to synchronizing access between the nodes and the different components of the proxy. This means additional overhead and complexity in parsing headers. Moving all this responsibility from the clients to the proxy means that the clients can spend less time actually executing the transfer, and instead spend more time on more important tasks.

The fact that the proxy handles every connection on the system, also means that it becomes a central point for configuration. Parameters such as a maximum number of local and remote connections are configured only at the proxy but affects all other clients in the cluster. It also enables the cluster bandwidth capacity to be throttled or boosted as the need arises.

### 3.3.1 Buffers

![Buffer Segment Diagram](image)

Figure 3.4: An illustration of the proxy buffer segment

From chapter 2 we know that RDMA and PIO transfers in SISCI can only be performed between SISCI segments. This segment is used both by the proxy and the client; It is used as a destination for client RDMA and PIO transfers when transmitting data to a remote node, and it is used by the proxy for transmitting incoming data to a destination client.

As one of the shared memory segments, the buffer segment is the biggest one. As illustrated in figure 3.4 it is further divided into many smaller buffers. As opposed to a segment with one big buffer, a segment with many smaller buffers is an advantage because it enables both the client and the proxy to use parts of it at the same time. As this quickly becomes the most heavily
utilized data structure in the system, a segment with only one giant buffer would mean that only one node could be using it at the time. With all the waiting, this would quickly become a major bottleneck.

These buffers are used by both the proxy and nodes in the local cluster, when transmitting data. In the buffer segment, every buffer is of a fixed size, and can contain as many buffers as long as there is memory left for SISCI to allocate. In our initial version the buffer segment contained 1024 buffers of 2048 bytes. These sizes were changed in later versions after testing proved that this resulted in poor performance. In chapter 5 we discuss how these sizes affects overall performance.

### 3.3.2 Transaction ring

As mentioned in the introduction to this chapter, the proxy is responsible for forwarding data from a local endpoint to the remote proxy. We already know that data ready to be sent are located in the buffer segment described in section 3.3.1, but the proxy also need a way to identify which of the buffers to process and send. The transaction ring is the data structure to fulfill this need; it is a shared doubly linked ring buffer where each element contains a size field and a reference to a buffer which is ready for processing and forwarding to the remote proxy. Refer to figure 3.5 for an abstract representation of the data structure.

![Buffer segment diagram](image1)

![Transaction ring segment diagram](image2)

Figure 3.5: Each element in the ring buffer segment refers to a buffer

As with the buffers in the buffer segment, the size of transaction ring segment cannot be changed after it is first initialized. An element in the ring must be allocated before use, and freed when it is no longer needed. Allocation and freeing of elements is performed with the slab allocator. This way, both an endpoint and the proxy can allocate and free elements from the same segment as needed.

Whenever the transaction ring contains entries and is ready to be processed the proxy will iterate the ring, starting at the head. For each element it encounters, the buffer it references is
retrieved, processed (see section 3.3.5), and sent to the remote proxy. Finally the element is unlinked from the ring and freed, so that it can be used later.

The proxy is not the only node to manipulate the transaction ring, therefore it is vital that only one thread, local or from a remote machine, can manipulate it at once. If, by chance, the proxy tries to unlink an element, and at the same time another machine attempts to link an element, it could lead to an unrecoverable state of the transaction ring. The mutex that ensures sequential access is defined in the third segment shared by the proxy - the control segment.

### 3.3.3 Control Segment

The control segment is the third and last SISCI segment shared by the proxy with local endpoint nodes. It is a special segment mainly used by the endpoint. It is used when a local node connects and disconnects from the proxy, and before getting access to the transaction ring.

```c
struct control_segment {
    u32 buffer_size, num_buffers; // Size and the number of proxy buffers
    spinlock_t tx_ring_lock; // lock for accessing the transaction ring
    volatile client_connect_t connect; // An integer used to synchronize (dis)connection to the proxy
    volatile int client_seg_size, client_seg_id, client_node_id;
};
```

Figure 3.6: Control segment

When a local node connects, the proxy will check the connect-field in the control segment (see figure 3.6) then connect to the SISCI segments hosted by the connecting client. Endpoint segments are discussed in section 3.2.

A possible solution for the proxy to detect the presence of a new endpoint connection is to periodically check the connect-field and based on the value, take the necessary action, either connect or disconnect. This solution quickly becomes both impractical. For optimal performance we could allocate a thread for the sole purpose of repeatedly check the connect-field, and whenever it changes do an action. This, however will occupy an entire processor core, which leaves less processing power for other tasks. Connection and disconnection of clients happens relatively rare compared to other tasks such as processing the transaction ring, so it is not very time critical anyway. This is why we introduced SISCI interrupts instead.

### 3.3.4 Connection establishment

The proxy has to establish a connection with a local endpoint before it can start performing remote transactions. At the final step in the endpoint initialization process, it will trigger a remote interrupt in the proxy. This is an important step because it allows the proxy to be notified that a new client is present and will make use of the proxy some time in the future. Before the interrupt is triggered, the endpoint node must first initialize three fields in the proxy control segment; the client_node_id field which is set to the dolphin ID of the node, the client_segment_id field which is set to the segment id of the node and the client_segment_size which is set to the size of the node segment.
When the node triggers the interrupt and the interrupt handler function is executed, the proxy reads these fields and uses them to connect to the client data-segment and its control segment.

In our first implementation, references to the two client-segments are kept in a global variable, thus only supporting one endpoint client. This was changed in a later version.

### 3.3.5 Proxy to proxy communication

We have already covered the buffers segment and the transaction ring data structures which is where the proxy retrieves the data to process and send, but we have not explained how these buffers are being processed and sent. In this section we explain how the actual data transfers between proxies are performed.

One of our requirements is that the proxy to proxy link should be an Ethernet link. An important property for our system to have is that the data being received in the remote node should be identical to the data being sent. This means that we need a protocol that can detect any errors in the data which might have occurred during transmission, and that the data is received in the correct order. Other important properties of the protocol must have is a flow control mechanism that can regulate the packet flow in such a way that gives the highest possible throughput, but at the same time does not send at a too high rate so that data is not lost along its way to the destination. A possibility for us to use is User Datagram Protocol (UDP). But although UDP has support for message integrity through checksumming it is too minimal as this is a transport control without a guarantee for delivery, correct message ordering or congestion control. For our purpose Transmission Control Protocol (TCP) is a good choice as it support all of our needs. TCP however has a wide range of options that can improve performance depending on the characteristics of the stream.

So, we know where the outgoing data is retrieved from and that our proxy-to-proxy link is a TCP connection. We do not yet know how and when a proxy-to-proxy connection is established. This connection is established as the transaction ring is processed and a packet is on its way to being processed. To understand this we need to take a look at how the transaction ring buffers are processed and their layout.

Every data buffer contained in the transaction ring has a header prepended to the actual data. As figure 3.7 illustrates, the header is 8 bytes in total and contains two fields: remote proxy IP address and destination node ID.

The first thing the proxy will do when processing a buffer is to retrieve the remote proxy IP and check if it has established a connection to it earlier. If a connection does not yet exist, the proxy will set up a connection, in the form of a TCP socket. A reference to this connection is stored in a global variable, and is kept alive for as long as both proxies are operational.

The remote proxy has no use for the IP address field in the buffer as it is received, this is why the IP address field is replaced before it is sent. The field is replaced with something more useful to the remote proxy, namely the total size of the buffer. There is no guarantee however that the entire buffer is actually in use, which is why each transaction ring element contains a field which describes how many bytes of the buffer is actually in use. It is this field the proxy replaces the remote IP address field with. This is essential because it provides the receiving proxy with the means to retrieve whole packets and not partial ones, which will quickly cause problems. The finalized packet as it looks after the proxy has processed it is illustrated in figure 3.8. When the buffer has been processed and prepared, it is sent to the remote proxy over the TCP connection.

From section 3.3.4 we know that when the endpoint connects to the proxy, he provides
some information to the proxy as to how the proxy can connect to the segments provided by the endpoint. We also know that a reference to the endpoint node along with the connected segments are stored on the proxy, so when the incoming packet is processed this reference is retrieved based on the "destination node ID" field in the packet. Should that node not have connected to the proxy yet, the packet is silently dropped.

Now the proxy has all it needs to transfer the data to the endpoints data segment. The two header fields are stripped away as they are not of any use to the endpoint, then the transfer is executed using RDMA using the SISCI function SCIStartDmaTransfer. Finally when the transfer has completed, the proxy updates a "number of bytes transferred" field in the endpoints control segment. This way the endpoint knows how much of the transfer has been completed.
3.4 Summary

In this chapter we have introduced the design aspects of our prototype system for performing inter-cluster data transmissions. The system comprises of two main components; the endpoint and the proxy.

The endpoint is responsible for initializing and starting remote data transfers which are sent through the proxy and forwarded to the destination cluster. The proxy maintains two main data structures - the buffer segment and the transaction ring, which are used to store and track which buffers are ready to be transferred to the remote proxy.

In the next chapter we go into implementation details as well as steps taken to optimize performance.
Chapter 4
Implementation

In the previous chapter we presented the design aspect of our prototype. In this chapter we go into greater details on how we implemented it, as well as the optimizations we made.

We present some implementation details of the most important mechanisms in our design as well as the optimizations we made to the different components, presented in the design chapter and this.

We have also implemented functionality to complement the SISCI API, that can be used to perform DMA transfers from a node in one PCIe cluster to another node in another PCIe cluster. This functionality is also presented in this chapter.

4.1 Slab Allocator

The proxy shares multiple segments with many local endpoints. The buffer segment and the tx-ring segment is split into multiple smaller fixed-size chunks of memory which local endpoints make use of when transferring data. Since many nodes make use of the same segments there must be a way to make sure no more than one node is using a buffer at the time. This is why we implemented a distributed slab allocator. A slab allocator is a memory allocator commonly used in modern operating systems such as Linux. It was first introduced by Jeff Bonwick in SunOS 5.4 [29] as a fast and space-efficient way to deal with kernel objects which is being allocated and freed frequently. Our slab allocator works in a similar manner, the main difference being that it is distributed. This means that any node that needs to allocate or free buffers can do so themselves, thus offloading this task from the proxy. It is a memory allocator that keeps track of free fixed-size chunks of memory (buffers) within a memory segment. By taking advantage of the fact that all buffers are of the same size, the slab allocator is able to allocate buffers very efficient and without wasting any space. As can be seen in figure 3.4, there is a section at the head of the buffer, that we named "Metadata". This is a data structure used by the allocator when allocating and freeing buffers. The metadata part of the buffer consists of 5 fields (see figure 4.1), the total number of buffers, the size of each buffer, a mutex for ensuring sequential access to the allocator, an integer containing the number of the next free buffer, and a pointer to an array which makes up the linked list of free chunks of memory.

Upon initialization of the slab allocator, each entry in the free list is initialized with its own index number + 1, which acts both as a pointer to the next free chunk of memory and as the next index in the array with a free buffer. For reasons explained in section 4.2 we store the index of the buffers in the free-list instead of the virtual address of the buffers. The next_free
integer contains the buffer number of the next free buffer. Mutexes ensure sequential access to
the allocator. Discussed in section 4.3

```c
struct slab {
    // size of a slab buffer
    unsigned seg_size;
    // number of slab buffers
    unsigned num_buffers;
    spinlock_t mutex;
    int next_free;
    int freelist[num_buffers + 1];
};
```

Figure 4.1: Slab header

### 4.2 Addressing

An important thing to note regarding the segments shared by the proxy is that more than one
machine uses them. After connecting to the segments, using the SISCI API (see section 2.3),
they are remotely mapped into the process’s virtual address space. Although we can suggest a
virtual address, there is no guarantee that different machines will actually adhere to our sugges-
tion and instead map the same segments to a different address. For this reason we cannot use
absolute addressing to reference a place in a distributed segment. Instead we must use relative
addressing. This means that we use an offset from the first byte of the segment so both the
node and the proxy can both reliably calculate the its absolute address. An example illustrating
relative vs. absolute addressing is shown in figure 4.3.

Our proxy is implemented on a normal desktop computer, equipped with a Dolphin IXH610
host adapter [22] for performing local DMA/Programmed I/O (PIO) transactions and a 10 gi-
gabit Ethernet card for inter-cluster transfers.

The proxy implements in total three shared memory segments, a buffer segment (discussed
in section 3.3.1), a transaction ring (discussed in section 3.3.2) and a control segment (discussed
in section 3.3.3. These are SISCI segments initialized at startup and that are made available for
any local node to connect to and are all essential for transmitting and receiving data.

### 4.3 Distributed mutexes

So far we have introduced the data structures provided by the proxy, the buffer segment and the
transaction ring, and we have mentioned the use of locking mechanisms to ensure that no two
local or remote threads accesses a critical region concurrently. This section explains how we
implemented a distributed mutual exclusion functionality.

Many algorithms for distributed mutual exclusion has been proposed over the years. Among
the first proposed algorithms for mutual exclusion in a distributed computer network was intro-
duced by Lampart in [30], and later improved upon by others [31–34]. Common to these algo-
// allocate a buffer
void *slab_alloc(volatile struct slab *s) {
    mutex_lock(&s->mutex);
    // get the buffer number of the next free buffer
    int n = s->next_free;
    // update the freelist
    s->next_free = s->freelist[s->next_free];
    // get the memory mapped address of the allocated buffer
    void *ret = slab_start_mem(s) + n * s->seg_size;
    mutex_unlock(&s->mutex);
    return ret;
}

// free a buffer
void slab_free(volatile struct slab *s, void *mem) {
    mutex_lock(&s->mutex);
    // convert the memory mapped address to the buffer number
    int slab_num = slab_get_buffer_num(s, mem);
    // update the freelist
    s->freelist[slab_num] = s->next_free;
    s->next_free = slab_num;
    mutex_unlock(&s->mutex);
}

Figure 4.2: Slab operation

rithms is that none of them relies on common memory, but instead relies on communication via
message exchanges.

We chose to implement our own algorithm that takes advantage of the shared memory ca-
pabilities in SISCI. The basic idea is that the proxy hosts all mutexes in the system, and is
responsible for granting access to one node at the time. Essential to our algorithm is that each
node must memory map the segment in which the mutex he wishes to use is hosted.

A mutex consists of the two fields; request and lock. Whenever a node needs to lock a
mutex, it writes its own node id into the request field, then checks the value of the lock field.
If it is equal to its own node id, it has got the lock. If it is not equal, the process of writing to
the request field and checking the lock field is repeated. When a node has finished executing
the critical section, the mutex is unlocked by writing the integer '0' to the lock field. Figure 4.5
contains code for locking and unlocking a mutex.

The proxy runs a separate "lock thread". This thread has the responsibility of maintaining
all mutexes registered on the system, and administer access to them. The proxy "lock thread"
is responsible for all mutexes registered on the system and keeps a list containing all mutexes
which it will iterate over periodically.

If there is a pending request, and nobody else has the lock it will copy the value of the request
field to the lock field in the same mutex, thus granting access to the node which happened to
update the request field with its node id last.

As can be seen from the code listings, the implementation is fairly simple and straightforward.
An advantage is that it is very fast, because it repeatedly spins on the mutex constantly
checking if it is claimed by another process. This means that the process will not explicitly yield
CPU cycles if the mutex is claimed by someone else. This is faster than mutexes that explicitly
typedef volatile struct spinlock {
    int request;
    int lock;
} spinlock_t;

// Function for locking
void mutex_lock(spinlock_t *lock, int node_id) {
    while (1) {
        lock->request = node_id;
        // We own the lock if this condition is true
        if (lock->lock == node_id)
            break;
        lock->request = 0;
    }
}

// Function for unlocking
void mutex_unlock(spinlock_t *lock, int node_id) {
    while (lock->lock == node_id)
        lock->lock = 0;
}

blocks while another process has the lock. A fast mutex implementation, especially distributed mutexes, is key for performance since our mutexes are used very often.

The downside with repeatedly spinning on the mutex without any blocking in between is that it is consuming a lot of CPU cycles. In a deployment scenario this might be a problem, and a solution involving interrupts might be better. However, since the focus on this thesis is primarily on implementing a fast system we chose to go with the fastest alternative.

4.4 Proxy to proxy communication

In Section 3.3.5 in chapter 3 we described how proxies communicate with each other to transfer data packets. In this section we go into implementation details of how a proxy sends data to a remote proxy.

Our initial implementation, as described here, had a somewhat primitive and poor solution
void lock_thread(void) {
    while (run_lock_thread) {
        for (int i = 0; i < num_mutexes; i++) {
            spinlock_t *current_lock = mutex_list[i];

            // check if anyone is requesting the lock
            if (lock->request == 0)
                continue;

            // check if anyone already has the lock
            if (lock->lock != 0)
                continue;

            // grant access if the mutex is requested and is not already taken
            lock->lock = lock->request;
        }
    }
}

Figure 4.5: Locking and unlocking of distributed mutexes

for detecting new nodes in the transaction ring. The method consisted of periodically checking the length of the ring, which included acquiring the lock and counting all nodes. To prevent the proxy from claiming the lock too often, which would lead to long waiting periods for the endpoint a small sleep period was done between each check. As we later realized this was a poor choice with regard to performance, which is why it was changed in a later version. This is discussed in section 4.6.

The first thing the proxy will do when processing a buffer is to retrieve the remote proxy IP and check if it has established a connection to it earlier. If a connection does not yet exist, the proxy will set up a connection, in the form of a TCP socket. A reference to this connection is stored in a global variable, and is kept alive for as long as both proxies are operational.

When the buffer has been processed and prepared, it is sent to the remote proxy over the TCP connection. At this point the proxy has done its work and there is no longer any reason to keep the transaction ring element. The element is therefore unlinked and freed so that it can be used at a later point. The same also goes for the buffer segment referenced in the transaction ring element; since it will no longer be used, it is freed so that it free for later use. Pseudo code for handling the tx-ring can be seen in figure 4.6.

At this point the packet has been sent, and the receiving proxy will notice that a new packet has arrived and is ready to be handled. In our first implementation we implemented a socket event functionality built on the system call select(2). This can be used to monitor activity on file descriptors, and execute configurable callback functions based on the type of event on the socket. The different types of events are a) accept, which means that there is a new connection on the socket; b) close, which either means that the remote side closed the connection, or an error occurred which lead to the connection being teared down; and c) read, which means that there is data on the socket ready to be received.

The main thread on the proxy runs the main function of the socket event functionality, which is responsible for monitoring and triggering callback functions based on events on the sockets it
```c
void handle_tx_ring(void) {
    // make sure only one process is using the tx-ring at the time
    mutex_lock(tx_ring_lock);

    // iterate over each node in the transaction ring
    for (each element el in tx_ring {
        // remove the element from the ring buffer
        tx_ring_unlink_node(el);

        // get a pointer to the buffer for this tx-ring element
        void *buffer = get_buffer(el);

        // extract the IP address from the buffer
        ip = get_ip(buffer);

        // find or connect to the proxy
        proxy = find_proxy(ip);
        if (proxy == NULL)
            proxy = connect_proxy(ip);

        // replace the IP field with the size of the packet
        set_packet_size(buffer, get_size(el));

        // send it to the proxy
        send_buffer(proxy, buffer, get_size(el));

        // free the tx-ring element and the buffer segment
        slab_free(ring_buffer, el);
        slab_free(data_buffer, buffer);
    }

    // release the tx-ring lock
    mutex_unlock(tx_ring_lock);
}
```

Figure 4.6: Pseudocode showing the process of transmitting proxy buffers

is configured to monitor. For each incoming packet the read callback is executed and the packet is received. First reading the first four bytes of the header, to get the total size of the packet. Then the rest of rest of the packet can be received, based on the size field. The final transfer to the endpoint is performed using Remote Direct Memory Access (RMDA). SISCI RDMA can only be performed between SISCI segments. For this reason, the proxy allocates a proxy buffer segment to use as a destination for the remainder of the packet (the size-field is already received). Pseudocode showing the handling of an incoming packet is shown in figure 4.7

4.5 Optimizations and Improvements – Endpoint

4.5.1 Reliability

Our initial implementation had some missing functionality when it came to reliability. Even though it was able to confirm the delivery it had no functionality to signal any faults that might
```c
void handle_receive(int sd) {
    unsigned size, node_id;
    recv_header(sd, &size, &node_id);
    endpoint_node = get_endpoint(node_id);
    uint8_t *buffer = slab_alloc(buffer_segment);
    receive_packet(sd, buffer, size);
    if (endpoint_node)
        dma_to_endpoint(endpoint_node, buffer, size);
    slab_free(buffer_segment, buffer);
}
```

Figure 4.7: Pseudo code showing how the proxy handles incoming packets

have arisen during the transfer. Because of how TCP network sockets work either side of the transmission also had the ability to detect whether the endpoint should, due to a bug causing a crash, system reboot/shutdown or a network error. TCP is a connection oriented protocol, so if an endpoint failed the other side would get a notification that the connection was broken.

However, it had no way of detecting errors caused by either of the proxies involved in a transfer. If a proxy should fail there is no direct way for the endpoint to know about it and report it to the other endpoint, which is why a timeout on the transfer was introduced. The timeout functionality was implemented on the receiver side of the transfer, with the introduction of a new control channel message; TRANSFER_TIMEOUT. The timeout message is sent to the sender endpoint when the timeout expires. If the receiving endpoint has not received any data in the past X milliseconds the timeout has expired.

### 4.5.2 Interrupts

In section 3.3 we described how the proxy detects new clients and new nodes in the transaction ring. The proxy would repeatedly spin on a shared variable in its control segment to detect the presence of a new endpoint. To detect new nodes in the transaction ring, it would repeatedly take the tx-ring lock and count the number of elements in it. While this approach works, it is very resource intensive, in terms of CPU. It required 100% of an entire core to perform these checks which is a waste or resources. Another disadvantage with this approach is that the proxy would hold the transaction ring lock most of the time without actually getting to process any buffers, this would lead to an artificially high interest in the lock, meaning the endpoint had to wait longer than what really needed which caused the transfers to become slower. This is why we introduced remote interrupts in the proxy.

As described in chapter 2 interrupts is a mechanism in SISCI which enables a machine to trigger an event in a remote machine. The proxy takes advantage of this by defining two interrupts, CLIENT_EVENT and TX_RING_EVENT, each with its own purpose.

In SISCI interrupts there is a function SCIWaitForInterrupt that blocks the calling thread until the interrupt is triggered. However, there is also an option which allows interrupts to be handled asynchronously. The CLIENT_EVENT interrupt is configured to call a callback function in the proxy whenever the interrupt is triggered. The callback function associate with CLIENT_EVENT handles the connection and disconnection of a new node.
The TX_RING_EVENT is the second interrupt defined by the proxy. When triggered it will cause a callback function to be executed in the proxy which will iterate, process and send every element in the transaction ring. When there are no more elements left in the transaction ring there is nothing more to be processed, and the function will return and the thread will block. This interrupt is the most important one with regard to improving transfer efficiency and CPU usage, because it allows the proxy to only take the tx-ring lock when there is something to process, thus the endpoint will not have to wait as long as with our first solution.

By introducing these two interrupts we completely freed up one CPU core from doing manual checking of the presence of new client and nodes in the tx-ring. Now the endpoint triggers these interrupts only when the proxy needs to take action.

4.5.3 Vectored DMA

During transmission of data to a remote endpoint the data is sent through proxies. The first version of the endpoint used regular DMA to transfer local data into a proxy buffer, and while DMA is fast, there is still some overhead associated with initializing the DMA engine located on the PCIe card. Vectored I/O is in essence a way to perform multiple I/O operations with a single function call. The most important reason for doing so is for the sake of efficiency but also convenience. One vectored read/write call can replace multiple regular reads/writes, thus saving time involved in context switching, and, in the context of SISCI save time with setting up the DMA engine.

The SISCI API provides support for vectored DMA transfers natively with the function SCIDmaStartTransferVec. The endpoint added support for this to greatly reduce the number of DMA function calls needed to perform large data transfers. To use vectored DMA transfers, an array of DMA descriptors is passed to the vectored DMA function, so instead of a loop with a DMA call to transfer everything, we now have a loop with array initialization and a lot fewer DMA calls. This leaves more time to allocate, initialize and insert buffer descriptors into the transaction ring of the proxy, before triggering the tx-ring interrupt.

While it is a good idea to add vectored DMA transfers from the endpoint to the proxy, we did not add this to the proxy. When the proxy receives data from a remote proxy it DMAs it into the data segment of the endpoint, so it is possible also here to do vectored DMA transfers, but, as the proxy has no way of knowing how much data it will receive in the future, and if that data is going to the same endpoint, it becomes a matter of practicality. A possible solution is to wait for X incoming packets to arrive, then initialize and start a vectored DMA transfer. But, because we do not how many packets are going to be received, we would have to add a timeout on the receiving socket, so that the DMA transfer could be started and thus completing the total end-to-end transfer. This timeout is the core of the problem with such a solution; every time this timeout expires the total latency and throughput as experienced by the sender and receiver endpoints gets poorer.

4.5.4 Smarter TX-ring interrupt

In section 4.5.2 we introduced interrupts to handle new clients and processing the transaction ring in a less CPU intensive way. While it reduced CPU resource usage, it had some increased the response-time to start processing the transaction ring. This is because waking up a sleeping thread to execute the callback function associated with the triggered interrupt takes a longer time
than checking the size of the ring or checking the state of a variable in a quick repeated manner. In order to reduce the reaction time of the proxy, we implemented a hybrid polling/blocking functionality.

The way it works is that we added a new variable in the proxy control buffer called `ring_buf_wait`. This variable can have two different values: `RING_WAIT` or `RING_GO`. The value of this variable is controlled only by the endpoint. Before starting any DMA transfers the endpoint will set `ring_buf_wait` to `RING_WAIT` then triggering the transaction ring interrupt, causing the proxy to execute the function used to process the transaction ring. However, it will not claim the tx-ring lock and start processing the buffer until `ring_buf_wait` contains the value `RING_GO`. It will spin on this variable, without blocking or yielding voluntarily.

With the introduction of the hybrid polling/blocking mechanism, all the client needs to do to signal the proxy to start processing the transaction ring is to set the variable `ring_buf_wait` to `RING_GO`. When everything is transferred, the endpoint will wait for confirmation that the remote transfer was completed from the remote endpoint.

To prevent the proxy from hogging too much CPU resources while spinning on `ring_buf_wait`, it will return from the interrupt callback and start blocking again after five seconds without any buffers to process.

### 4.5.5 Reducing mutex usage

Critical code regions where only one process can execute at the time has the potential to decrease performance by a huge factor. In an ideal scenario we would have no critical code sections, and no waiting for turns, but this is not possible in our design. The one critical section used most is the access to the transaction ring. Endpoints will insert nodes into it, and proxies will iterate and unlink nodes from it, so in an effort to avoid processes to wait for access as much as possible we took action to reduce the time these locks were held.

Measuring the time spent holding the transaction ring lock, we determined that the proxy had the most potential for reducing the time spent holding the lock. When the proxy forwards buffers transferred by endpoints it will take the transaction ring lock, step through the ring node by node, send buffers then unlink the nodes in it. First when all nodes has been processed and all buffers has been forwarded to a remote proxy is the lock released. In an effort to reduce the time the mutex was held, the first thing we tried was to offload some of the work to a different thread. Thread creation however, involves much work and is relatively expensive, to avoid the overhead of constantly creating and terminating new threads we implemented a thread pool. The thread pool was initialized at startup with a fixed amount of threads in it. In this thread pool, jobs had to be posted for the threads to start performing any work. The jobs we posted to the pool was the step consisting of sending buffers to a remote proxy and freeing them after use. This would in theory lead to less work for the main thread to perform while processing the transaction ring, and thus being able to release the lock quicker. As we find out however, this approach did not lead to any significant performance increase. While the actual processing of the ring was slightly faster we did not see any gains in throughput because the buffers themselves did not get freed quick enough, so the endpoint node would have to wait until it was able to allocate buffers to start transferring more data. When testing for latency we did see a tiny reduction in latency time however, but not enough to call this optimization to be a success.

In another more successful attempt we realized that the critical part was the removal and insertion of nodes in the same ring structure. Being able to remove and insert nodes in a dynamic
Long-range RDMA over PCIe

fashion is a great advantage, and one we made use of here. Instead of holding the lock while processing elements only to unlink nodes one at the time, we tried removing all nodes in the transaction ring at once. Put in other words, we split the transaction ring into two separate rings, connecting the tail to the next node after the head node. This allowed the lock to be held only while performing the two simple operations to split the ring; linking tail->next with head->next and head->next->prev with head->prev. As illustrated in figure 4.8, this will effectively leave us with two rings, one containing every node, and the one now empty transaction ring. This way the endpoint can get access to the transaction ring faster, and the proxy can iterate the new ring containing all the elements previously contained in the transaction ring. This was more effective than our first attempt and there was no need for offloading any work to a secondary thread.

![Figure 4.8: Transaction ring split into two separate rings](image)

4.5.6 Communication channel protocol optimization

The communication channel protocol, as described in section 3.2.3, is used to synchronize transfers and detect errors that might occur during a transfer. After doing a review of the protocol, we realized there was much room for improvement. We measured the time spent communicating back and forth both before and after the actual transfer has begun, and found out that a major portion of time was spent here. Looking back at figure 3.3, we see an example of a successful transaction and that the total amount of messages going back and forth adds up to three sends for endpoint A (the sender), and three sends for endpoint B (the receiver).

The first two HELLO messages were intended to be a way to be certain that both nodes were still running and functional, but did not really contain any information useful to the potential transfer which was about to happen. As the control channel uses TCP the HELLO messages are in reality redundant because if any of the two endpoints have crashed, or something else
happened which caused one of the endpoints to not respond it would become obvious after the next control message. Because of this fact we removed the **HELLO** messages from the protocol, thus saving one Round Trip Time (RTT) of delay.

The next two messages are the **DMA_START** and **DMA_START_OK** and are the two most important messages, as they contain a request along with transfer information, and an accept or decline response. Neither of these messages can be stripped away.

Similarly, the **DMA_COMPLETED** is also essential because it informs the sender when and if the transfer was completed or not. The very last message however is simply an **OK** message informing the receiving node that it got the **DMA_COMPLETED** message and thus concluding the transfer. As the final **OK** message does not carry any useful information other than being the message finishing the transfer, we removed it, and thus making **DMA_COMPLETED** the message concluding the transfer.

After stripping down the communication protocol we reduced the cost of synchronizing a remote data transfer with a total of 1.5 RTTs, leaving us with only one send for the sending endpoint and two sends for the receiving endpoint. An example of a successful transfer using the altered communication protocol is illustrated in figure 4.9. The gray arrows in the example shows the messages which was stripped away.

### 4.5.7 Control channel optimizations

In the previous section we presented the optimized protocol used on the communication protocol. Now we will present the steps we took to optimize the channel itself to better fit the characteristics of the traffic on it.

The control channel is not subject to heavy traffic and does not demand high throughput at all. Every message on the channel is exactly 9 bytes and three messages are transmitted each time a remote data transaction is performed. Much more critical than throughput is latency. This is because a transmitting endpoint will not start transmitting data to its proxy for further forwarding to the remote cluster, so the faster the transmitter can get the “go-ahead” the sooner the transaction phase can begin.

Our Network Interface Card (NIC) was firstly configured for high throughput and low latency. The first thing we did with regard to reduce latency was to disable interrupt coalescing on the NIC. Interrupt coalescing is a feature implemented by many devices supports and is designed to reduce load caused by hardware interrupts triggered by the NIC. When the NIC receives a packet from the network, it will trigger a hardware interrupt to inform the operating system kernel of new incoming packets. Interrupt coalescing causes the NIC to wait for a short while either until a certain amount of data has been received, or until a timeout has expired, before triggering the interrupt that causes the kernel to retrieve the data. By disabling interrupt coalescing the NIC will trigger the interrupt when it receives data instantly, without waiting for more data to arrive. This approach has been known to shave off a few microseconds of the latency [35].

The steps taken which reduced latency the most however was to tune the TCP sockets to best work with the traffic characteristics of the control channel. Linux lets us TCP sockets to be tuned from user space by utilizing the system call **setsockopt(2)**. By default, TCP in Linux uses the Nagle algorithm [36]. It is used to concatenate multiple smaller TCP buffer messages into fewer and bigger packets. It is designed to increase the efficiency of network usage by decreasing the number of packets sent, or as described in the manual sections for TCP in Linux:
"data is buffered until there is a sufficient amount to send out, thereby avoiding the frequent sending of small packets, which results in poor utilization of the network" [37]. For a channel that is highly dependent on low latency we do not want the behavior introduced by Nagles algorithm, it can be disabled on a per socket basis by setting the socket option `TCP_NODELAY`.

Another TCP socket option that we enabled is the `TCP_QUICKACK` option. This socket option is used to prevent TCP Acknowledgments (ACKs) from being delayed. With this option enabled, ACKs are instead sent immediately. This option is not permanent. According to the Linux manual pages, "subsequent operation of the TCP protocol will once again enter/leave quickack mode depending on internal protocol processing and factors such as delayed Ack timeouts occurring and data transfer." [37], for this reason the QUICKACK option must be set after every receive call on the control channel.
4.5.8 Endpoint transfer optimizations

Our first version had a somewhat poor method of transferring data and signaling the proxy to start forwarding buffers. The endpoint would allocate as many buffers as it needed to complete a transfer (potentially all available buffers at the time), do the transfer, then tell the proxy to start forwarding the buffers it had filled. When doing small transfers, e.g., 2-3 times as big as one buffer cache, this approach is fine because the time spent transferring, adding nodes into the transaction ring, and signaling the proxy takes such a short amount of time. For larger transfers, however, it becomes an issue. It makes little sense that the proxy should have to wait until the endpoint has transferred every single byte of its total transfer, before the proxy starts to forward the buffers. For this reason we improved the endpoint transfer mechanism. Instead of transferring everything at once the total transfer is divided into smaller chunks. After a chunk is transferred to the proxy, nodes for the tx-ring is allocated, initialized and inserted into the tx-ring, then the proxy is signaled to start processing the transaction ring.

The advantage with doing this is that the proxy and endpoint can work on the end-to-end transfer simultaneously, thus achieving higher throughput and latency. Another advantage is that buffer caches are freed much sooner so the chances of having to wait for buffers to be freed are greatly reduced. The disadvantage with this mechanism is that the number of mutex accesses are increased both for the proxy and endpoint as buffers are allocated, freed, inserted and removed from the transaction ring. Number of mutex accesses are increased both for the proxy and endpoint as buffers are allocated, freed, inserted and removed from the transaction ring.

4.5.9 Optimistic transfers

As we have already learned, the endpoint will not start the actual data transfer until both the endpoint nodes have agreed on the transfer. With this in mind we attempted a method referred to as optimistic transfers. The idea here is that more times than not, the remote transfer request is approved, so by assuming this is the case the endpoint can start the transaction before it actually was approved. This would, in theory, save time on the total transfer because by the time the sender got the request approved, the initial batch of data had already been transferred to the proxy and all that is left to is adding nodes to the transaction ring and signaling the proxy to start forwarding the buffers. If the transfer should not be approved, the buffers already filled had to be freed by the endpoint before aborting the process. Apart from the fact that it did not yield the expected results, the great disadvantage is that buffers are potentially allocated and filled with data only to free them again later because the transfer request did not get approved. The results we expected with optimistic transfers was that the end-to-end latency would decrease, but our tests showed no noticeable improvements, but no reduction in performance either. The results weighed against the negative side effects of this made us achieve this as a failure and it was not included in the final version.

4.5.10 Programmed I/O for small transfers

So far the endpoint to proxy transmissions was performed with DMA regardless of the size of the transfer size. There is an initial overhead associated with DMA. The CPU must first initialize the DMA engine before the DMA can start, so for smaller message sizes, PIO is
faster. For this reason we implemented functionality to use PIO as a choice for endpoint-to-proxy transmissions. It works the exact same way the DMA method does, in the sense that the allocation of buffer caches, tx-ring nodes, end-to-end control channel protocol is done in the exact same manner.

By using this method we saw great improvements in end-to-end latency which was greatly reduced. As expected the improvements in latency is especially noticeable with smaller transfers, but with larger sizes we did not see any improvements. This was also expected and can be explained by the notion that DMA catches up with PIO as the message sizes increases.

With these observations in mind, it would make sense to implement a hybrid PIO/DMA mechanism that automatically switched between PIO and DMA depending on how big the transfer size is. But as we seek to provide an API extension to the already existing SISCI API, it makes more sense to implement PIO and DMA transmission functionality as two separate features.

4.5.11 Parallelization

We also looked into the notion of paralellizing parts of the transfer process. There are not many places this is possible, but one possible place is the endpoint to proxy transfers. A possible solution is to execute the allocation of buffer caches, transfer of smaller individual chunks of the data and linking nodes into the tx-ring in parallel. This does not make much sense however, simply because the bottleneck does not lie in this step in the flow. The endpoint nodes transfers data to the proxy at a much higher rate then what the proxy can manage to forward the data out on the TCP connection to the remote proxy.

If anywhere, the proxy would be the place to add parallelism, yet there are not many places this could be done. A possible solution is to have multiple transaction rings, and allocate one proxy thread for each transaction ring to monitor. This way, the endpoint could switch between the number of different transaction rings, each time it adds nodes to it. This has the potential to reduce possible waiting periods caused by getting access to the critical regions of the tx-rings. But even this would not improve anything, as it also introduces more complexity to the sending mechanism on the proxy. Since each transaction ring now holds data which is going to the same destination cluster, yet another synchronization mechanism on the proxy-to-proxy channel would need to be added to make sure the buffers are transmitted in the correct order, so that they will arrive at the destination endpoints buffer segment in the same way the sender endpoint intended.

A possibility however is to allocate one transaction ring for each connection an endpoint has to a remote segment. This way the proxy could spawn multiple threads to monitor a subset of the total amount of transaction rings, and there would be no need for any synchronization to make sure the buffers are sent in the correct order. While this could work, it is a poor solution because of the scalability problem it introduces. If there are many local endpoint each with many connections to remote segments, the number of transaction rings hosted on the server would become too many for the proxy to be able to handle with a sensible amount of threads. Another problem with this approach, is that it would require a lot of space to host all those transaction rings. All that space would be better to use for the buffer segment.

For these reasons we have not parallelized any parts of the transfer process, neither in the endpoint or the proxy.

In contrast to how the proxy connected to the local endpoint in the first version, the new
Figure 4.10: The buffer cache layout after being processed by the proxy and with the added segment ID field

version did not connect to the endpoint when the endpoint connected to the proxy. It still maintains the same data structure over local endpoints however, but it will connect to the segment in question the first time only when it receives a packet from a remote proxy containing the ID of the destination segment. Since the proxy now needs to keep track of several segments for each endpoint, the endpoint data structure now contains a list structure with the necessary SISCI descriptors for all its segments.

The final change made was done on the endpoints control segment. The endpoints control segment contained an integer that held the current offset into its data segment, which was used to keep track of how many bytes had been transferred. It was continuously updated by the proxy, and monitored by the endpoint. Because the endpoint now can share many segments, we need one of these offset-fields for each shared segment. Because the proxy also needs to update this field, it must be able to find the correct offset field. To facilitate this, an array of nodes containing the segment ID and segment offset was defined in the endpoints control segment.

4.5.12 Transparent handling of receiving data

Our mechanism that waits for a remote DMA transfer can be seen as event on the receiver side. Functionality so far requires that the receiving node must explicitly wait for and handle a remote transfer. The SISCI API works a little different in that it does not have to explicitly wait for a remote transfer in order for the transmission to take place. To better fit the framework of
the SISCI API, we added a mechanism that lets the reception of a remote transfer be handled transparently by the system. The control channel synchronization both before and after the transfer starts is performed in the exact same way as before. The difference being that a separate thread handles everything for the user. This functionality was implemented with the help of the socket event mechanism we implemented. The socket event main loop is started in a separate thread, the function responsible for handling the reception of the remote transfer is registered as a callback function, and when a read-event is triggered on a socket, the socket event calls the callback function in its own separate thread, thus taking care of everything.

4.6 Optimizations and Improvements – Proxy

Now that we have an understanding of the components and overall flow we shift the focus over to the improving the system with regard to both functionality and performance. We will explain the improvements in functionality and the steps taken to optimize bandwidth, latency and CPU usage of our initial implementation. The first version had some missing functionality with regard to the requirements we set; E.g reliability, support for multiple local endpoints and multiple remote cluster connections.

4.6.1 Multiple remote cluster connections

The initial proxy version only had support for one connection to a remote cluster. While this is sufficient for a proof-of-concept used for demonstrating and measuring performance, it is not sufficient in a real-world scenario. For this reason, support for multiple remote cluster connections were added. Remembering that the proxy gets instructions from the endpoints about which remote clusters it should connect to from the header field in each buffer it processes, it did not turn out to be a complex task to add support for this.

The first version stored the remote socket connection in globally accessible variable making retrieval very fast. A goal with adding support for multiple remote connections were to not introduce a significant amount of overhead with searching and retrieving remote cluster connections. The solution we went with turned out to be sufficient for our purpose and did not introduce any significant overhead. We implemented a linked list structure were each node contained the IP address of the remote cluster along with the socket descriptor. Finding the correct connection was then done by iterating the list and comparing the IP addresses, making the worst case scenario for retrieval $O(n)$. The linked list data structure also have the added benefit of easily swapping and rearranging the nodes. This is something we took advantage of by implementing a Least Recently Used algorithm were the node which last was retrieved was put at the head of the list, this makes the retrieval of the node quicker the next time.

4.6.2 Multiple local endpoints

In similarity with multiple remote cluster connections, the first version only had support for one local endpoint. The proxy needs to keep track of the local endpoints to be able to transfer data to them, so it needs a data structure to store them. We chose to use a similar scheme as with the remote cluster connections; a linked list. Each node contains the endpoints data segment, control segment and node id. When the proxy receives a packet from a remote proxy,
it contains a header field with the ID of the destination node. This header field is used when searching through the linked list to find the correct local node.

With the introduction of multiple endpoints, it is however not enough to store multiple endpoint nodes in a linked list. From section 3.3.4 we remember that the endpoint connects to the proxy by initializing information about itself in the proxy control segment then triggering an interrupt to notify the proxy (4.5.2). There is a potential for a race condition here. Should two or more endpoints try to do this simultaneously data will get corrupted. To solve this we introduced a new distributed synchronization primitive, called connect-lock, in the proxy control segment. When an endpoint connects to the proxy, it must first take the new lock, initialize the information about itself and then set a the connect field in the control segment of the proxy to CLIENT_CONNECT. To prevent other nodes from overwriting the information written by the previous endpoint before the proxy got a chance to handle the new connection, the connect-lock is not released until the proxy sets the connect field to CLIENT_CONNECT_OK.

### 4.6.3 Buffer segment optimizations

Earlier in this chapter we introduced the buffer segment. It is a data structure hosted by the proxy and shared with nodes in a local cluster. It is used as a temporary storage for data that the proxy will forward to a remote cluster. Endpoints will allocate fixed sized slabs/buffer caches, and use as a destination for data transfers. In our first implementation the buffer caches were 1024 bytes in size and with a total of 2048 buffers. As it would turn out, these two properties of the buffer segment has a big impact of how much data the proxy can manage to send per time unit. After some experimentation with different buffer sizes and number of buffers, we realized that the numbers we initially set were not optimal with neither latency or throughput in mind. The packets forwarded by the proxy are of a fixed size, corresponding to the size of the buffers in the buffer segment, and to achieve the highest possible throughput we want to send as much data as possible at the time. By increasing the buffer sizes in the buffer segment the proxy can send more data at the time, and less time is spent in processing buffers. In addition to the buffer sizes, the number of buffers available for allocation also has an impact. The more buffers that are available the less time is spent waiting for buffers to be freed, and the endpoints and the proxy can work parallely. Ideally we would have a lot of huge buffers, but practical limitations taken into consideration there is a limit on how big and how many buffers that can be used. There is a trade off; we could use very big buffers, but that would restrict the number of buffers we could use. We could also use small buffer sizes and many number of buffers, but this did not yield the throughput we were looking to achieve. For best throughput we ended up with a balance between the two factors, 512 KiB per buffer and a total of 100 buffers, resulting in a total of 512 KiB * 100 = 50 MiB + 420 bytes for slab allocation metadata.

Depending on the application however, a different configuration might be better suited. If an application using our system is not utilizing the full size of the buffers, a lot of precious space is wasted that could have been used either by other nodes or in subsequent transfers. As the goal with our prototype is to show that we can achieve high performance with respect to throughput and latency and with no particular application in mind, we chose not to concern ourselves about implementing a mechanism for automatically tuning buffer sizes depending on usage patterns, or an alternative more complex buffer allocation algorithm such as a distributed version of dmalloc [38] or ptmalloc [39].
4.6.4 Slab allocation

We also made an addition to the slab allocator, designed to reduce the number of times a mutex is taken. As described earlier, the endpoint allocates proxy buffer segments using the slab allocator, to use as a destination for DMA transfers. For large transfers the endpoint often ends up allocating multiple buffers, which ultimately leads to multiple slab allocations and mutex locking/unlocking. An advantage is that the endpoint can calculate how many buffers it will need in total to complete a transfer, so instead of allocating N buffer with equally many slab function calls, we added a method to allocate multiple buffers at the time, while only taking the slab lock that one time.

After the endpoint calculated the number of buffers it would need and synchronized with the remote endpoint, it would allocate all the buffers and transaction ring nodes it needed and then go on to start transferring and adding nodes to the transaction ring. By dealing with the slab lock only one time during a transaction phase saves time for the endpoint, because it does not need to do as many accesses to the proxy slab segment, but also saves time for the proxy as it increases the odds for not having to wait for the lock to free the segments it has sent.

4.6.5 Added support for connecting to individual segments

Our initial version had a solution where an endpoint made a connection to a remote endpoint via the TCP control channel to connect to that endpoints shared data segment. When connecting, the connecting node would get a message containing that nodes ID and the IP address of its proxy. When transferring data to the endpoint the data transferred would end up in a default data segment. There was no functionality implemented to connect to a chosen remote segment. As this is something that should exist and would better fit the current functionality provided by the SISCI API, we added support for this.

To enable this functionality we had to make some changes to how the proxy connects to the endpoints segments, the way the endpoint connects to the endpoint with the control channel, and the header appended to each buffer cache processed by the proxy.

To connect to a certain segment at a remote endpoint, it is now assumed that the connecting node knows the ID of the segment to which it is going to connect. The data structure initialized upon a successful connection now also contains the segment ID for that connection. That means that one control channel is the control channel for one segment and one only. If an endpoint wishes to connect to more segments, it has to establish another control channel as well.

The segment ID stored in the control channel data structure is then used again when the endpoint initializes the headers for the buffer caches it transfers through the proxy. Figure 3.7 defines the header fields initialized by the endpoint in the initial version; Two four byte fields, containing the remote proxy IP and the remote endpoint node ID. These buffer caches needs to contain the segment ID to the segment where the buffer is destined, so we could either add a new header field, or we could split the node ID field in two, and use the other half for segment ID. The choice fell upon the latter option because node IDs never needs more than 16 bits anyway, the remaining 16 bits are enough to store a segment ID, plus we do not have to spend another 16 bits of the total buffer size for headers. The packet format after the new change as illustrated in figure 4.10 is how the packet looks like when it is sent to the remote proxy.
4.6.6 Proxy to proxy channel optimizations

The proxy to proxy channel is used to transfer data to another proxy on behalf of local endpoints. It is a socket channel using TCP and is the only channel that actually carries data, and can be subject to very huge amounts of data. Therefore it is important to tune this channel to best fit our needs, which is the highest throughput possible.

The main tuning of the TCP socket that we did was the TCP buffer sizes. To achieve the optimal bandwidth possible, the buffers should be set to an optimal size. The optimal buffer size can be calculated like so: \[ BDP = 2 \times \text{bandwidth} \times \text{delay} \] or alternatively \[ BDP = \frac{\text{bandwidth}}{\text{RTT}} \]. [40] argues that the default TCP send and receive buffer sizes should be set to the optimal buffer size (BDP) to achieve the best performance. However, Linux kernels, starting from 2.6.17 implements a mechanism called autotuning. Autotuning automatically adjusts socket buffer sizes and TCP window size as needed to balance TCP performance and memory usage in an optimal way. Since autotuning was introduced, there is generally no need for manually tuning the TCP read and write buffer sizes, some sources does not even recommend doing so [41]. We experimented with setting different buffer sizes, both BDP and bigger, and came to the conclusion that it did not have any improvements in performance, rather than getting any improvement in throughput we saw a decrease in throughput. Manually adjusting the buffer sizes will disable autotuning [41]. This was the reason for the decrease in performance we saw.

In our testing setup we used two 10 Gigabit Ethernet cards for the data channel. While the standard Maximum Transmission Unit (MTU) is normally 1500 bytes, the 10GE cards we used supports jumbo frames. Jumbo frames can normally carry up to 9000 bytes, but some devices can support up to 16128 bytes per frame [42]. We set the MTU to the maximum value of 9000 bytes using the following command: `ifconfig ethX mtu 9000`.

We also performed tests with different TCP congestion control algorithms. A congestion control algorithm is an algorithm used in TCP to achieve high performance and to avoid congestion collapse. It is a mechanism that actively strives to keep the data flow below a rate that would trigger a collapse. In Linux, the default congestion control algorithm is cubic, but other algorithms are available and can be changed on a per socket basis using `setsockopt(2)`, or system wide by overwriting `/proc/sys/net/ipv4/tcp_congestion_control`.

4.7 API for long-range RDMA

In chapter 2 we gave an introduction to the SISCI API, with its essential functionality and workings. In this section we will present the functionality intended to serve as an extension to the existing SISCI API that can be used to program PCI Express enabled devices produced by Dolphin. The functionality we present here is all for doing long range inter-cluster data transfers and is used on the client/endpoint. They make use of the proxy to relay data to the remote cluster, and uses the end-to-end channel for synchronizing the transfers.

4.7.1 Data structures

sci_ctrl_channel_t

`sci_ctrl_channel_t` is a data structure needed to use the rest of the functionality. As the name suggests it is a handle to the control channel of a remote segment which is used to
synchronize data transfers. One handle is created per remote segment an endpoint is connected to.

**sci_proxy_t**

sci_proxy_t is a handle to the proxy in an endpoints local cluster. Only one of these handles can exist per node. This is because a cluster contains only one proxy, and multiple connections to its segments is not needed, no matter how many remote endpoint connections the endpoint has established. All remote transfers, regardless of its destination is done through the one sci_proxy_t handle. This handle is used together with the data transfer functions, and is initialized upon startup of the endpoint node.

### 4.7.2 Functions

**SCIRemoteInitialize**

SCIRemoteInitialize is similar to SCIInitialize. It is the main initialization function for the remote functionality, and it must be called before any other function related to remote transfers can be used. The function takes two arguments; The IP address and Dolphin node ID of the proxy. SCIRemoteInitialize is responsible for connecting to and mapping the necessary data structures hosted by the proxy as well as interrupts and DMA queues for performing DMA transfers to the buffer segment also hosted on the proxy. The data structure returned by this function is of the type sci_proxy_t and contains a reference to all the proxy-related handles initialized by this function, and are used together with the other functions related to remote transfers.

**SCIRemoteTerminate**

SCIRemoteTerminate is the inverse function of SCIRemoteInitialize in the sense that it is responsible for disconnecting and freeing resources initialized by the initialization function. This function must be called before the process is about to terminate. Not only is this important for the endpoint, but also for the proxy, as the endpoint will notify the proxy of its disconnection which allows the proxy to unmap and disconnect from any segments to that endpoint, thus saving resources.

**SCIConnectToRemoteSegment**

SCIConnectToRemoteSegment is the function used when establishing a connection to a segment hosted in an endpoint located in a remote cluster. It returns a data structure of the type sci_ctrl_channel_t which acts as a handle to the remote segment. The function establishes the control channel associated with the segment and receives the proxy IP of the remote cluster as well as the Dolphin node ID of the endpoint. The data structure returned is used internally in the functions related to receiving and retransmitting data to and from a node in a remote cluster.
SCI Close Remote Segment

 SCI Close Remote Segment is the inverse function of SCI Connect To Remote Segment. Its responsibility is to disconnect from a segment hosted by an endpoint located in a remote cluster, this involves tearing down the control channel and freeing other resources allocated by the connect function.

SCI Wait For Remote DMA

 SCI Wait For Remote DMA is the function that must be called when receiving data from a remote endpoint. It is responsible for synchronizing the transfer. If the request is OK, it will return only when every byte of the transfer has arrived, or if an error occurred. If the proposed transfer should not be acceptable, it may refuse the transfer request. When the transfer is completed it notifies the transmitter endpoint through the control channel, then the function returns.

 This function can either be called explicitly, or it can be handled transparently in the background by a separate thread in combination with a socket event mechanism like the one we implemented, if the program design allows for receiving data transparently.

SCI Start Remote DMA Transfer

 SCI Start Remote DMA Transfer is the first of three functions associated with transferring data to an endpoint located in a remote cluster. The data being transferred is transmitted via the local proxy. This means that the data is first transmitted to the proxy via DMA before the data is forwarded to its destination.

 Before the actual data transmission is started, a transfer request takes place using the control channel. the request may be accepted or it may be rejected, in which case the transmission will not start.

 The function blocks on I/O and does not return until the remote endpoint has received every byte of the transfer, or if an error occurred.

SCI Start Remote DMA Transfer Vec

 SCI Start Remote DMA Transfer Vec is the second of the remote data transfer functions. The data being transmitted is transferred via the local proxy. In contrast to SCI Start Remote DMA Transfer this function uses vectored DMA on the initial transfer to the local proxy.

 Before the actual data transmission is started, a transfer request takes place using the control channel. the request may be accepted or it may be rejected, in which case the transmission will not start.

 The function blocks on I/O and does not return until the remote endpoint has received every byte of the transfer, or if an error occurred.

SCI Remote Mem Cpy

 SCI Remote Mem Cpy is the last function that can be used to perform data transfers to a segment hosted by an endpoint in a remote cluster. It works in a similar fashion to the two other data transfer functions, but instead of using DMA or vectored DMA to transfer data to the local proxy, it uses PIO. As PIO is usually faster for small message sizes it may provide a better alternative to the two other functions, depending on the needs of the application.
Before the actual data transmission is started, a transfer request takes place using the control channel. The request may be accepted or it may be rejected, in which case the transmission will not start.

The function blocks on I/O and does not return until the remote endpoint has received every byte of the transfer, or if an error occurred.

4.8 Summary

In this chapter we have presented implementation aspects of our prototype system for performing inter-cluster data transmissions. We also presented optimization steps, for both proxy and endpoint, to improve performance.

Finally we presented the proposed functionality for inter-cluster data transfers as an extension to the SISCI API.

In the next chapter we will present the performance results, along with an evaluation, of tests we performed on the system.
Chapter 5

Evaluation and Results

In chapter 3, we presented how the prototype performs inter-cluster data transmissions. We first presented what our initial version was like, then went on to discuss the improvements and discussion, and finally we presented a proposed extension to the SISCI API for performing inter-cluster data transfers.

In this chapter we will look into results obtained by performance tests of our system. To best evaluate the performance of our prototype we performed several different benchmarks, so that we can identify where in the flow path the main bottlenecks are located.

5.1 Testing environment

5.1.1 Setup

To perform our benchmark tests we used a setup with two machines with which we emulated two separate remote clusters in a lab environment. Each virtual cluster consisted of one proxy and one endpoint instance. In figure 5.1 the setup is illustrated. Each virtual cluster hosts the endpoint and the proxy on two separate machines. The reason for using the two machines for each cluster, instead of using one machine for each virtual cluster, is to avoid any shortcuts to be taken internally. For example loopback devices could be utilized instead of the actual NIC or PCI Express adapter. The figure also highlights the steps a transfer between the endpoint in cluster 1 and the endpoint in cluster 2. The red arrow is the first step, where the endpoint in cluster 1 transfers data to its proxy, the green arrow is the second step, where the proxy transfers the data to the remote proxy in cluster 2, and lastly the blue arrow, where the remote proxy transfers the data to the destination endpoint.

Both machines had a Dolphin IXH610 PCI Express host adapter connected back to back, and a 10-Gigabit Intel Network card, also connected back to back using a Cat6 Ethernet cable. The specifications for the machines we used are listed in table 5.1.

Both machines ran Ubuntu 12.04.5 LTS, with Linux kernel version 3.5.

The Dolphin IXH610 card we used is a second generation PCI Express card that facilitates high performance interconnects. Using 8 PCI Express lanes its theoretical bandwidth is 40 Gbits/s. The card can operate in either transparent or non-transparent bridging mode. We use these cards to share and map memory in remote machines, which is why we used the non-transparent bridging mode.
5.1.2 Tests

To evaluate the performance of inter-cluster transfers using our prototype, we have conducted a series of different benchmarks so that we can understand what the absolute maximum performance we can expect is and how it compares against the actual results we got. This is also useful for illustrating where in the flow of execution the main bottleneck is. All the timestamps collected when benchmarking our system were collected using the function `clock_gettime`. This is a function that retries the time from multiple sources on the host machine. The time is retrieved in a data structure containing the current Unix timestamp and the number of nanoseconds since the last second.

<table>
<thead>
<tr>
<th>Machine</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel i7-2600 @ 3.40GHz</td>
<td>Intel i7-2600 @ 3.40GHz</td>
</tr>
<tr>
<td>Motherboard</td>
<td>Alienware 046MHW</td>
<td>Alienware 046MHW</td>
</tr>
<tr>
<td>RAM</td>
<td>8 GB</td>
<td>8 GB</td>
</tr>
<tr>
<td>Networking card</td>
<td>Intel X540-AT2</td>
<td>Intel X540-AT2</td>
</tr>
<tr>
<td>PCIe card</td>
<td>Dolphin IXH610</td>
<td>Dolphin IXH610</td>
</tr>
</tbody>
</table>

Table 5.1: Specifications of the machines used in the testing environment
Benchmarks

To get a good evaluation of the overall performance of our system, we performed bandwidth benchmarks on the complete system, meaning from one endpoint, through its proxy, to the remote proxy, then to the destination endpoint. To get an indicator of where in the system the main bottleneck lies we also performed benchmarks on individual components on the system. We benchmarked the bandwidth achieved on the endpoint-to-proxy transfers, proxy-to-proxy transfers, and proxy-to-endpoint. For all these cases we benchmarked all three of the transfer functions presented in section 4.7 chapter 3.

We also benchmarked the DMA bandwidth for the PCI Express equipment we used using a tool provided with the Dolphin software pack called dma_bench. The tool uses the SISCI API internally, so it is a good benchmark to use for comparison. The Dolphin software pack also contains a tool called scibench2 which was used to perform PIO write performance.

Another important thing we benchmarked is the bandwidth of the Ethernet link between the proxies. For this we used a standard tool widely available for most Linux distributions netperf [43].

Another important factor to determine performance is latency. As with the bandwidth measurements we performed latency benchmarks on end-to-end transfers using the API extensions presented in chapter 3. To test latency on the Ethernet link we used a program called latency_bench which is a tool included in the Dolphin software pack that uses TCP or UDP to measure latency on regular sockets or Supersockets (chapter 2 page 13).

5.2 Bandwidth benchmarks

Here we will present the performance benchmarks for a full transfer between a local and a remote endpoint, in addition to benchmarks of sub-components of our prototype. We have benchmarked all three transfer methods, presented in chapter 3 with different buffer sizes. We will compare these results against raw SISCI DMA performance of the Dolphin equipment as well as performance of the Ethernet link.

Endpoint to proxy DMA

In order to identify in which component the bottleneck in an end to end transfer lies, we present bandwidth performance results for the copy process from the endpoint to the proxy. This step compares bandwidth using parts of the SCIStartRemoteDMATransfer function, presented in chapter 3. The complete function handles synchronization with a remote endpoint over the synchronization channel, and will perform a full end to end transfer. In this test we have only benchmarked the part which transfers data to the proxy using DMA transactions. This involves every step that the complete function does, except one; The endpoint will allocate buffers on the proxy using the distributed slab allocator (taking the segment lock while allocating), DMA data into the allocated buffers then allocate and initialize transaction ring nodes, as if the data was to be forwarded by the proxy. The only step which is not taken is the linking of the nodes, into the transaction ring itself and signaling the proxy to start processing the buffers. Instead, both buffers (from the buffer segment and transaction ring segment), are immediately freed after being initialized. These buffers are immediately freed by the endpoint so that they can be reused later.
As described in chapter 3, the size of the buffer segments has a big impact on the overall performance. The results, as can be seen in figure 5.2, includes benchmarks with buffer segment sizes ranging from 64 bytes to 512 KiB. The endpoint to proxy results are represented in the green plotted line. The blue line represents the raw SISCI performance between the same two machines, and was obtained from the dma_bench tool which is included in the Dolphin software stack. The red plotted line represents the raw SISCI PIO performance between the same two machines. These results were collected using the scibench2 tool, also included in the dolphin software stack.

![Figure 5.2: Endpoint to Proxy DMA bandwidth benchmark](image)

As can be seen from figure 5.2, both raw DMA and PIO are much faster than our SCIStrartRemoteDMATransfer function. With smaller buffer segment sizes, below 32KiB the differences in performance is especially obvious. A difference in performance is expected, because the SCIStrartRemoteDMATransfer also has to things other than just repeatedly transferring data. The very low bandwidth achieved with small buffer sizes, comes from the combination of additional traffic and waiting caused by the locking and unlocking of shared mutexes, allocation and freeing buffer slabs (which also generates traffic), and the fact that between each raw DMA transfer, the endpoint will wait, using the function SCIStrartRemoteDMATransfer. Because the remote transfer is split up into multiple smaller chunks, then transferred to the proxy so that it can start forwarding buffers to a remote cluster, the wait period is a necessity because it is important that the proxy will not start processing the buffers until all of the data has been transferred.
transferred. Should the proxy start processing the buffers before they were fully initialized (both with header fields and data) transaction errors will happen.

The graph shows that the bigger the buffers are, the greater the bandwidth is, the peak performance is with the buffer segment buffers set to 512 KiB (524288 bytes), with just below 20,000 Mbits pr. second. As we will see, this is also the trend with the other transfer methods. Our networking card, used for proxy to proxy data transfers have a theoretical upper limit of 10 Gigabits, so any buffer segment configuration above 64 KiB for remote DMA transfers, is sufficient to utilize the capabilities of the Ethernet link. As this configuration proved to give the best performance also for other tests, it was used in all the end to end measurements, presented later in this chapter.

**Endpoint to proxy DMAVec**

Here we present the bandwidth benchmarks for endpoint to proxy transfers for the `SCIStartRemoteDMATransferVec` call. Like in the previous section, we will only measure the bandwidth of the copy process from the endpoint to the proxy, so that we can identify which part in the end to end flow is the bottleneck. No end to end synchronization or transfers are performed in this benchmark. The endpoint will allocate slab buffers from the proxy, transfer data to them using vectored DMA, allocate transaction ring elements, initialize them but not link them into the transaction ring itself. So, everything is performed in the exact same way as if it was a complete transfer from endpoint node to a remote endpoint node, except that the packets only goes as far as the local proxy. After the buffer has been filled with data, and the tx-ring node has been initialized, they are immediately freed so that they can be used later.

The results can be seen in figure 5.3, and are presented in a similar manner to the previous section where we presented endpoint to proxy bandwidth for DMA transfers. The red plotted line represents the raw SISCI PIO performance obtained with the `scibench2` tool, the blue represents the raw SISCI DMA performance obtained with the `dma_bench` tool, and the green plotted line represents our remote DMA Vec results.

We see from the plot that the raw SISCI PIO performance has a lot more throughput that both our `SCIStartRemoteDMATransferVec` implementation as well as raw SISCI DMA performance. With the proxy configured with small buffer sizes we see that the performance is especially poor, and only starts to catch up with raw SISCI DMA performance when the buffer sizes are around 512 KiB. For buffer segment sizes below 8 KiB the bandwidth is only around 10 Mbits/sec with 64 byte buffer sizes and up to around 500 Mbits/sec for 4096 byte buffer sizes.

The reason for the large performance gap for smaller packet sizes is due to the fact that everything else that is going on takes so much time that it simply does not manage to keep up. While the benchmark tools, `dma_bench` and `scibench2`, has the advantage of repeatedly doing DMA/PIO transmissions with very little delay between each call to `SCIStartDMATransferVec`, our version has to do much more in between: Two critical regions when allocating buffer segment buffers, and transaction ring elements and initialization of headers and transaction ring nodes. Although some of that work is performed simultaneously with the DMA transmission, the time spent in between each call to `SCIStartDMATransferVec` prevents it from keeping up with the raw SISCI DMA performance.

The peak performance is gained with the individual buffers in the proxy buffer segment set to 512 KiB, achieving just above 19 Mbits/sec. This is more than enough to utilize the full
In this section we present the bandwidth benchmarks for endpoint to proxy transfer for the third and last remote transfer function \texttt{SCIRemoteMemCpy}. As the name implies it uses PIO to transfer data to the proxy instead of DMA like the two other functions. This test too, measures the bandwidth achieved for transferring data to the proxy, which includes every aspect of a complete endpoint to endpoint transfer except for the control channel is not in use, and the packet flow ends at the proxy.

The results can be seen in figure 5.4. As with the two other transfer methods we see a similar trend with the buffers in the buffer segments configured to small sizes. This function does however prove to be slightly faster than the other two functions for small buffer sizes. PIO transfers for smaller transfer sizes are known to be faster than PIO because there is no overhead associated with initializing the transfer, like with DMA. With buffer sizes from 64 bytes up to 32 KiB the PIO method is faster, but Vectored DMA taking over from the 64 KiB mark.

Compared to raw PIO performance (the red line) obtained from \texttt{scibench2} our \texttt{SCIRemoteMemCpy} is significantly slower. The reason being that the time spent in between the memory copy operations from the endpoint to the proxy is large enough to prevent running the memory
copy operations in a rapid enough succession to achieve similar results to that of raw PIO performance. As explained in the two previous sections this is also the case for the two other transfer methods. From the graph we can see that bandwidth performance of \texttt{SCIRemoteMemCpy} reaches above 10000 Mbits/sec with the buffer sizes configured to 128 KiB, which is enough to utilize the full theoretical potential of the 10 Gigabit link between the proxies.

**Proxy to proxy**

Here we present the results of bandwidth measurements of the data channel between two proxies. This is a TCP link established between the proxies of individual clusters, that is used indirectly by the endpoints of local clusters to send and receive to a remote cluster. This test will highlight any factors that might contribute to a bottleneck in full end to end transfers. We already know that the theoretical maximum possible bandwidth that can achieved is 10 Gigabit which is limited by the NIC used in our tests.

This test was performed by establishing a connection to the remote proxy when the proxy was started. Then the sender instance of the proxy allocated buffer segments and transaction ring nodes that was linked into its transaction ring. The spent initializing the transaction ring is not included in the measurements. After the transaction ring was filled, a timer was started, then the proxy would iterate the transaction ring, node by node, and transmit the contents of each of
them to the remote proxy. The sender proxy iterated the transaction ring repeatedly (taking and releasing the transaction ring lock for each iteration), without freeing and unlinking the buffer segments and transaction ring nodes, until enough time had expired so we could get an accurate measurement. Like with the endpoint to proxy measurements, we tested with the buffers in the buffer segment configured to different sizes, ranging from 64 bytes up to 512 KiB.

![Proxy to proxy bandwidth](image)

**Figure 5.5: Proxy to proxy PIO bandwidth benchmark**

The results are shown in figure 5.5 with the proxy to proxy bandwidth plotted in green. The blue line are the results of a bandwidth benchmark performed on the same link using a tool called Netperf [43]. This is a bandwidth benchmarking tool which is widely available for most Linux distributions.

We can see from the plot that both plots are fairly similar. Both the proxy to proxy and netperf performance stabilize at around 9100-9500 Mbits/sec with message sizes larger than 256 and 512 bytes. The proxy to proxy performance does however come short to that of netperf measurements with buffer sizes smaller than 1024 bytes. For 64 bytes message transfers netperf achieves around 1280 Mbits/sec more than the proxy achieves. The difference is the largest at 512 byte message sizes where netperf achieves 9115 Mbits/sec compared to the proxy sender which achieves roughly half as much.
**Endpoint to endpoint DMA**

In this section we will present the results that was obtained from full end to end transfers using the function `SCIStartRemoteDMATransfer`. The measurements obtained in this benchmark includes timing of every aspect of a transfer; synchronizing with the remote endpoint over the control channel, allocation of proxy buffers and transaction ring buffers, header initialization, linking nodes into the transaction ring, DMA data to proxy using `SCIDmaTransfer` and finally waiting for a confirmation from the remote endpoint on the control channel.

As with the other tests we performed we benchmarked using different size configurations of the individual buffers in the proxy buffer segment, ranging from 64 bytes up to 512 KiB. We compare the results with DMA performance achieved for endpoint to proxy transfers, and proxy to proxy transfers, presented in previous sections.

The results are shown in figure 5.6, where the green line plot represents the end to end results, the blue represents endpoint to proxy performance and the red proxy to proxy performance.

![Endpoint to endpoint DMA](image)

**Figure 5.6: Endpoint to endpoint DMA bandwidth benchmark**

We can see from the plot, that proxy to proxy performance is superior to the endpoint to proxy performance for message sizes up to the 128 KiB mark. This is because the proxy has significantly less work to do than the endpoint has to do in between each individual call to `SCIStartRemoteDMATransfer`; all the proxy needs to do is iterate the transaction ring, and forward the buffers located in it.
The end to end transfer bandwidth is limited by the endpoint to proxy bandwidth for smaller buffer configurations, this can be seen in the plot where the green line follows that blue (end to proxy DMA bandwidth) all the way from 64 bytes to 128 KiB until it flattens out and stabilizes around 9400 Mbits/sec only limited by the 10 Gigabit Ethernet NIC.

**Endpoint to endpoint DMAVec**

Here we present the end to end bandwidth results for the `SCIStartRemoteDMATransferVec` function. This is the second of three in total functions to perform inter cluster transfers, and uses vectored DMA to transfer data to the proxy. We present the results in the same way we presented the results for the non-vectored DMA function; We compare the end to end results with the measurements for the bandwidth achieved for vectored DMA from the endpoint to the proxy, and the proxy to proxy bandwidth.

![Endpoint to endpoint DMAVec](image)

*Figure 5.7: Endpoint to endpoint vectored DMA bandwidth benchmark*

From figure 5.7 we can see similar a similar tendency compared with the `SCIStartRemoteDMATransfer` performance presented in the previous section. The performance is still limited by the small endpoint to proxy bandwidth for the smaller buffer size configurations, while it reaches the max potential of the Ethernet link with buffer sizes of 128 KiB and above.
Endpoint to endpoint PIO

The bandwidth performance for code the SCIRemoteMemCpy function is presented in figure 5.8. The test was laid out in the same fashion as was done with the two other end to end transfer functions. We tested with different size configurations for the individual buffers in the proxy buffer segment, ranging from 64 bytes and up to 512 KiB. The results are compared with the endpoint to proxy PIO bandwidth, and proxy to proxy bandwidth.

![Endpoint to endpoint PIO](image)

Figure 5.8: Endpoint to endpoint PIO bandwidth benchmark

We can see also here that the bandwidth for smaller buffer size configuration is limited by the endpoint to proxy transfer performance. As this function works in the same way as the other two transfer functions, with regard to allocation of remote buffers, linking transaction ring nodes etc. we can see that all this work that the endpoint must perform is the greatest limiting factor for the bandwidth that can be achieved for smaller buffer sizes. In the cases where the buffers were configured to 128 KiB and larger however, the bandwidth is limited more by the 10 Gigabit NIC than the endpoint to proxy transfer speed.
5.3 Latency benchmarks

While bandwidth is an important factor of an inter-cluster data transfer system, latency is equally important. The previous section presented the performance of our prototype with regard to bandwidth. In this section we present the latency benchmarks for all three remote transfer functions.

We will measure the latency achieved for end to end transfers as well as latency on the Ethernet link. We will also present latency introduced by the control channel, at connection setup and during end to end transfers. We will compare our results to the latency performance on the Ethernet connection used for data exchange and the control channel. These numbers were measured by using a tool called `latency_bench`, included in the dolphin software stack.

Control channel

The control channel is established between two remote endpoints and is used to synchronize data transfers between two remote endpoints. The control messages exchanged on this channel is done over Ethernet using a TCP connection. The traffic on this channel is significantly different than that of the data channel used between the proxies, in the sense that it exchanges small messages (9 bytes) at the beginning of a remote transmission and when a transmission has finished.

At connection setup the sender node connects to the remote node using normal socket operations, then there is a message exchange between the two remote nodes that contains information that is used during data transmissions, after this message has been received the control channel is considered to be established. We measured the time it takes to set up the control channel and found that it averages at 120-125 micro-seconds (µs). As mentioned, the messages exchanged on the control channel are of a fixed size of 9 bytes. We benchmarked the average latency for this channel using fixed size messages equal to the size of the control messages. In the test a sender and a receiver node exchanges 9-byte messages over 100,000 repetitions, running for a total of around 3-4 seconds. We measured the average latency to be 18-19 µs.

Endpoint to endpoint DMA

Here we present the latency benchmarks for full end to end transfers using the `SCIStartRemoteDMATransfer` function. We compare the results with latency benchmarks obtained from the `latency_bench` tool. This tool measures latency with different message sizes between two machines using a TCP connection.

The test was set up in such a way that `SCIStartRemoteDMATransfer` was called with message sizes ranging from 4 bytes up to 64 KiB in size. For each message size a timer with micro-second resolution was started, then the transfer function was called in a loop with 100,000 iterations before the timer was stopped. Each message size was transferred 100,000 times in order to obtain an accurate measurement of the average latency.

Figure 5.9 shows the results we obtained (green line) and raw TCP latency obtained from `latency_bench` (red line). For message sizes smaller than 2048 bytes the latency is stable at 150 µs. From message sizes above 2048 bytes we see that the latency increases substantially up to 64 KiB where the latency is just above 300 µs. As expected, the latency for our end to end transfer is higher than that of the direct latency between two nodes, as represented in the red line. In our system, the message itself travels through an additional two hops before it reaches its
destination; the proxies. In addition to the delay of the data transfer there is also the delay added by the control channel communication. In a successful transfer there is a total of four transfers across two TCP connections; one over the proxy-to-proxy data channel, and three messages over the control channel a remote transfer request, an accept response from the receiver and a completed message, also from the receiving endpoint. This is illustrated in figure 4.9 on page 44. As stated above, the average latency for message exchanges across the control channel is 18-19 µs. Knowing this, we can calculate that the absolute minimum latency for an end to end transfer is \(18 \times 4 = 72\mu s\). This means that the process of allocating and transferring data to proxy buffers, linking tx-ring nodes, processing tx-ring buffers, receiving buffers on the remote proxy and transferring them to the destination endpoint takes \(150 - 72 = 78\mu s\) for message sizes ranging from 4 bytes up to 2 KiB.

**Endpoint to endpoint DMA Vec**

Figure 5.10 contains the latency benchmarks for the `SCIStartRemoteDMATransferVec` function. The results are compared to the latency on a direct link between two nodes using the `latency_bench` tool. Our test was performed in the same way we performed latency tests for the first transfer function above. For each message size, ranging from 4 bytes up to 64 KiB, we ran the transfer function in a loop of 100,000 iterations so that we can get a good measurement for the average latency.
The latency measurements obtained from `latency_bench` for message sizes smaller than 256 bytes are stable at 14-18 µs before it increases to 47 µs for 512 byte and up to 187 µs for 64 KiB message sizes. By comparing the two lines we can see that the differences in delay is at the greatest for the smaller message sizes, while the difference decreases as the message sizes increases.

The latency performance for the vectored DMA data transfer function is very similar to that of the DMA function and differs by only a few microseconds. This can be explained by the fact that the complete transaction process is also very similar, the only place it differs is in the way the messages are transferred to the proxy from the endpoint. Vectored DMA operations are designed to transmit multiple buffers, in order to reduce the overhead associated with initializing the DMA engine, and only a single message is transmitted at the time. This is why the DMA call is slightly faster than the vectored dma call with regard to latency.

**Endpoint to endpoint PIO**

The PIO transfer function is the third and last transfer function and uses PIO when transferring data to the proxy in contrast to the other two transfer functions which uses DMA operations. Here we present the latency benchmarks for it. The test was performed in the same manner as the two other latency tests was; The same number of iterations, and message sizes. The results can be seen in figure 5.11.
Starting at 4 byte message sizes, the latency is at 147 µs and steadily increases to 151 µs at the 256 byte mark. This is 3-4 µs less than the other two transfer methods which starts off at 150-151 µs. The reason that the PIO function provides slightly less delay comes from the fact that PIO transfers are considered to be faster for small message sizes.

The dominating factor that limits the latency from going any lower that it is at this point is not the way in which data is transferred from the endpoint to the proxy however. For PIO transfers directly between two SISCI segments, the latency is below 1 µs for 4-2048 byte transfers, and around 22 µs for 64 KiB transfers. The main contributing factor for the latencies we see for our end-to-end transfers is introduced by the process of sending the message itself and all the machines it passes through on its way to the destination.

5.4 Evaluation

With our test setup we have performed latency and bandwidth benchmarking tests which we have presented in this chapter. Looking back at the bandwidth performance results we presented in section 5.2 we see that we are able to achieve bandwidth very close to the maximum theoretical limit of 10 Gbit/sec imposed by the NIC. We were able to achieved this with buffer sizes configured to 256 KiB or more. Vectored DMA performed best of all the three functions we implemented, DMA performed close to Vectored DMA, and PIO slightly less. PIO proved to
perform slightly better bandwidth performance over the two other functions, for smaller buffer sizes. This is because PIO is known to perform better with smaller buffer sizes, both bandwidth and latency.

The bandwidth results also highlighted the shortcomings of our system. The bandwidth our implementation were able to achieve with small buffer sizes did not get close to the maximum theoretical speeds. In an ideal system, the performance of endpoint to endpoint data transfers should only be limited by the hardware that is used to transmit the data. Our tests show that the limiting piece of equipment is the 10 GE NIC we used, which was able to perform around 9.5 Gbits/sec with message sizes of 1 KiB and above. Below the 1 KiB mark it was the PCI Express interconnect hardware that was the limiting factor for DMA transfers. Testing shows the main issue that causes the low bandwidth results for small buffer sizes is that the endpoint has to do a lot of work in order to do a transfer. The biggest bottleneck comes from allocation of buffers from the remote proxy buffer segment, which are used as destination memory locations for outgoing data. While the algorithm for allocating buffers itself is fast it also generates traffic between the endpoint and the proxy because the internal state of the slab allocator is maintained by each process that uses it. While the traffic generated by the slab allocator is minimal, it adds up over time when the endpoint transfers large amounts of data at the time and only transmitting small pieces at the time. In order to fix this issue, the first step would be to improve the way in which the endpoint uses memory segments shared by the proxy in a way that minimizes communication overhead, currently associated with the buffer allocation process. A possible solution would be that the proxy could assign the endpoint a chunk of its data segment when the endpoint connects to the proxy. This way the endpoint would not need to contact the proxy in order to let it, and any other nodes in the same cluster, know that this chunk of memory is currently being used by someone. The endpoint simply uses it and maintains it the way it wants without worrying about others using it at the same time. When the endpoints wants the proxy to send its chunk of memory (or parts of it) to a remote destination, the proxy transaction ring would still have to be used. While this might improve the performance for small transfer sizes it does not fit our initial requirements. In a local cluster, there are typically many nodes, and assigning each node with a chunk of memory for it to keep as long as it is alive would quickly exhaust the resources of the proxy. The alternative is of course to only assign the nodes with a small chunk of memory so that memory would not be exhausted as fast, but this would prevent the endpoint from achieving the maximum potential of the 10 GE NIC.

A better way (in theory) to avoid communication overhead between the endpoint and the proxy would be to let the proxy handle the all the data transfers itself. All the endpoint would need to do is to insert nodes into the transaction ring of the proxy, that points to buffer segments local to the endpoint, instead of the proxy. When the proxy iterates the transaction ring it would retrieve data from the remote buffer belonging to the endpoint and into local proxy buffers. This would totally eliminate the need to allocate buffers from a remote segment.

The latency benchmarks we presented also shows that the latency for all three transfer functions are very similar, differing by only a few microseconds for small message sizes. Compared to raw latency between two directly connected machines the latency for our system is outperformed, because of the cost of synchronizing the transfer with the endpoint over the control channel, and the other aspects of the transfer. As mentioned above, the allocation of proxy side buffers proved to be an expensive operation, especially when transmitting many small pieces of data in a rapid succession. With a change in the way the proxy retrieves buffers to forward to a remote destination, that eliminates or minimizes the communication overhead with the proxy.
the latency would likely also be reduced.

5.5 Summary

In this chapter we have presented the performance benchmarks that our implementation was able to achieve. We have shown the overall performance for end-to-end data transfers, as well as individual components of the transfer process. Our benchmarks shows that we can transfer data between two endpoints in separate clusters, achieving bandwidth close to the maximum capacity of the NIC we used, for the optimal settings of the proxy side buffer segment buffers. We have also highlighted some issues with our implementation that prevents it from achieving better performance in cases where small amounts of data is transmitted in a rapid succession.

In the next chapter we conclude the thesis, where we summarize the work we have done in this thesis, list main contributions and look into possibilities for future work.
Chapter 6

Conclusion

In previous chapter we presented the performance results of the current version of our prototype. In this chapter we will summarize the work we have done in this thesis, look into possibilities for future work building on the work we have done, and present the main contributions.

6.1 Summary

In this thesis we have designed and implemented a working prototype that allows for nodes in separate computer clusters to connect to each other and perform data transfers between each other as if they were in the same cluster. The thesis starts with a brief introduction in chapter 1, along with the problem definition and the limitations that were set for the thesis.

In chapter 2 we gave an introduction to PCI Express standard on which the interconnect hardware we used in our implementation is based on. It is a standard defining a high performance serial I/O interconnect designed to replace older standards. PCI Express can be used to connect components internally in a computer, but can also be used to connect multiple individual machines to form a computer cluster. We also introduce Dolphin and the PCI Express based interconnect equipment they provided us with for the work in this thesis. Lastly we gave an introduction to SISCI. This is an API that can be used to program the interconnect hardware produced by Dolphin. It enables the programmer to connect to memory segments shared by other cluster nodes. After a node has successfully connected to a remote memory segment, he can perform memory read and write operations on it using either Direct Memory Access (DMA) or Programmed I/O (PIO). The SISCI API also implements the concepts of remote interrupts, which can be used to trigger events in a remote machine.

In chapter 3 we present the design and implementation details of the prototype. We presented an extension to the SISCI API (see section 2.3, page 15) that can be used by nodes in two separate PCI Express clusters to perform memory transfers between them in a similar fashion to the way it is done between nodes within a local cluster. We implemented three different functions that can be used to do inter-cluster memory transfers, one using PIO and two using DMA. All the components of the system is implemented in user space using the SISCI API which facilitates programming of the PCI Express cards used as the cluster interconnect medium. In our design each local cluster hosts a special node called the proxy. Nodes in a local cluster that wishes to transfer data to nodes in a remote cluster must utilize its local proxy to forward the data to the remote cluster, where the data is received by the remote proxy which in turn transfers it to the destination node. As mentioned, the transfer functions we implemented
uses DMA as a method to transmit data to a remote cluster-node. The DMA session extends from a local node to its local proxy where it is terminated, only to be initiated again at the remote proxy when the data is received from the Ethernet link.

Lastly, in chapter 5 we presented and evaluated the bandwidth and latency performance of data transfers. We started by looking at the performance of individual components of a full data transfer, before we presented the bandwidth and latency performance for all three data transfer functions we implemented.

### 6.2 Main Contributions

The goal with this thesis was to implement a prototype for doing long distance DMA transfers over PCI Express. This thesis presents the prototype we developed. We started by introducing the technology that the prototype is built on.

We have designed a system which relies on a common proxy being presented in each cluster. This proxy works as an interface to other remote clusters, and is used by local nodes when transferring data to a remote cluster.

We have also designed an end-to-end control channel protocol. This is a protocol used on a separate channel established directly between two nodes which is used for synchronization and error detection which might occur during transfers.

In chapter 5 we presented the bandwidth and latency performance of our system. We have shown that the prototype is capable of achieving bandwidth close to 10 Gbit which is the theoretical maximum bandwidth for the Network Interface Card (NIC) used in our testing. With certain configurations however, we have shown that the bandwidth is not as good, and that a redesign in the way a node transmits data might be needed in order to achieve better performance. With this we have learned that a crucial part of the system, with regard to performance, is the memory allocation in a remote memory segment has a big impact on the overall performance.

The result of this thesis is a working prototype of a system, capable of performing inter-cluster data transfers using functionality intended to extend the SISCI API.

### 6.3 Future work

In chapter 5 we presented benchmarks that showed sub-optimal bandwidth performance for small data transfers. The most important work that remains to be done for our design is to improve bandwidth for small data transfers. To get an increase in bandwidth one might have to rethink the parts of the transfer process in order to reduce the communication overhead between an endpoint and the proxy, when allocating proxy-buffers. In our design the endpoint starting a transmission has many responsibilities and duties it must perform. It makes sense that much of the work is implemented in the endpoint machine, because it leaves the proxy with less work, so that it can serve more nodes in an efficient way. However, a design where more responsibilities are moved to the proxy should be investigated and tested to see if an improvement in performance can be gained.

Another possibility for future work is to move functionality from user-space down to kernel-space. This is a natural next-step which should be investigated further. With a solution running in kernel-space there are many possibilities for optimizations not possible in user-space. For
example connecting different components more tightly, like the process of moving data from a SISCI memory segment out to the data link utilized by proxies.

An interesting possibility that could be investigated is that the endpoint machines can lend the Network Interface Card (NIC) hosted on the proxy machine, then use it as if it was a local device. This has been shown to work, and is presented in [44].

To further increase latency and bandwidth, one might need to utilize a data transfer protocol that is faster than TCP, which is what we used in our design. A good alternative is to use Multiprotocol Label Switching (MPLS). This is a high-performance mechanism which is situated between layer 2 and 3 in the OSI model. The idea is to attach a short fixed sized label to packets as they enter an MPLS domain. This label is used to make forwarding decisions within the MPLS domain [45]. This is a very simple and fast process, in contrast to packet forwarding based on IP addresses which includes complex lookups in routing tables.
Bibliography


Long-range RDMA over PCIe


