National Technical University of Athens

School of Electrical and Computer Engineering

Division of Computer Science

Extending an asynchronous messaging library using an RDMA-enabled interconnect.

Diploma Thesis

of

Konstantinos S. Alexopoulos

Supervisor: Georgios Goumas
Assistant Professor N.T.U.A.

Computing Systems Lab

Athens, October 2017

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science
Computing Systems Lab

Extending an asynchronous messaging library using an RDMA-enabled interconnect.

Diploma Thesis

of

Konstantinos S. Alexopoulos

Supervisor: Georgios Goumas
Assistant Professsor, N.T.U.A.

Approved by the committee on October 30th 2017.

(Signature)	*(Signature)*	*(Signature)*
.....	.....	.....
Georgios Goumas	Nektarios Koziris	Dimitrios Soudris
Assistant Professor N.T.U.A.	Professor N.T.U.A.	Associate Professor N.T.U.A.

Athens, October 2017

(Signature)

.....

Konstantinos S. Alexopoulos
Electrical and Computer Engineer N.T.U.A.

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science
Computing Systems Lab

Acknowledgments

I would like to thank Associate Professor Georgios Goumas, for the trust he showed in me and the opportunity he gave me to carry out my diploma thesis as part of the Computing Systems Laboratory of NTUA. I would also like to thank PhD candidate, Stefanos Gerangelos for his resolute guidance during the last phases of my work.

I want to thank my former colleagues at CERN, Olof Bärring, Sima Baymani, Alexandru Grigore and Aram Santogidis, for eagerly communicating their wisdom and experience to me, helping me to become a better professional.

I thank my family for their support during the years I spent studying, my father, Spiros, for teaching me to have foresight and my mother, Sandy, for teaching me the values of resilience and grit, but first and foremost I thank my grandmother, Vana, who has always supported me unconditionally and to which I will be eternally grateful for helping me become the person I am today.

Furthermore, I want to thank my classmates, who taught me the value of cooperation and eased the burden of my academic endeavour.

Lastly, I want to thank Eleni, who patiently stood by my side during the difficult process of compiling the contents of this thesis, constantly offering her encouragement.

Abstract

As computing power and I/O performance is increasing at an aggressive rate several RDMA enabled interconnect technologies have been entering the market, promising low latency and high throughput. RDMA concepts are based on the support for zero-copy operations and CPU-offloading by supporting writes directly to remote memory areas. However, the majority of distributed, network intensive, applications today are designed around socket interfaces, which are inherently incompatible with the RDMA approach.

The purpose of this thesis is to address this incompatibility between the well-established, and emerging, communication paradigms, and to offer an interface for exploiting memory coherent communication in an HPC context. This is achieved by extending ZeroMQ, a high-performance asynchronous messaging library, to use the RapidIO transport, a high-performance, packet-switched, RDMA-enabled interconnect technology. ZeroMQ lends itself well to the scope of this thesis, as its transport layer is abstracted and has already been extended to a number of different protocols. Moreover, it allows for trivial employment regardless of transport used, facilitating the effortless application of an extension.

Through this work, the effort of extending a distributed application, heavily reliant on socket interfaces, is documented, while evaluating the challenges that accompany the aforementioned, paradigm translation. Conclusions are drawn concerning performance differences as well as limitations in the development process, that come with the employment of a new technology, in our case an RDMA-enabled interconnect.

Keywords

Interconnect Networks, High Performance Computing, Distributed Systems, Remote Direct Memory Access, ZeroMQ, RapidIO

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As Moore's law predicted, processing capabilities in modern computers continue to meet growth in an exponential manner. However, an interpretation of Amdahl's law states that the efficiency of a system can only be assessed as the balance between CPU processing power, memory bandwidth and input/output performance. Interconnection networks exhibit design constraints greater than those found in semiconductor design. A bottleneck is therefore exhibited. At the same time the component number that needs to be networked is increasing rapidly. Current I/O architectures are not optimally designed to support this. [1]

To try and mitigate this, system designers have lately started to abandon the traditionally used shared bus design, in favor of a switched fabric, which favors bandwidth and scalability. [2]

With the latest developments in the Cloud and Internet domains, data centers are more relevant than ever. An increasing number of services and applications are run in a data center context. This has exhibited an unprecedented need for scaling, causing the High Performance Computing field to define new performance requirements. Furthermore, data centers today often also employ FPGA, GPU, or Storage systems, which may employ their own, exclusive, proprietary interconnects. [3]

Several interconnect architectures have been proposed, aspiring to bridge intra- and inter-chassis communications and to support heterogeneosity. These switched fabric architectures usually support memory coherency schemes, which require

the respective coding effort to be applied. Additionally, the upgrade costs are a problematic factor, as new hardware across the fabric needs to be employed.

1.1 Thesis Objective

The objective of this thesis, is to provide a communication interface for a distributed, multi-node environment, that operates over an RDMA-enabled transport, consistent with modern, cutting-edge interconnect technologies. By doing so, the coding effort is concealed in the interface implementation, allowing for seamless employment in already existing systems. Additionally, the bandwidth and scalability capabilities of an RDMA-enabled interconnect can be utilized in an HPC context.

Modern networking applications are designed around the use of socket interfaces. Sockets provide a convenient and reliable interface for network communication. However, this comes at the cost of processing power. Protocols like TCP/IP traverse the network stack and consume the CPU cycles of both the local and remote endpoints. This negatively affects both latency and scalability.

These aspects are of importance in an HPC context, were performance outweighs the need for convenience. There, a memory coherency scheme can achieve better latencies for inter-node communication, by off-loading the CPU. RDMA-enabled interconnects also excel in multi-node systems, where lower transaction overhead is translated to higher bandwidth. However, RDMA-enabled transports require programming effort to implement and maintain, as special setup and memory operations need to take place. They are thus employed only when performance is crucial.

Consequently, an inherent incompatibility between socket interfaces and memory coherence schemes is observed. In the scope of this work, an solution that bridges the two is investigated.

To address these issues, ZeroMQ [4], a distributed messaging library, has been extended to use the RapidIO [5] transport. ZeroMQ offers a messaging interface, which allows the user to choose the transport used in a trivial

way. On top of that it is optimized to overcome major messaging overhead, maintaining overall system performance. Lastly, it is strongly tied around socket programming, allowing us to propose a scheme to tie the socket and memory coherency communication paradigms.

1.1.1 Contributions

Contributions done as part of this diploma thesis can be summarized as follows:

- The ZeroMQ library has been extended to use the RapidIO transport, based on an RDMA-enabled interconnect
- The RapidIO protocol has been applied in a multi-node, HPC context, outside of its usual embedded setting.
- The application of RDMA concepts is investigated in a practical context.

1.2 Document Structure

The structure of this thesis is described as follows.

In Chapter 2 the reader is introduced to the necessary theoretical background on which this thesis is based, as well as key points of the technologies and frameworks used.

Chapter 3 focuses on the design of the library's extension, as well as its implementation details.

The benchmarks used to collect performance results, and their evaluation are presented in Chapter 4.

Related work is then presented in Chapter 5, before Chapter 6, where conclusions and possible future work is discussed.

Chapter 2

Background

This chapter aims to introduce the reader to the terms and concepts, which are necessary to follow the presentation of this paper. These are divided into three parts: The first one introduces concepts related to interconnect networks in general. The second is about RapidIO, the interconnect protocol and technology used in the scope of this project. The third refers to ZeroMQ, the messaging framework that was extended.

2.1 Interconnect Technologies

The term interconnection refers to the networking of two or more computer components, or endpoints. These can be anything from integrated circuits, boards and computer chassis to wide area networks. Interconnects define the I/O capabilities of every component, like the CPU and the memory. However, the efficiency of the system is dictated by the balance between CPU performance, memory bandwidth as well as the speed of I/O operations, as stated by Amdahl's law. In other words, even if individual components can achieve high throughput, it will not be beneficial until that component can be efficiently linked with the rest of its system. According to Moore's law, semiconductor speed and size tend to improve faster compared to the electrical and mechanical limitations of interconnect design. Conclusively a fundamental imbalance presents itself to system design, where component power cannot be used efficiently as part of a system. In order to overcome these limitations, several

high performance interconnects have been introduced.

The most established system-level interconnect is PCI (Peripheral Component Interconnect). PCI supports a bus architecture which has been dictating electrical and mechanical design of computer parts for the better part of the last 30 years. Due to the effect PCI has had in design, it has gathered significant inertia as the industry standard, making it very difficult for new technologies to seize one of its application domains.

However, an interconnect does not need to only target the system level. Out of the box connections could also be improved as a part of a unified interconnection fabric. This prospect paves the road to other application contexts. A number of applications today is run by high performance cluster topologies, where load is distributed among the system's servers most often as part of a data center. The performance of such a system could be further improved by employing the same interconnect both on the PCB (Printed Circuit Board) and the networking level. Maintaining a single protocol across nodes favors scalabality, by significantly lowering routing overhead between components. Conclusively, a high performing interconnect could benefit the whole hardware stack, currently present in the rapidly evolving HPC (High Performance Computing) field. [1]

2.1.1 Bus and Fabric Architectures

The most common interconnect architecture today is the shared bus architecture. Although shared buses are imposing many limitations in system design today, their traditional use has allowed them to stay prevalent through inertia. Modern server clusters require throughput as well as reliability, that current PCI revisions are not able to support. This problem has been addressed by the introduction of interconnects that are based on a fabric architecture, that enable high speeds and scalable physical network topologies.

In a system that employs a bus architecture, all components are connected to the same bus. Therefore, all transactions will share the same communication medium. As a direct consequence, increasing the number of connections on the same bus decreases the bandwidth per capita. In order to mitigate this constraint shared bus hierarchies have been utilized, separating high- performance

components from their low-performing counterparts, see figure 2.1. However a bottleneck still persists, since these two groups will again have to be connected through a central bus. Such design techniques impose mechanical and electrical limitations which soon lead to congestion, keeping the possible number of connected endpoints low and undermining fault tolerance. Moreover buses cannot be extended to external communication, requiring the use of a separate interconnect. [6, 7]

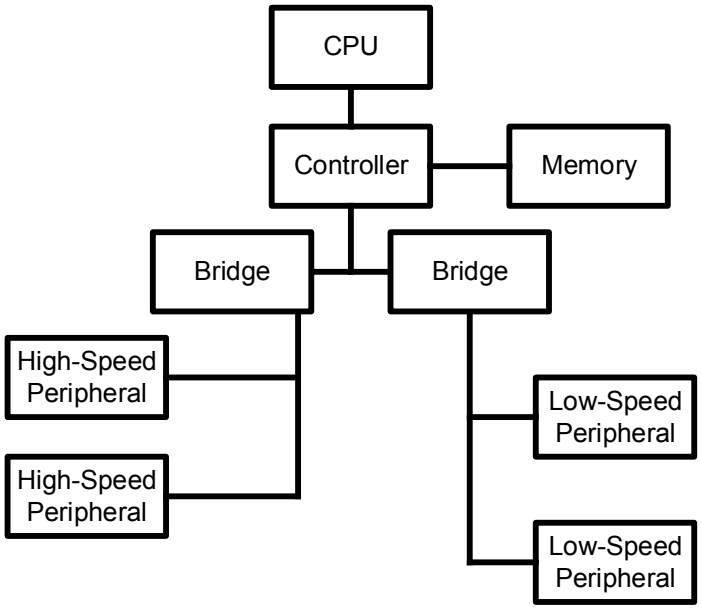

Figure 2.1: Typical shared bus architecture

Switched fabric architectures are point-to-point switch-based interconnect designs. Every component employs a link with exactly one device on the other end. This guarantees much higher performance compared to a bus architecture, while also maintaining low complexity for producing and detecting errors. Furthermore the switched architecture allows for better scalability, as the bandwidth of the overall system will not be hindered when new nodes are added, given a sufficiently-performing number of switches. This scheme also allows for multiple paths between devices, enhancing robustness in case of failure. [7]

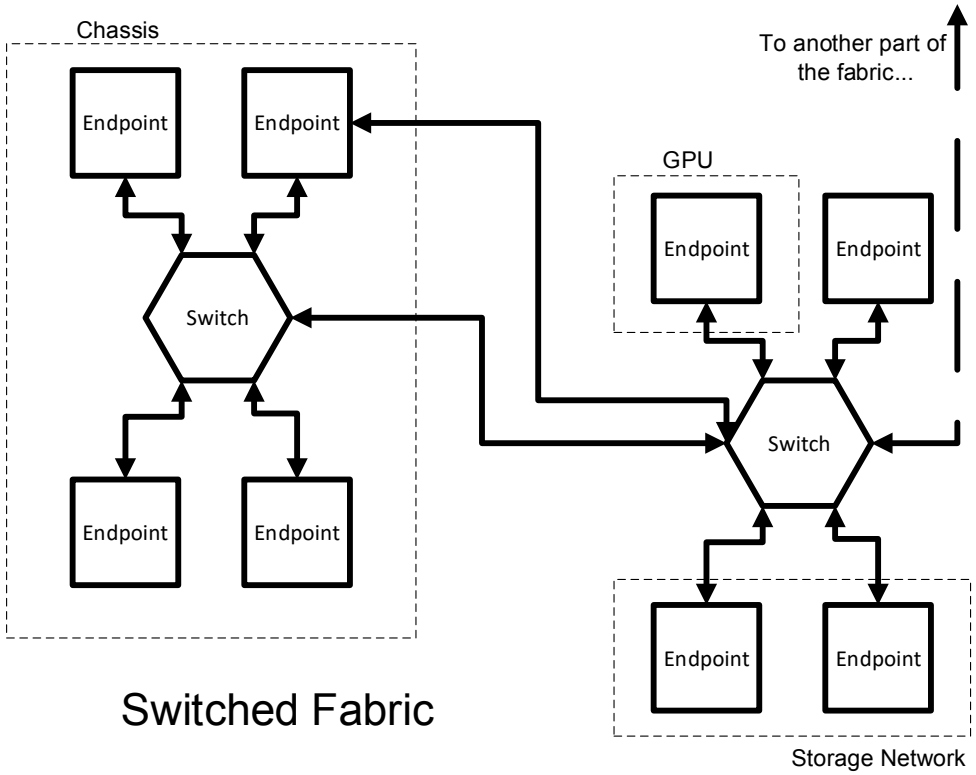

Figure 2.2: Switched Fabric

2.1.2 Interconnection in High Performance Computing

High Performance Computing, or HPC for short, is a term that is used to describe the computing domain responsible for computationally intensive tasks. Such tasks often stem from fields than involve physics, forecasting, simulations and modeling, among others. In the context of HPC, supercomputers are used, as opposed to consumer, general-purpose computers. The first supercomputers were manufactured in the 60s and were essentially boosted versions of their consumer-grade counterparts. As years went by an increasing level of parallelism was introduced, with supercomputers having a large number of processing cores. As is natural, at one point the number of cores were interconnected, by using multiple chassis in conjunction with one another. And thus supercomputers started to be clusters of units, with each one of them contributing a number of processing cores. It was at this step that the interconnect performance started being critical at the chassis-to-chassis level. A

chassis-to-chassis level interconnect in the context of HPC, need not only offer high bandwidth, but also extremely low latency, as commands may also need to be propagated between units. Of course, scalability and fault tolerance are also important requirements when designing a large-scale system.

In response to these developments several interconnect technologies have hit the market in recent years, the most widespread of which is Infiniband. At the same time, other technologies that already have a market share are evolving to adapt to the new data. An example of this is the upgrade of Ethernet to speeds of 10, 40 and 100 Gbps. An interconnect standard that has also evolved to claim a share of this market is RapidIO. Following are some details about the aforementioned interconnects.

Infiniband

Infiniband (IB) is a switch-based serial I/O interconnect architecture. It supports high throughput and low latency, with link speeds ranging from 10Gb/s to 56Gb/s. It mainly finds application in data interconnection between computers. Due to its performance it is usually used in an HPC context, like computational intensive tasks or storage. Infiniband uses RDMA and messaging for operations, compared to TCP/IP over Ethernet. This could mean the potential need for development effort, in order for previous software to be able to take advantage of Infiniband's performance.

10Gb/40Gb/100Gb Ethernet

The 10Gb,40Gb and 100Gb Ethernet, most commonly refered to as 10GbE, 40GbE and 100GbE, are the updated standards of the Ethernet technologies. Some of these high speed Ethernet versions support backward compatibility. In other words, the cables and switches used for these technologies may be compatible with previously used versions of the protocol. Since, most HPC contexts will already be set up with some kind of mainstream Ethernet backbone, this translates to significantly lower upgrade costs.

The next chapter introduces RapidIO, a system-level interconnect that is also

targeted to the System Area Network field. A System Area Network (SAN) is another term for describing a "local" network, usually found in the context of a data center, which is designed for high speed interconnection in both cluster and multiprocessing environments. However, it's possible that SAN may also refer to intra-chassis interconnection.

2.2 RapidIO

RapidIO is a high-performance, low pin count, packet-switched system level interconnect standard.

Contrary to shared memory bus designs, like PCI, RapidIO is a switched fabric interconnect technology. Infiniband also falls in this category. Switched fabric interconnects are similar to switched network architectures. Every endpoint in the system has exactly one link to another component, which may be another endpoint or a switch element. Operations in switched fabric interconnects normally include a mechanism to exchange messages and send events, in the context of the network. Memory-mapped I/O, or MMIO, is usually defined as an optional communication means in this domain.

RapidIO is an open standard and its applications include the interconnection of microprocessors, memory, memory mapped I/O devices, storage and computing systems. It originally intended to function as a front side bus, but has since evolved into a prominent system level interconnect, with wide applications in an embedded context. At the time of writing, more than 200 million RapidIO fabric ports have been deployed worldwide. Primarily in the field of telecommunications, RapidIO is present at every 4G base station in the world. However, the protocol is independent of a physical implementation, and thus the architecture is a great candidate for interconnecting systems that would traditionally be found in different positions of the shared bus hierarchy. Essentially RapidIO can be applied to any domain, from chip-to-chip communication to cluster connection in data centers. see Figure 2.3 [8].

This presents the opportunity to seamlessly integrate FPGAs, GPUs, CPUs and storage system, as part of the same fabric, making RapidIO ideal for heterogeneous systems, see Figure 2.4. RapidIO is currently implemented natively

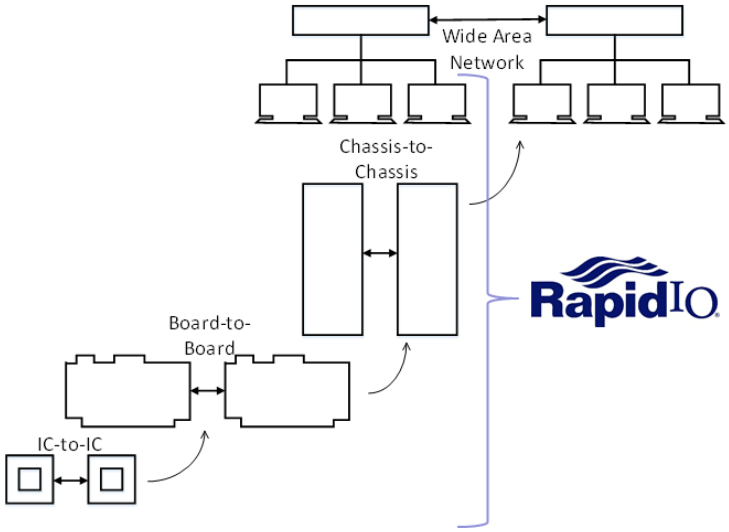

Figure 2.3: Interconnect application domains where RapidIO is applicable

on-chip and on PCIe bridge cards.

Its latest specification [9] was released in June 2016, stating that speeds of up to 25Gbaud per lane can be achieved. This translates to a RapidIO network speed of 100Gb/s.

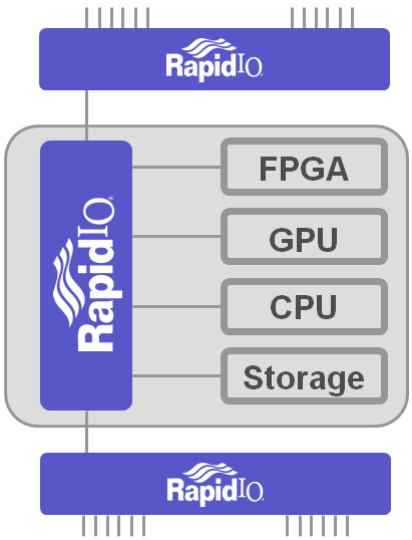

Figure 2.4: RapidIO exploitation in heterogeneous systems

2.2.1 Protocol

RapidIO is primarily hardware implemented. Error handling mainly takes place at the physical level and hardware termination is offered. This allows for the achievement of low latencies and, most importantly, offloads the CPU. By transferring the protocol overhead to the hardware, CPU cycles are freed, lowering power consumption and allowing the application of the technology on realtime systems, as well as the scale-up of the system. The protocol supports destination based routing and permits the deployment of any network topology. Most importantly, the specifications of the protocol may be partitioned, in order to meet the specific needs of the physical implementation in question. In this way, the complexity of the system and the component may be limited, while simultaneously allowing for future expansions.

The RapidIO architecture is specified as a three-layer hierarchy, comprised of the logical, the transport and the physical specification. [5]

- Logical Layer

The Logical Layer defines the overall protocol and packet formats, in other words, the manner in which transaction are handled by the endpoints. The transactions types defined in this layer are too.

The first one is a message-based access to devices. A hardware port that can send and received messages is defined as the mailbox. Messages conveyed in this manner can consist of up to 16 packets, of up to 256 bytes each. This makes out to a maximum message size of up to 4KB. Another port accepting messages is the doorbell. These lightweight messages are commonly used for event notifications, in a way similar to the signals and interrupts. They are able to carry information in a 16-bit software definable field. This transaction type can also be referred to as the message passing programming model of the protocol.

A globally shared distributed memory programming model is also supported. This allows for the use of memory mapped I/O operations. This transaction type enables the access of local memory space to perform read/write operations that are translated to the respective operations on the remote memory space. This concept can most commonly be refered to as Remote Direct Memory Access

or RDMA. This model is the one that enables RapidIO, an RDMA-enabled interconnect, to find application in multi-processing, multi-node systems that exhibit HPC needs.

- Transport Layer

The Transport Layer provides the necessary information for the successful routing of the RapidIO packets from one endpoint to an another. RapidIO packets are routed based on a unique ID assigned to each device. The assignment of IDs is done through a process that is called enumeration, which takes place in software when setting up the system. The ID can take a maximum value of 65535, allowing for 65536 devices to coexist in the same system.

It is important to note, that the RapidIO protocol does not care for the topology of the physical interconnect. It can be configured to function in the context of virtually any network topology, from trees and meshes, to hypercubes and toroids. RapidIO employs source routing. Each packet has a destination address that is specified by the source. This information is then handled appropriately by the fabric, which will route the transaction. In this manner only resources at the source and destination endpoints will be occupied, allowing for concurrent transactions to take place unburdened. This contradicts the broadcast scheme, found in traditional bus-based systems like PCI, in which a packet is sent to all devices.

- Physical Layer

The Physical Layer offers the, nowadays, serial implementation of the physical interconnect. This includes the mechanism to transport the packets, flow control information, hardware error management as well as the electrical characteristics of the device. The packets of the physical layer are small, with low protocol overhead, allowing for fast processing and CPU offloading. When error recovery takes place in hardware, acknowledgment and retransmission of corrupted or lost packages are commenced immediately. The above provide for a swift and reliable delivery of individual packets.

Flow control is also defined in the physical layer, and special care is taken to ensure that overhead and complexity is limited. Three types of flow control mechanisms are defined, so that higher priority transaction take precedent over lower priority ones; retry, throttle, and credit based. The retry mechanism, the

simplest one, simply states that when a receiver cannot receive a packet for any reason, can reply with a "retry" control symbol to the source endpoint. Then the source may retransmit the packet. The throttle mechanism utilizes the "idle" control symbol, allowing for the insertion of packets that act as wait signals. Lastly, the credit based mechanism exploits certain control symbols to communicate buffer state between endpoints. In that manner, a transaction can be initiated only when a buffer becomes available.

RapidIO provides a rich set of maintenance and error management functions. A dedicated maintenance port can be found in every device. It offers access to registers that contain information about the device, including capabilities and memory information, error detection and status registers. These registers can also be used to reset a device if need be. The physical layer specification also handles error coverage. Error detection is managed through the exchange of control symbols between the endpoints. In case re-synchronization o retransmission is not successful the RapidIO hardware can generate a software trap, interrupting the software, handing over the responsibility.

The protocol is defined in a way to be physical layer independent. This means that RapidIO can be used over serial or parallel interfaces, copper or fiber media. RapidIO addresses power consumption concerns by employing LVDS, Low Voltage Differential Signaling.

A rough correspondence to the layers presented above and the Open Systems Interconnection (OSI) model, can be seen in Figure 2.5.

2.2.2 RDMA

RapidIO is RDMA-enabled. This is another way of stating that the RapidIO protocol supports memory mapped I/O operations. Generally RDMA-enabled interconnects refer to the group of interconnects that employ techniques for maintaining memory coherency. This enables memory that is physically located in different places in or outside of the machine, to be shared among different processing components.

RDMA stands for Remote Direct Memory Access. RMDA-enabled intercon-

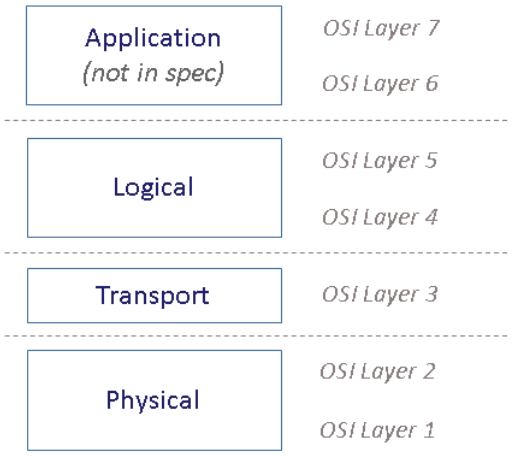

Figure 2.5: RapidIO protocol specification layers and their corresponding OSI layers.

nects share three characteristics that make them attractive for HPC related applications.

- **Zero-copy** The term Zero-copy refers to the ability to move a piece of data from a buffer residing in node A to a buffer residing node B without the involvement of the network software stack. This is obviously possible with memory mapped I/O operations if the programmatical interface allows for it. Normally a send/receive operation should involve the following. The sender has to pass the source buffer address pointer, and an address of the shared memory to the interface, and the receiver simply has to read from that address in shared memory. This eliminates the need for an intermediate buffer, and the memory copying overhead that accompanies it.

- **Kernel bypass** Kernel involvement is not required to perform a transfer. In other words, a context change will not necessarily take place in order for a buffer to be successfully written/read on/from the remote end. Any necessary information is exchanged during the setup phase of a connection. This includes the exchange of memory addresses to be used between the endpoints and their respective mapping to local physical memory and application address space. Consequently a system call does not need to take place when a transaction is performed for an established connection.

- **CPU offloading** By bypassing the network stack and the kernel, the CPU cycles consumed as a result of a transaction are effectively lowered. At the same time CPU cycles of the remote system do not need to be consumed. This

side-effect is refered to as CPU offloading.

2.2.3 Challenges in RDMA

Memory setup

RDMA can have many upsides, however the respective effort has to be put in from the programmer to tackle its use.

First and foremost, certain memory areas have to be reserved beforehand, to be used exclusively for RDMA operations. For a regular Linux system this will happen at boot time, by modifying the kernel boot command-line. There, a parameter needs to be added that specifies the beginning address of the memory to be reserved, as well as its size. After the required memory is reserved, the application that intends to use it has to map it to its process address space. For example, when utilizing RapidIO, the physical memory will have to be mapped to the address space that is visible to the RapidIO device. This memory scheme is depicted in Figure 2.6.

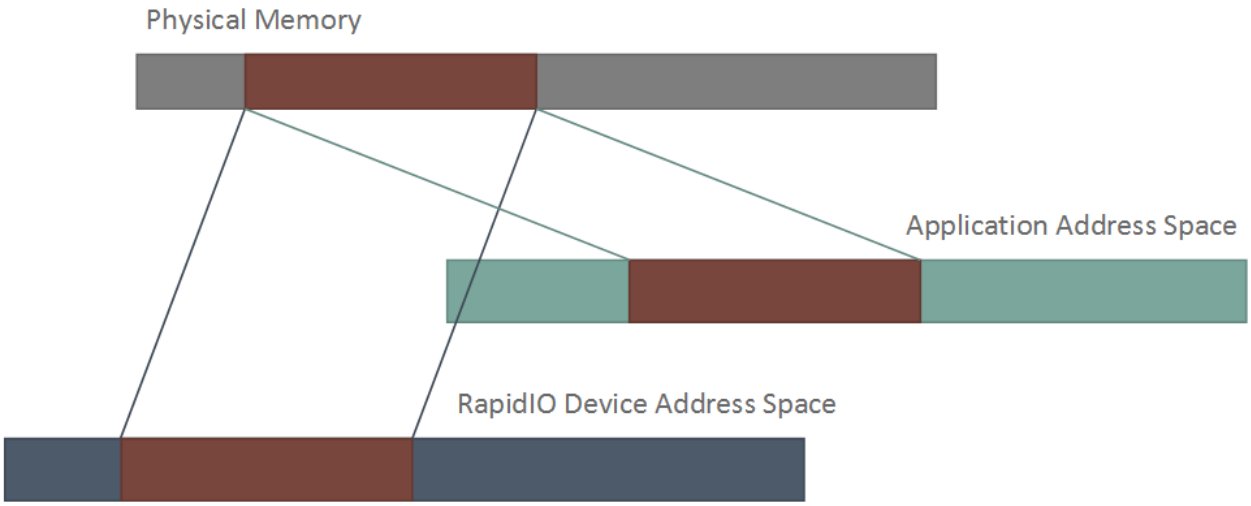

Figure 2.6: Memory mapping when using RDMA

The mapped memory address then has to be communicated to remote end, or ends, in order to perform RDMA operations. This is something that has to be done manually by the running application.

Orchestration

RDMA is often described as "one-way communication". This stems as a comparison to the "two-way communication" of a Send/Receive model. In the Send/Receive model, the sender will send data to a target, with that target waiting to receive this data and in turn save it at a specified location. For RDMA operations this is not the case, as one end will make a direct access to memory, which does not require the involvement of the remote system's CPU. The obvious and direct side-effect of this is that the remote end has to somehow be notified about the memory access. In other words, RDMA operations require orchestration as part of the user's involvement.

In normal circumstances, setting up the RDMA environment, in order to perform a single transaction would abstractly require the following steps:

On the receiving side:

1. Map Reserved Physical Memory to process address space
2. Allocate inbound window
3. Send inbound window target address to the remote end
4. Wait for write completion notification
5. Read memory

On the sending side:

1. Receive RapidIO target address from the remote end
2. Perform write operation to remote memory
3. Notify remote end about completion of write operation

For RapidIO this orchestration can be done through the message passing programming interface introduced previously. The possibilities include messages which use a sub-performing mailbox port interface, and doorbells, which are hardware events. Alternatively even the polling of RDMA memory could be exploited, by waiting until some pre-agreed alteration takes place, such as the value change of a certain bit. Similar methods are normally used for other interconnects of comparable design.

Another important aspect is data preservation. Memory can and will be overwritten if a locking mechanism preventing operations is not in place. This weight falls to the programmer and can be proven quite challenging when handling a large number of memory chunks, especially when these are originating from multiple nodes.

2.2.4 Link Speed

Theoretical Approach

The network interface cards used for the purposes of this research are PCIe to RapidIO bridge cards. They offer a line rate of up to 16Gbps. Both the PCIe and RapidIO sides support:

- 5 GBaud link speed
- x4 link width
- 8b\10b encoding : 8bit words are mapped to 10bit symbols

Thus, the link speed in bits is calculated as follows:

$$\textbf{Link Speed} = \frac{(linkwidth) * (linkspeed) * (\#bits)}{(\#symbols)} Gbps \implies$$

$$\textbf{Link Speed} = \frac{(4) * (5) * (8)}{(10)} = 16Gbps$$

Nevertheless, the PCIe to RapidIO translations, and vice versa, also have to be accounted for. This means that the speed calculated in the previous equation is never observed in reality. The actual link speed falls to around 13.4Gbps. However, this is only true in regards to the transmit operations. Due to better protocol mapping for the receive operations, the overhead is lower, and the receive operations can reach 14.6Gbps.

The above values refer to DMA reads and writes, since these operations require the least overhead, and can support transactions of sufficient size to observe high throughput.

The analysis of both of these operations is presented below. Each action is

referred to later by an action step number in brackets. This part refers to both the PCI [10] and RapidIO [11] specifications.

DMA Transmit

For a DMA Transmit operation the data starts from the CPU, it is then forwarded to the Tsi721 card, which then pushes it to the RapidIO network. A description of this procedure follows, with detailed calculation of the number of bytes every transaction requires. This clears up the percentage of the link speed occupied by overhead.

– The CPU wants to write to the Tsi721 –

CPU -> Tsi721

The CPU does a register write to trigger Tx on the PCIe side of the Tsi721. [1]

$S_c = (PCIeRegisterWrite + RegisterWritePaylaod) = 28Bytes$

CPU <- Tsi721 [2]

PCIe responds with a PCIe Acknowledgment and Credit Update. [2]

$S_p = (PCIeAcknowledgment + PCIeCreditUpdate) = (8 + 8) = 16Bytes$

– Tsi721 requests a read descriptor from the CPU –

CPU <- Tsi721

PCIe makes a read request to the CPU. [3]

$S_{p0} = (PCIeReadRequest) = 24Bytes$

CPU -> Tsi721

The CPU acknowledges the request. [4]

$S_{c0} = (PCIeAcknowledgment + PCIeCreditUpdate) = (8 + 8) = 16Bytes$

And then sends the descriptor. [5]

$S_{c1} = (PCIeReadCompletionOverhead + DescriptorSize) = (24 + 64) = 88Bytes$

$S_c = S_c0 + S_c1 = 104Bytes$

CPU <- Tsi721

PCIe acknowledges the last operation. [6]

$S_{p1} = (PCIeAcknowledgment + PCIeCreditUpdate) = (8 + 8) = 16Bytes$

$S_p = S_{p0} + S_{p1}40Bytes$

– Read the data from the CPU –

CPU <- Tsi721

PCIe makes read requests. [7]

$S_{p0} = ((MTU/PCIeMaxReadSize) * ReadRequestSize) = (65536/4096) * 24 = 384Bytes$

CPU -> Tsi721

The CPU acknowledges the requests [8]

$S_{c0} = ((MTU/PCIeMaxReadSize)*(PCIeAcknowledgment+PCIeCreditUpdate)) = (655636/4096) * (8 + 8) = 256Bytes$

And sends the data [9]

$S_{c1} = MTU/PCIeEfficiency = 65536/84.21\% = 77824Bytes$

$S_c = S_{c0} + S_{c1} = 77824 + 256 = 78080Bytes$

CPU <- Tsi721

PCIe acknowledges the data. [10]

$S_{p1} = (MTU/PCIePayload) * (PCIeAcknowledgment + PCIeCreditUpdate) = (65536/128) * (8 + 8) = 8192Bytes$

$S_p = S_{p0} + S_{p1} = 8576Bytes$

– Tsi721 sends the data to the RapidIO network –

Tsi721 -> RIO

Tsi721 sends the data. [11]

$S_{p0} = MTU/RIOEfficiency = 65536/94.12\% = 69630Bytes$

And a control symbol for every packet sent. [12]

$S_{p1} = (MTU/RIOPayload) * RIOControlSymbolSize = (65536/256) * 4 = 1024Bytes$

$S_p = S_{p0} + S_{p1} = 70654Bytes$

Tsi721 <- RIO

RapidIO acknowledges every packet received with a control symbol [13].

$S_r = (MTU/RIOPayload) * RIOControlSymbolSize = (65536/256) * 4 = 1024Bytes$

– Tsi721 has to inform the CPU of the completion –

CPU <- Tsi721

PCIe performs a write completion to the CPU [14]

$S_p = (PCIeTotal - PCIePayload + WriteCompletionSize) = 152 - 128 + 64 = 88Bytes$

CPU -> Tsi721

The CPU Acknowledges the write. [15]

$S_c = PCIeAcknowledgment + PCIeCreditUpdate = 16Bytes$

DMA Receive

For a DMA Receive operation the data is Received from the RapidIO network to the Tsi721, and is then forwarded to the CPU.

For this operation the direction of the data is: RapidIO Network -> Tsi721 -> CPU.

– Tsi721 receives data from the RapidIO network –

Tsi721 <- RIO

This is exactly the same as the opposite direction from the data transmission step on DMA Tx.

Data, [1]

$S_{r0} = MTU/RIOEfficiency = 65536/94.12 \ = 69630Bytes$

And a control symbol for every packet sent, [2]

$S_{r1} = (MTU/RIOPayload) * RIOControlSymbolSize = (65536/256) * 4 = 1024Bytes$

$S_r = S_{r0} + S_{r1} = 70654Bytes$

Tsi721 -> RIO

As before, the same as the opposite direction from the data transmission step on DMA Tx.

Tsi721 acknowledges every packet received with a control symbol. [3]

$S_p = (MTU/RIOPayload) * RIOControlSymbolSize = (65536/256) * 4 = 1024 Bytes$

– Tsi721 sends the data to the CPU –

CPU <- Tsi721

Tsi721 sends the data from the RapidIO packets on the PCIe link to the CPU. [4]

$S = (MTU/PCIeEfficiency) = (65536/0.8421) = 77824 Bytes$

CPU -> Tsi721

And the CPU acknowledges. [5]

$S = (MTU/RIOPayload) * (PCIeAcknowledgment + PCIeCreditUpdate) = (65536/256) * (8 + 8) = 4096 Bytes$

All the numbers used in the above calculations can be found in the specifications for RapidIO and PCIe, as well as the relevant table under Supporting Materials.

Speed Calculations

DMA Tx

By summing up every S_c and S_p on the **PCIe bus side** of the DMA Tx operation we get the following:

$sumS_c = 28 + 104 + 78080 + 16 = 78228 * 8 = 625824 bits$

$sumS_p = 16 + 40 + 8576 + 88 = 8720 * 8 = 69760 bits$

Now the number of transactions performed on the PCIe bus per second can be calculated:

$numT_c = (PCIeSpeed * 1G)/sumS_c = 16000000000/625824 = 25566$

$numT_p = (PCIeSpeed * 1G)/sumS_p = 16000000000/69760 = 229358$

The speed for this part of the transaction will be dictated by the lowest number of transactions, as that will be the limiting factor. Thereafter, the PCIe bus speed will be limited by the traffic originating from the CPU.

$PCIeBusSpeed = (MTU * numT_c) * 8/1G = (65536 * 25566) * (8/1G) = 13.40Gbps$

Repeating the procedure for the **RapidIO network side**:

$sumS_p = 70654 * 8 = 565232bits$

$sumS_r = 1024 * 8 = 8192bits$

$numT_p = (RIOSpeed * 1G)/sumS_p = 16000000000/565232 = 28306$

$numT_r = (RIOSpeed * 1G)/sumS_r = 16000000000/565232 = 1953125$

Following the same reasoning, traffic originating from the Tsi721 is limiting the speed on the RapidIO network.

$RapidIONetworkSpeed = (MTU * numT_p) * (8/1G) = (65536 * 28306) * (8/1G) = 14.84Gbps$

It is clear that the PCIe bus side is considerably slower than the RapidIO side, shaping the maximum DMA Tx speed at **13.40 Gbps**.

DMA Rx

On the **RapidIO network side**:

$S_r = 70654 * 8 = 565232bits$

$S_p = 1024 * 8 = 8192bits$

Which give:

$numT_r = (RIOSpeed * 1G)/S_r = 16000000000/565232 = 28306$

$numT_p = (RIOSpeed * 1G)/S_p = 16000000000/8192 = 1953125$

The lower one dictates the maximum speed on this portion of the network:

$RapidIONetworkSpeed = (MTU * numT_p) * (8/1G) = (65536 * 1953125) * (8/1G) = 14.84Gbps$

On the **PCIe bus speed**:

$S_p = 77824 * 8 = 622592bits$

$S_c = 4096 * 8 = 32768bits$

Yielding:

$numT_p = (PCIeSpeed * 1G)/S_p = 16000000000/622592 = 25700$

$numT_c = (PCIeSpeed * 1G)/S_p = 16000000000/32768 = 488281$

And the speed for this part of the transaction:

$PCIeBusSpeed = (MTU * numT_p) * (8/1G) = 13.47Gbps$

As before the PCIe side shapes the speed of the DMA Rx operation at **13.47 Gbps**

Conclusively, the PCIe bus is always outperformed by the RapidIO network irregardless of transaction type. However the previous calculations were made with the assumption that the PCIe Max Completion size for our system was 128 Bytes. The next table also contains the respective results for a system that supports 256 Bytes of PCIe Max Completion size.

	128 Bytes	**256 Bytes**
DMA Tx Speed	13.40 Gbps	14.55 Gbps
DMA Rx Speed	13.47 Gbps	14.63 Gbps

Table 2.1: PCIe Max Completion Size - Speed correlation

Supporting Materials

In figures 2.7 and 2.8 the sequence diagrams for the DMA Tx and DMA Rx scenarios, respectively, are given. The objects in these diagrams are comprised of:

- The main memory and CPU [CPU]
- The Tsi721 chip [Tsi721]
- The RapidIO network, including endpoints and switches [RIONET]

It is worth noting that in the CPU side of the Tsi721 all transactions are PCIe, while on the RIO network side all transactions are RapidIO.

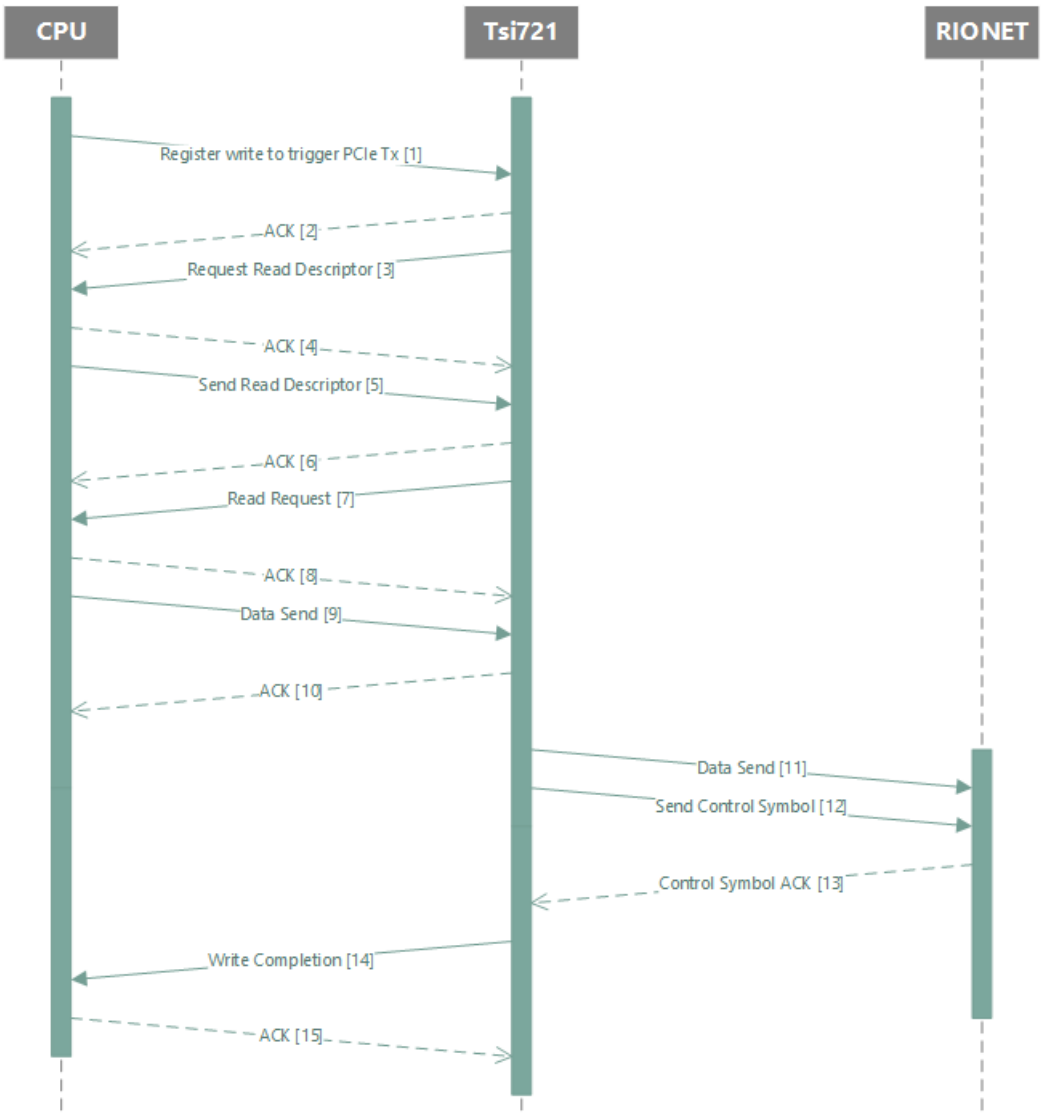

Figure 2.7: Sequence diagram for DMA Tx

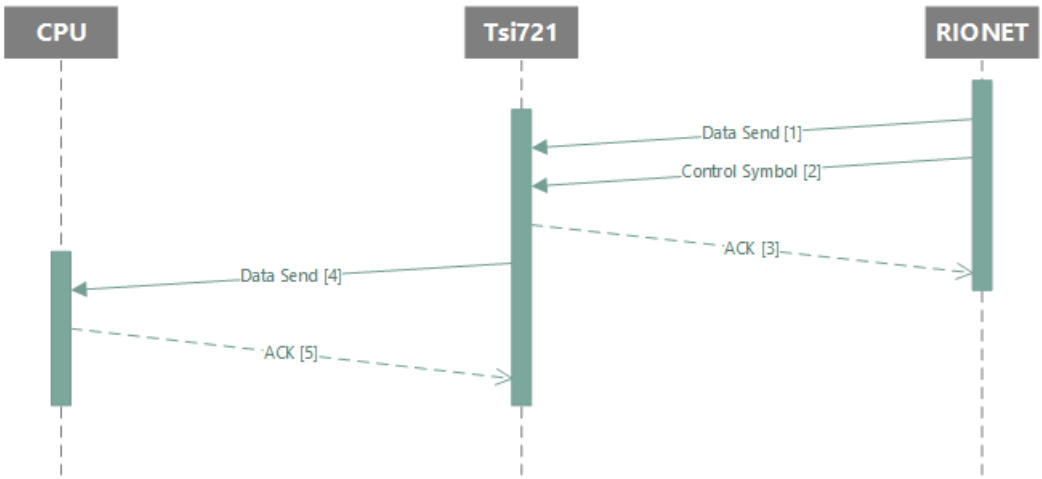

Figure 2.8: Sequence diagram for DMA Rx

SWRITE RapidIO Efficiency	Bytes
Physical Header	2
Transport Header	2
Address	6
Data	256
CRC	4
PAD	2
Total	272
Efficiency	94.12%

Table 2.2: RapidIO Packet

PCIe Efficiency	Bytes
Address Size	64
Header	16
MAX Completion Data	128
LCRC	4
ECRC	4
Total	152
Efficiency	84.21%

Table 2.3: PCIe Packet

	Bytes
PCIe Read Request	24
PCIe Register Write Overhead	20
PCIe Register Write Payload	8
PCIe Acknowledge	8
PCIe Credit Update	8
PCIe Read Completion Overhead	24
RapidIO Control Symbol	4
RapidIO MAX Payload	256
MAX Completion Data	128
Descriptor Size	64
MTU	65536

Table 2.4: Miscellaneous Values

2.2.5 Experimental Approach

We can check the above results, by measuring time around the dma_write and dma_read operations. 512MB of data were repeatedly sent, scanning through increasing DMA sizes. The DMA buffers tested ranged from 4K up to 256MB, which was the size of the reserved memory of the running system, and thus the maximum possible DMA buffer to be used. The result of the plot can be seen in figure 2.9.

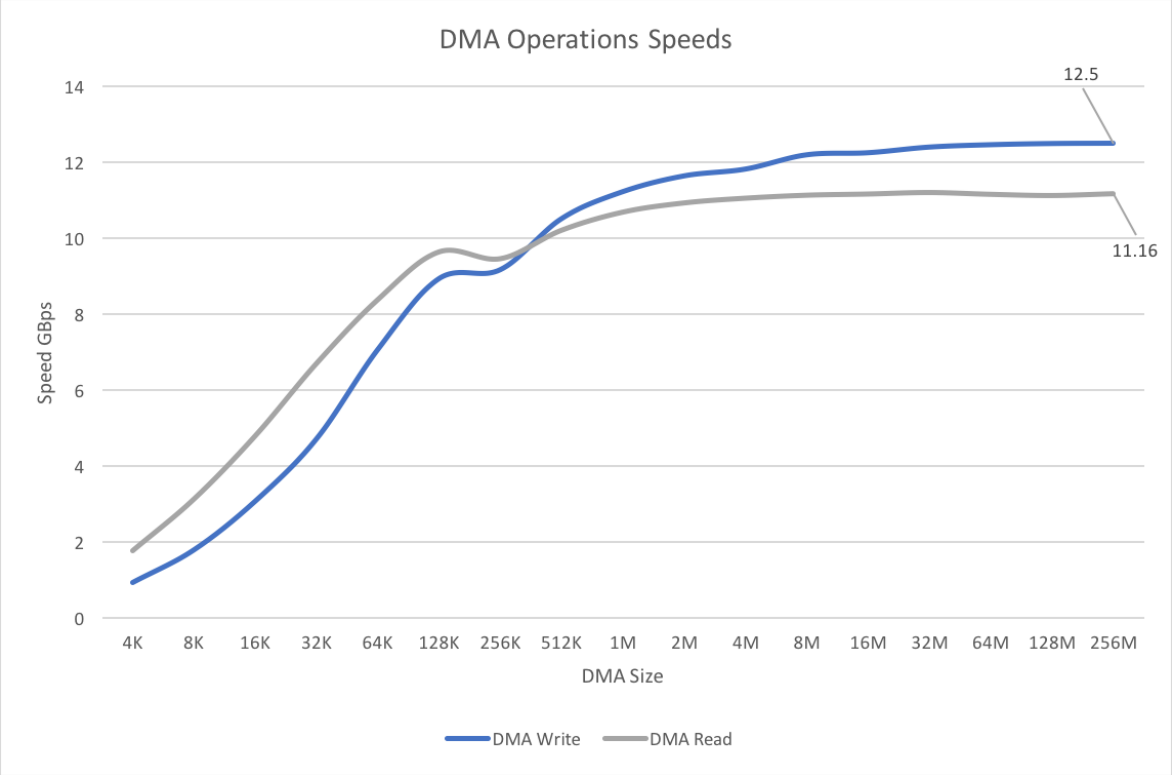

Figure 2.9: Speed of DMA operations

The figure above shows that at approximately 8MB a plateau is reached of around 12.5 Gbps for the Write operation, while the read operation is bound at 11.16 Gbps. These speeds dictate the overhead introduced by the library and the driver, which is presented in table 2.5

	Write	Read
Theoretical	13.40 Gbps	13.47 Gbps
Experimental	12.50 Gbps	11.16 Gbps
Overhead	1.10 Gbps	2.31 Gbps

Table 2.5: DMA Operations Speeds

2.2.6 Software Stack

The RapidIO software stack available today, consists of Linux kernel drivers [12] and the RRMAP libraries [13], developed by IDT. The software package is under active development.

User-space programs interface with the interconnect through library calls, which in turn interface with the underlying driver. Library calls will handle all management and transaction operations.

The transport layer messages are called Channelized Messaging within this stack. Library calls which are responsible for the following operations, are available; Initializing a "socket" which is related with a specific mailbox channel. Listening for connections to that socket. Accepting connections on that socket. Connecting to a remote endpoint's socket at a specific channel. Sending and receiving in the context of a given socket. Channelized Messages will always consist of buffers of 4096 bytes. Of these 2o bytes are required for protocol overhead, lowering the payload size to 4076 bytes.

As discussed before, doorbells carry the source device ID, the destination device ID and a 16-bit user-defined value field. Additionally, a callback function has to be registered in the event of a doorbell. Before an endpoint can accept and serve incoming doorbells a value range has to be allocated. Unless an incoming doorbell belongs that range, the doorbell is dropped. All these operations are executed through specific functions offered by the software stack.

RDMA operations can take place with arbitrarily large memory spaces. However these spaces are dependent to the size of the reserved RDMA memory, discussed in the protocol section. When executing an RDMA transaction, one end will have to make a library call, and one a simple memcpy. This depends on the way the connection was set up, and is essentially interchangeable.

2.3 ZeroMQ

ZeroMQ, or 0MQ, is a high-performance asynchronous messaging system. Message-oriented middleware is used in diverse settings where distributed or concurrent systems are present, ranging from financial services to physics simulation. ZeroMQ is used as a messaging library, that allows for the different components of the system to exchange messages between them. These messages can be arbitrarily larger, ZeroMQ is oblivious of the payload it transfers and thus imposes no limitations on possible applications. Unlike other messaging systems, ZeroMQ is a library, meaning that it does not employ the use of a message broker. This lowers the complexity and lifts the need for maintenance.

While initially ZeroMQ was targeted towards stock trading, focus shifted to make it a generic system supporting application in distributed and concurrent contexts. The library has found use in a wide number of applications in organizations like Cisco, NASA, Samsung and CERN. ZeroMQ is an open source project and has been forked, see JeroMQ, Crossroads I/O, which later evolved to Nanomg.

ZeroMQ has two attractive qualities. The first one revolves around performance. The absence of a server in the middle has a direct impact on performance, as a message's path is essentially halved. It does not need to go from the sender to the broker and then from the broker to the receiver. The second one has to do with usability and employment. By eliminating the need for a broker the effort the set up the system and run it falls significantly. Additionally, the ZeroMQ API closely resembles Berkeley sockets, which makes setting up a distributed system with certain requirements easier. An example would be employing a server that would route messages from group A to group B according to a set of rules, without burdening components from the two groups with the respective logic.

The internal architecture of the ZeroMQ library is abstracted in a way that it lends itself well to porting to other technologies, without interfering with other parts of the system. [14]

2.3.1 Internal Architecture of ZeroMQ

Context

ZeroMQ uses the Context class, `ctx_t`, for holding the global state of the library. This is created by the user explicitly, as the first action when setting up the infrastructure. The context holds information about sockets, I/O threads and endpoints.

Concurrency Model

ZeroMQ is a multithreaded application with each object living in its own thread. In order for two objects to communicate with each other, commands (not to be confused with user messages) will be exchanged. This eliminates the need for any concurrency orchestration to be regulated through locks. Consequently no mutexes, semaphores or conditional variables can be found within ZeroMQ. In order to be able to exchange commands, a class has to be derived from the `object_t` class. Available commands can be found in `command.hpp`. Commands can also carry arguments.

Threading Model

There are two ways to approach ZeroMQ threads, from the OS's point of view and from ZeroMQ's point of view.

The OS sees two kind of threads. The first kind is "application threads". These threads are created outside ZeroMQ and are used to access the API. The second kind is "I/O threads". These are created inside a ZeroMQ context and are utilized to perform send and receive operations in the background. For threads, the OS-agnostic portability class `thread_t` is used.

From ZeroMQ's perspective, any object that has a `mailbox` is considered a thread. The mailbox is the struct implemented in the `mailbox_t` class that is used in order to queue the commands destined to any object currently residing in said thread. The commands are processed in a sequential manner.

However, both I/O threads as well as sockets have a mailbox. Each I/O thread corresponds to an OS thread, and has a single mailbox to process incoming commands. On the other hand, multiple sockets can be residing in a single OS thread, or, in certain cases, migrate between OS threads.

I/O Threads

I/O threads are running in the background, and are responsible for handling network traffic in an asynchronous way. The `iothread_t` class is derived from `thread_t`, which was mentioned earlier. It is also derived from `object_t` enabling the exchange of commands with other objects for its orchestration.

Moreover, each I/O thread owns a poller object. The poller (`poller_t`) offers an abstraction for different polling mechanisms provided by the OS, such as poll and select.

Furthermore, each object living in an I/O thread is derived from the `io_object_t` helper class. An `io_object_t` allows for the registration of a file descriptor to the poller of the thread, with the `add_fd` function. This means, that when a file descriptor event takes place (e.g. POLLIN, POLLOUT) a callback will be invoked. Thereafter, every `io_object` implements the `in_event()` and `out_event()` functions, to handle file descriptor events. When the file descriptor is no more needed, it can of course be unregistered, using the `rm_fd()` function. Timers can also be registered in a similar manner.

Another significant remark, is that the `io_thread` itself will register a file descriptor with its poller. This file descriptor is the thread's mailbox file descriptor. When a command arrives, the poller will trigger an `in_event()` on the `io_thread_t`, which will proceed to forward the incoming command to the destination object that lives within it.

The following figure, see 2.10, presents an overview of the above.

In the above image the object X for example, may have registered a file descriptor with the poller. Let's assume that this file descriptor is the handle to a file. When someone performs a write to that file a POLLIN event will take place, which will be caught from the poller. Subsequently the poller will call the `in_event()` function of object X, to handle the event.

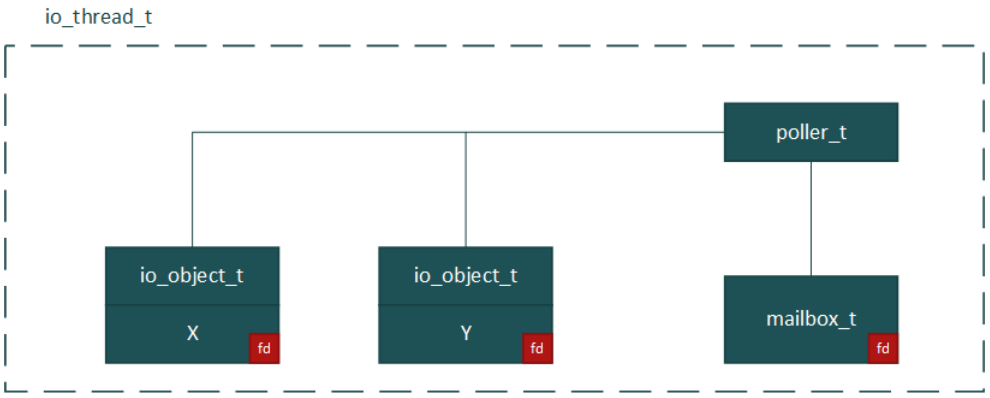

Figure 2.10: I/O Thread Overview

Object trees

The objects that are created within the ZeroMQ library are organized in a tree hierarchy, with its root always being the socket. As explained earlier, each object can live in a different I/O thread, with the exception of the socket which resides in an OS thread. The purpose of this design choice, is the provision of a deterministic shutdown mechanism. Generally speaking, when an object is asked to shut down, it will send shut down commands to all its children before shutting down itself.

In order to account for edge cases, like a simultaneous shut down decision from both parent and child, when an object wants to shut down, it will ask its parent to shut it down.

The object tree mechanism is implemented in the `own_t` class, which is derived from `object_t`, enabling the exchange of commands between the tree's objects. This `own_t` class also implements the function `launch_child()` which will associate the child object with the current I/O thread, effectively launching it.

Messages & Pipes

ZeroMQ fulfills complex requirements for its messages, which are scheduled through efficient implementations of pipes, contributing to its performance. However, these data structures are not of particular interest in the context of this thesis and therefore this matter is not inspected.

2.3.2 Overview & Critical Parts

The context is the first interaction of the user with ZeroMQ. It needs to be created so that it can assume ownership of the socket, or sockets, to be used.

The socket object lives in an OS thread. Depending on the socket's role, the socket will create a child that is either a listener or a connecter. The listener is the result of a *bind()* operation while the connecter is the result of *connect()* command.

The listener is waiting for connection requests on the bound interface/address. In case a connection request is successful the listener will create a session object, which will in turn create an engine object. Information is exchanged between socket and session through pipes. The engine object is the object that handles communication with the network. Session and engine live in an I/O thread, and are thus derived from *io_object_t*.

In an analogous way, the connecter requests a connection to an interface/address, which will result in the creation of a session/engine object in case it gets accepted.

The above are summed up in figure 2.11.

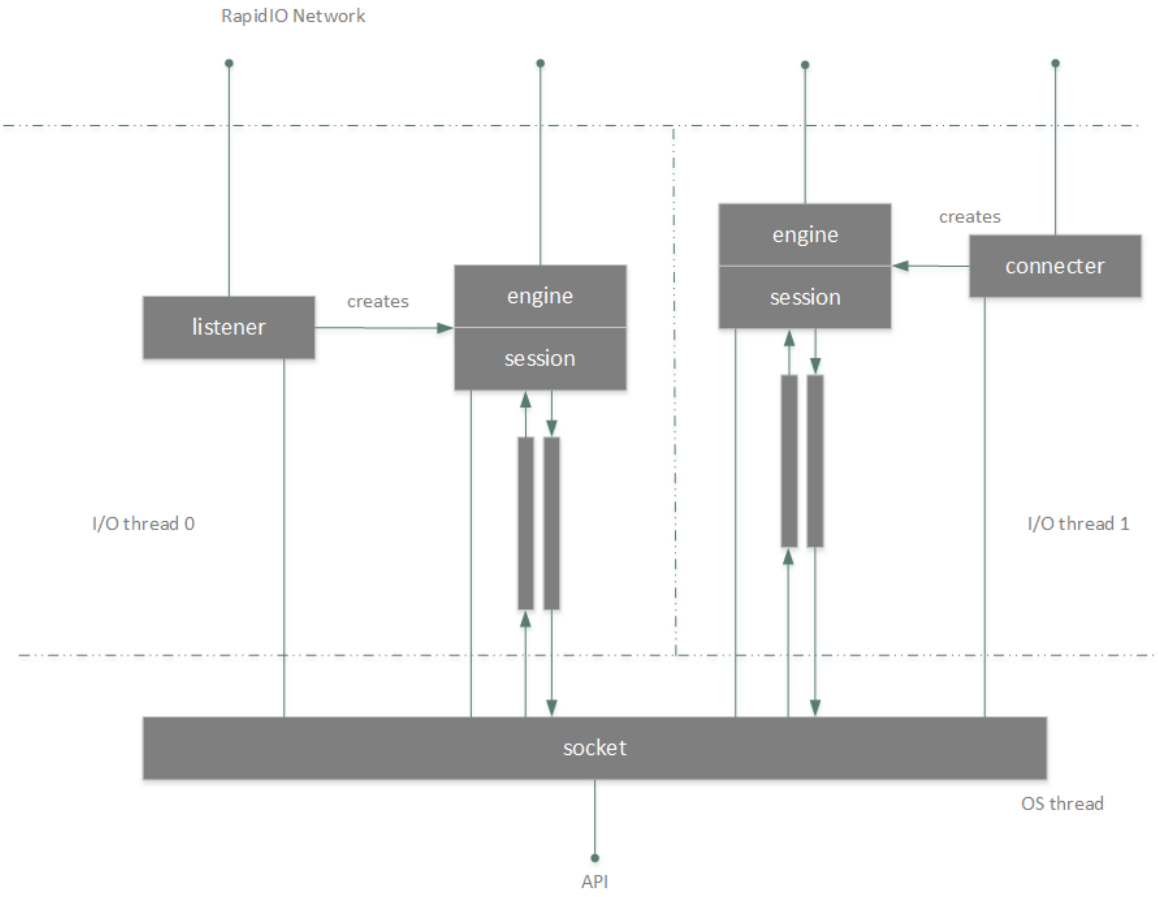

Figure 2.11: Internal Architecture

Chapter 3

Design & Implementation of the RapidIO extension of ZeroMQ

3.1 Introduction

The RapidIO extension of ZeroMQ, introduces a new transport layer to the messaging library. The purpose is to present a Proof of Concept for an abstraction layer around an RDMA-enabled interconnect, that allows the use of a well-established, socket interface.

This enables ZeroMQ users to take advantage of the benefits RapidIO has to offer, in an effortless way. Existing ZeroMQ users only need to change the address in their programs, in order to employ a different transport. The user does not have to be familiar with any RDMA-specific concepts nor with the RapidIO protocol. This eliminates the need for coding effort, allowing for the use of any structures and methods in place, as well as seamless swapping between transport layers.

3.2 Architecture Extension

In order to extend ZeroMQ to use the RapidIO transport, certain RapidIO-specific classes need to be implemented in addition to changes in some basic classes of the library's engine.

Below, a list of the core classes that need additions can be found, as well as a quick description of the necessary changes.

The Address Class

The ZeroMQ address class needs to know that an address could possibly be for RapidIO, and thus have a different format. This will allow for address resolution to be done by the appropriate class, in this case the RapidIO address class.

The Socket Class

The ZeroMQ socket class is called `socket_base`. Certain functions will be implementing by the socket type classes. However other functions like `bind()` and `connect()` share the same base implementation for every socket type and transport.

For binding, the base socket class will check the type of transport, and will accordingly create a listener object, passing the address as an argument. From that point forward the transport specific listener class is responsible for the rest of the procedure.

For establishing a connection, the socket class will correctly resolve the address, before creating a session object. That session object will be responsible for the new connection. It will also set up pipes between the socket and the newly created session according to transport restrictions.

The above operations had to be extended in the `socket_base` class to add support for the RapidIO transport.

The Session Class

The transport type-specific part of the ZeroMQ session class refers to the connection. The connection attempt is initiated indirectly when the socket class creates the session object. The session is then responsible for creating the appropriate connecter object, which will handle the rest.

If the address passed down to the session from the socket matches the RapidIO format, the `rio_connecter` object will be instantiated.

The following text describes the RapidIO classes that were implemented, to allow for ZeroMQ to exploit the RapidIO transport. To fulfill the requirements imposed by the architecture of ZeroMQ the following types of classes need to be implemented:

- An address class
- A listener
- A connecter
- An engine

On top of these four classes, a fifth one was implemented. The RapidIO mailbox's function is described in detail below.

The RapidIO Address Class

ZeroMQ mobilizes an address class is order to resolve an address, as this is passed down in the form of a string, from the API. Let's assume the user will use an address like "tcp://192.168.1.2:4545". This string contains three identifiers that are necessary. The first is the protocol identifier, i.e. tcp. This will instruct ZeroMQ to use the appropriate classes for the subsequent operations. The second and third identifiers are the IP address and the port, which of course translate to a specific endpoint in the system. This is covered, among others, by the `tcp_address` class in ZeroMQ.

In a completely analogous way, the `rio_address` class was implemented. For RapidIO, no strict addressing scheme is specified, like there is for TCP/IP. An endpoint is made up by a pair of a `destination id` and a `channel`. In order to retain a resemblance to the TCP/IP addressing, a RapidIO address in a ZeroMQ context is expected to have the following form: **rio://[destination id]:[port]**. This will then be parsed by the `rio_address` class, which will resolve the address, holding the relevant identifiers. Other parts of ZeroMQ will use this address class to get endpoint information, whenever needed.

The RapidIO Mailbox

ZeroMQ has mailbox classes, as these are described in Chapter **??**. The `rioh_mailbox` class is in no way related to them. Its purpose is to simulate file descriptor events, which normally manifest as a result to socket operations. For this simulation to take place, the `rioh_mailbox` spawns a dummy file descriptor, which is associated with the respective object. Such objects would be the listener, the connecter or the engine, in other words an open connection. An event on this file descriptor is triggered through a doorbell, whose value is uniquely tied to the descriptor. The simulated fd event will then be picked up wherever was anticipated. For this scheme to operate successfully the `rioh_mailbox` class holds the dummy file descriptor and the destination ID of the related endpoint. For the doorbell handling, it runs a background thread to catch and process incoming doorbells, if necessary. Functions to send a doorbell to remote `rioh_mailboxes` are also available through this class.

More details and a figure which help understand the above can be found under section 3.3.

The RapidIO Listener

The listener class in ZeroMQ is the one that is responsible for listening to incoming connection requests, approving them and creating a session/engine pair for every new connection. The listener will initially create a RapidIO socket and then start listening on that socket. The first event that will then take place, will be the arrival of a doorbell, which will be handled by the `rioh_mailbox` of the listener, triggering a file descriptor event on the fd associated with the listener, in turn triggering an `in_event` on the listener. There, necessary initializations for the new connection will be made, before spawning the linked session/engine pair.

At this point, it necessary to stress the following points. The RapidIO socket that is created here only supports Channelized Messaging. The RapidIO subsystem uses sockets like this to set up the connection and trade necessary information like target RDMA addresses. After this kind of information has been exchanged a bound socket only serves subsequent connection requests,

repeating the same handling process, until the user cleans up before exiting the program.

The RapidIO Connecter

The connecter class is responsible for connecting to the remote end. As a first step it creates a RapidIO socket, which will serve as the socket for the new connection. Next, it will send a doorbell to the listener, so that it triggers an `in_event`, entering an accepting state. After that, the RapidIO connect operation will take place. Following this, the relevant initializations will be made, before creating the session/engine pair for the connecting part. After that, the connecter will exit.

Same as before, the RapidIO socket is a Channelized Messaging socket, that is exclusively used for connection setup.

The RapidIO Engine

The engine is the part of the system that interfaces with the RapidIO network for any data transfer needs. It will handle sending and receiving data, assuming both roles, regardless of whether its creation was the result of a connect or an accept operation.

On send, the engine will perform an RDMA write, directly to the memory of the remote end. After the write call has returned, the engine will send a doorbell to the remote engine, in order to trigger a file descriptor event on the remote end. This procedure takes place within the `out_event` function.

In order to receive, the engine utilizes a thread, as part of the `rioh_mailbox` object which is constantly waiting for a doorbell, in order to trigger a file descriptor event. This triggering will cause the callback of the `in_event` function, so that the engine can read the newly written data.

The above read/write operations need to be done under a very strict orchestration, so that memory corruption does not occur. These matters are discussed in the next section.

The RapidIO Helper Functions

Certain functions and data structures that are common for some or all of the above classes are concentrated in a single header file. The `rio.hpp` file includes functions supporting the necessary actions, to successfully initialize a new RapidIO connection, for both the listening and the connecting side. These actions include the following; Initialization of the Channelized Messaging socket, the allocation of RDMA-enabled memory and the exchange of the inbound and outbound memory addresses that are a result of this operation. The header file is also provides functions for the allocation and the proper allotment of doorbell ranges. A struct carrying the necessary information about a RapidIO connection is also defined in this file. It is this struct that carries crucial information about the new connection, which is passed down from the listener/connecter to the engine. Proper shutdown functions are also part of this file. Lastly, various global RapidIO-specific values, such as the RDMA reserved memory base address, its size, the size of the DMA cell size or the length of the circular buffer are also defined here.

3.3 Implementation Details

File Descriptor Event Simulation

The biggest challenge in porting ZeroMQ to use RapidIO is the inherent need of the former for file descriptor events. The ZeroMQ framework is strongly tied to file descriptor events for triggering virtually any operation. This need stems from the use of sockets for an endpoint representation for any TCP/IP connection. Since sockets are essentially treated as open files the file descriptor describing them is prominent across the ZeroMQ codebase.

Fundamentally, an RDMA-enabled interconnect does not provide, support nor need a socket interface and by extension a file descriptor. This is also the main problem for the use of RDMA today. Existing applications do not provide a fitting place for importing a new, non-standard, transport layer. This is, as well, the case for RapidIO. A socket interface is not provided, introducing the need to simulate file descriptors if it is to be incorporated in a standard

setting.

In the scope of the current extension, this complication is encountered in the following way. The linux kernel offers a way to create a ”dummy” file descriptor which can be used as a wait/notify mechanism, through the eventfd(2) call. eventfd() returns a file descriptor on which the write(2) and read(2) operations can be performed, respectively increasing and decreasing a counter of the eventfd object which is maintained by the kernel. Through this counter the number of file events can be tracked and, in a way, queued, effectively creating a dummy file descriptor, over which we can have full control. This allows us to exploit it and use it interchangeably as a “socket interface” for our needs.

Consequently, in order to simulate a file descriptor event, we can simply send a doorbell to the concerning endpoint, which will already have a doorbell handler in place. The handler will process the incoming doorbell event, and according to the value communicated, a write(2) operation will be performed for the corresponding file descriptor. This will trigger a file descriptor event, which will be handled by the poller (poll(2)) object, with which the file descriptor is associated. Afterwards, the poller will issue a callback, on par with the codebase's logic.

Doorbell Handling

In order for the file descriptor events described in the previous subsection to function correctly, doorbell operations are implemented as part of the RapidIO Mailbox class, as described before. Each of the connecter, listener and engine classes will initialize a RapidIO Mailbox on creation. The newly created mailbox will create a file descriptor through an eventfd(2) call, which will act as the class' file descriptor. On top of that the `rioh_mailbox` will create a thread which will run in the background as the doorbell handler, catching incoming doorbell events and taking the necessary action. Every class will check the doorbell's source and payload. If any one of those two are not correct, the doorbell event will be ignored.

When the `rioh_mailbox` of the listener receives a valid doorbell, it will perform a write(2) to the file descriptor, which will subsequently trigger the `in_event()` function of the `rio_listener`. In the callback, the `accept()` call will take place

for the new connection.

The `rio_connecter` is never expected to receive a doorbell. Its `rioh_mailbox` is only used to send a doorbell before issuing a connection request. It does not create a thread listening for doorbells, and does not enable any doorbell range.

The doorbell handling for the `rio_engine` is tied to the memory handling logic and will be discussed in the next subsection.

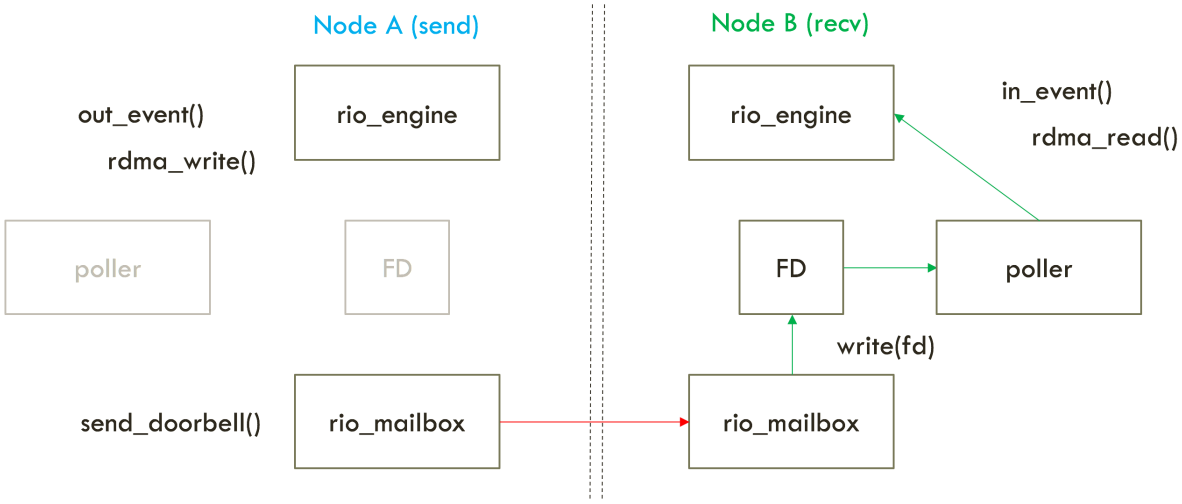

Figure 3.1: Mailbox functions overview

Doorbell Allocation

Doorbell events carry the following information; The destination ID of the source endpoint, the destination ID of the target endpoint and a `uint64_t` value. Since different components from the same endpoint are listening for doorbells, it is important to have a mechanism to differentiate between them. In the next paragraphs the scheme for the doorbell range allotment is presented.

The listener object will only care about a doorbell with the proper target destination ID carrying a value of `RIO_ACCEPT`. The `RIO_ACCEPT` is defined as a magic number in the `rio.hpp` header file. A doorbell with the same value should not be observed in any other scenario, except from when a connection attempt is being made. The connecter object follows the same rule, but expects a value of `RIO_CONNECT`, which is also defined in the `rio.hpp` file. In the current implementation these two magic numbers are the first two numbers, namely 0 and 1, in the available doorbell range.

For the RapidIO engine, things are more complex. The ZeroMQ instance must be able to hold many parallel connections open at the same time. Every one of those connections should allow for the transfer of data even if some or all of the other connections are operating. This part is tied to the memory handling logic which is presented just below. The RDMA memory is initially divided into as many parts as there are possible connections. In other words if the system consists of four nodes, one node (A) should have incoming memory for the other three (B, C & D). Consequently the reserved memory on node A will be split to three, equal parts. Each part will then be further partitioned to individual RDMA cells, which are part of a circular buffer structure, also explained in the next part.

In order to support the scheme described above, doorbell ranges are mapped to memory positions in the following way. For node A the first connection established will always refer to the first of the three pieces of reserved memory. The engine object that will be created to handle this connection will drop any doorbells, whose values are not tied to the cells within that region. The same logic applies for subsequent connections and subsequent engine objects.

Doorbell values are calculated as follows. The initial values of 0 and 1, reserved for the listener and connecter, are skipped. The doorbell range for the first connection will start at 2. It will end at a value equal to the total positions of the circular buffer, divided by the expected number of connections. Applying the same logic every subsequent doorbell range can be calculated, as follows.

$$range\_start = prev\_range\_end$$
$$range\_end = prev\_range\_end + cbuf\_length/num\_conns$$
$$prev\_range\_end = range\_end$$

Where $cbuf\_length$ is the length of the circular buffer and num_conns is the maximum number of connections. For the first range group the initial value of $prev\_range\_end$ is 2, which is `RIO_CONNECT`$+1$. The actual allocation would be [$range\_start$, $range\_end$)

In order for this to become more understandable, let's assume that we have the following values:

num_conns	8
cbuf_length	16

In that case the doorbell allocation would be the following:

RIO_ACCEPT	[0]
RIO_CONNECT	[1]
CONNECTION_0	[2-18)
CONNECTION_1	[18-34)
CONNECTION_2	[34-49)
CONNECTION_3	[49-65)
CONNECTION_4	[65-81)
CONNECTION_5	[81-97)
CONNECTION_6	[97-113)
CONNECTION_7	[113-129)

Message & Memory Handling

When a ZeroMQ user wants to make a transfer, the relevant data will be passed, in a raw fashion, to a `zmq_send()` API call. The user will be sending a batch of data and will be expecting the same exact batch of data to be available at the remote end. For this subsection, this batch of data will be referred to as ”the message”.

When RapidIO is about to make a data transfer, it is concerned with the bytes that are available for sending, and making them available at the predetermined memory area of the remote endpoint. It is common for the message size to be larger than the available RDMA memory, introducing the need to splice the initial message to smaller parts, in order to carry out the transfer. It is also possible, that multiple connections are active at a given moment in time, introducing a second need to assign RDMA memory exclusively to each of them. It is thus necessary to implement a scheme that allows for such scenarios.

We begin with two RapidIO endpoints, that have the same amount of RDMA memory. When establishing the connection, the two endpoints exchange inbound memory addresses, so that every endpoint knows where to write. The

RDMA memory is split into an arbitrary number of RDMA buffers. Each `send/recv` operation will take place for the number of bytes that is equal to an RDMA buffer. Each engine object holds two arrays of flags. The first one represents whether an inbound memory position holds data that has been read or not. The second one represents whether an outbound memory position is available for writing. The engine also has two indexes, one for each of these arrays. Let's examine three occasions.

Firstly, let's assume that the message to be transmitted is smaller or equal the RDMA buffer size. In that case, the sender will perform a check to see if the current outbound index's slot is available for writing. If it is, the sender performs a dma write, updates its outbound array to mark the position as unavailable for writing and sends a doorbell to the receiver to inform it of the new data. After this action, the receiver will be "woken up" by the doorbell, which will be carrying the outbound index of the sender (i.e. the inbound index of the receiver), as its payload, and will update the respective flag in its inbound memory array. The receiver will then check whether the corresponding memory position is indeed set, and if it is, it will continue to read the data, before pushing the message to the RapidIO session, to be later return through the API `zmq_recv()` call. It is noted, that at this step, no memory copying takes place. The system component waiting for the data is simply passed the pointer to memory. Just after the data is "read", i.e. is no longer need, the receiver will send a doorbell back to the sender, so that the sender updates its records, making the just-read memory position available to write to again. However, this is not happening in sync. This allows the sender to "queue" subsequent data chunks, before the previous ones have been processed.

Now, let's assume that the message to be transmitted is larger than the RDMA buffer size, but not larger than the totally available RDMA memory. In this case, the message will be split into chunks. For every chunk the procedure outlined in the previous example, will be repeated. In this scenario, the asynchronicity of the design will be exploited, and subsequent chunks will not be limited by read speed. At some point in time, before the transaction is actually finished, all the data will be present at the receiving node, and the sender will have returned. This favours performance compared to a completely synchronized design.

Lastly, we have the occasion where the message to be transmitted is larger than the reserved RDMA memory. In that case, the operation sequence remains the same, but at some point the sender will possibly have to block, in case it is vastly faster compared to the receiver. Both parties will have to do a complete cycle through the circular buffer, possibly more, depending on the message size.

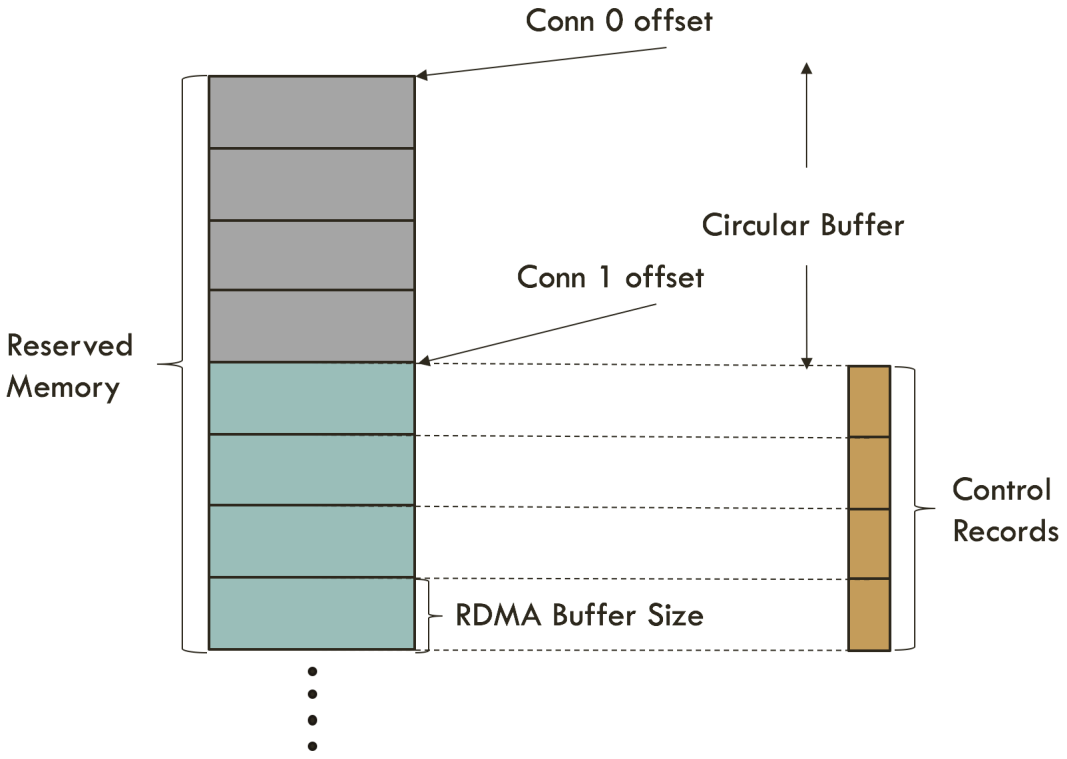

Figure 3.2: Memory Scheme Overview

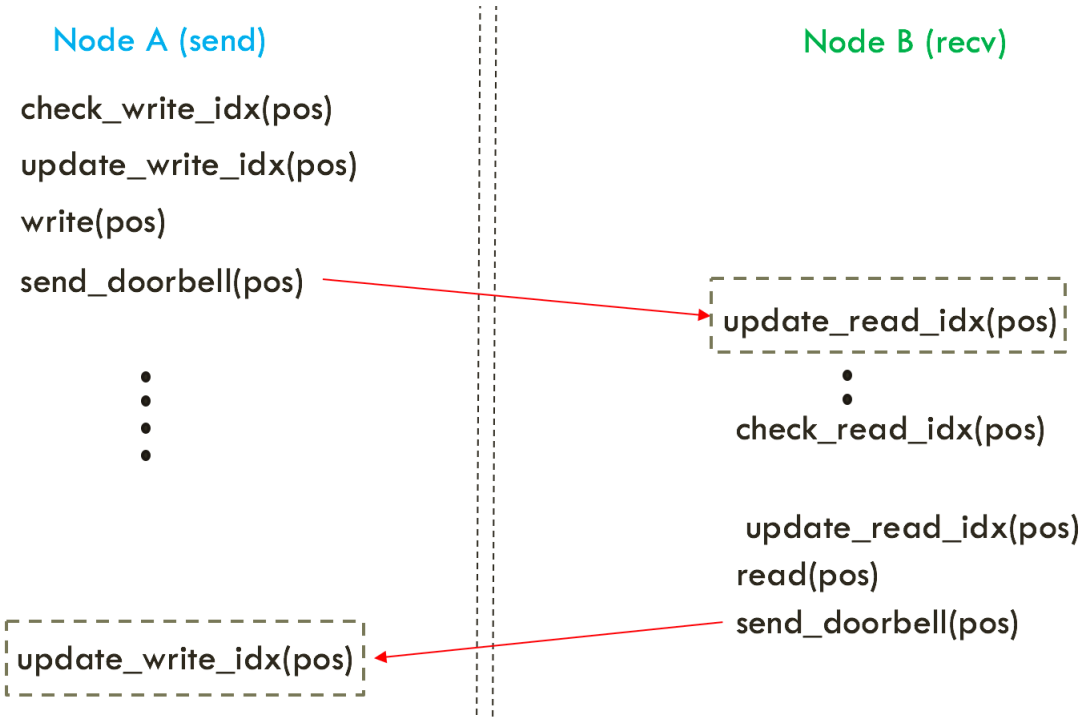

Figure 3.3: Sequence for single RDMA Buffer Transfer

Chapter 4

Evaluation

This chapter aims to evaluate the work presented in the previous chapters of this thesis. Initially the choices of benchmarks and the various analyses are explained, before presenting the respective methods. Next, the measurements are presented and an interpretation of the results is offered.

4.1 Hardware & Software Setup

The hardware setup used for the purposes of benchmarking consists of four 2U Quad units, each of them housing four nodes, totaling to 16 nodes. Each node is fitted with an Intel Xeon L5640, clocked at 2.27Ghz and 48GB of RAM. Every node is equipped with an IDT Tsi721 PCIe to RapidIO bridge card, with a nominal line rate of up to 16Gb/s. The NICs (Network Interface Cards) are in turn connected to a 38-port Top Of Rack (ToR) RapidIO Generation 2 switch box. The switch ports are configured to 20Gb/s. Connections are done using QSFP+ cables, or Quad Small Form-factor Pluggable cables, which are cables that enable the network interfacing of hardware to fiber optic cable, active or passive electrical copper connections. They support four channels of 10Gb/s, rendering them more than adequate for our needs.

The RapidIO software consists of the RRMAP libraries, developed my IDT, and the Linux kernel drivers.

4.2 Benchmarks & Measurements

For benchmarking, the ZeroMQ performance tests for latency were used. The two actors in these tests are the client, which will be the first one the perform a send operation, and the server, which will be the first one to receiver a message.

The server is run first. It initiates the ZeroMQ context and a socket with is of type "ZMQ_SERVER". Afterwards the socket is bound to the RapidIO address that consists of the destination ID of the receiving node and the channel to receive Channelized Messages to. Then, a ZeroMQ message is initiated, before making a receive call, which will block until the message is received, , an error occurs, or a timeout is reached. Afterwards, the message will be sent back to the client. This procedure will be repeated for as many times as specified in the program arguments. Finally, the sender will clean up, by closing the message structure, the socket and terminating the context. No measurements are taken on the server side.

The client is executed after the receiving socket has been bound. It will also initialize the ZeroMQ context and create a ZeroMQ socket of type "ZMQ_CLIENT". It will then attempt to connect to the network address of the server. If all goes well, the message structure will be created, before starting a timer and sending the message to the server. After the send call the sender will wait to receive the message back. The send/receive pair will be run as many times as specified by the program arguments. After all loops are finished, the timer is stopped. This provides the client with the time it took to send and receive a message of fixed size for a specific number of times.

If this elapsed time is divided by the roundtrip count and then by two, the round trip time of the system between sender and receiver, or A and B, is given. Furthermore, if we divide the round trip time by two, we get the latency of the system, in other words the time it takes for a fixed-size message to get from point A to point B. Of course, after the latency is known, the throughput can be calculated by dividing the message size with the latency.

In the next subsections, performance is evaluated while modifying different parameters .

4.2.1 Latency

Here, the latency of the system is evaluated, juxtaposed with the message size.

The results are presented in the graphs below. A line which represents the values taken from the same test over TCP with 1GbE is also part of the graph.

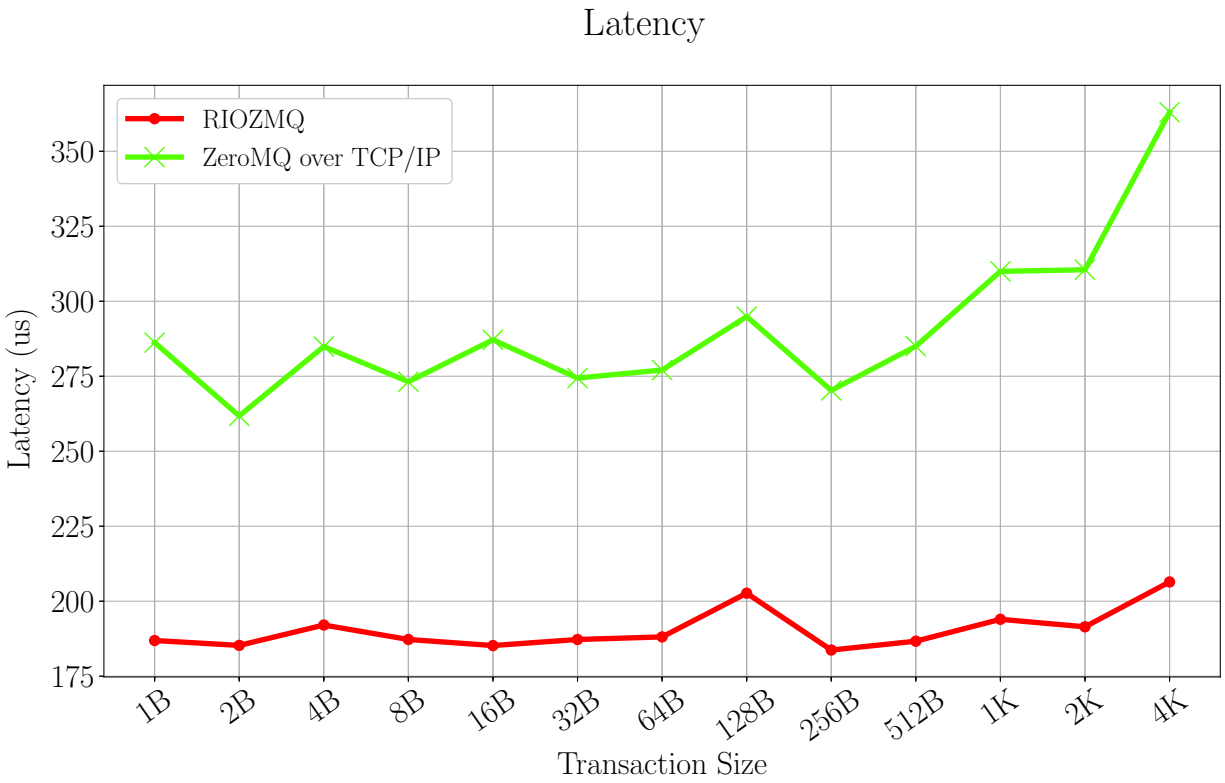

Figure 4.1: Latency over RapidIO and TCP/IP [Small]

The benchmark was run for a roundtrip count of 10000 times for each transaction size. The DMA Cell Size for RapidIO was set at 4KB, the smallest possible, using only one position of the circular buffer.

As we can see the performance of the system over RapidIO is faster compared to its TCP counterpart. This was expected, as the RapidIO protocol can achieve much better latency results compared to TCP. This is mainly due to the overhead difference between the two methods. TCP introduces a high complexity overhead, through the need to traverse the network stack, even for sending one byte, requiring context changes and consuming CPU cycles. RapidIO on

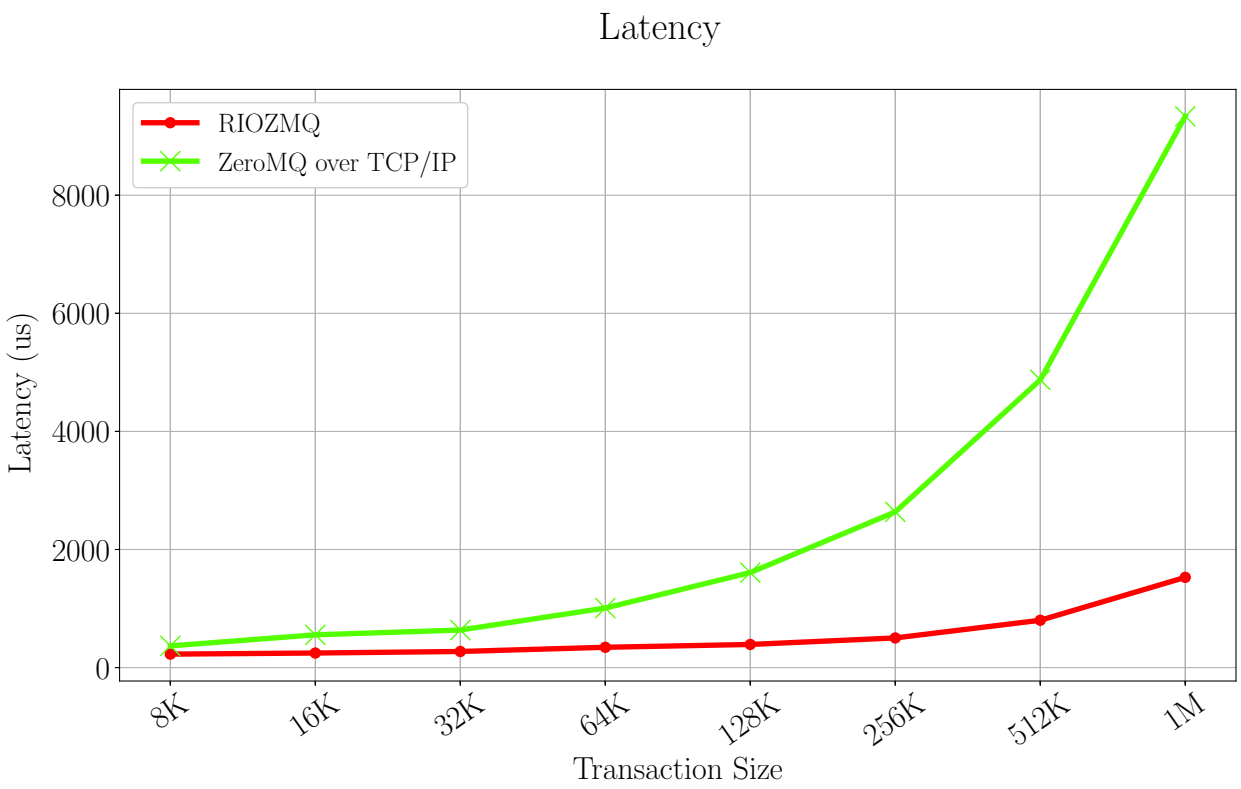

Figure 4.2: Latency over RapidIO and TCP/IP [Large]

the other hand just copies the byte to remote memory, bypassing the host's CPU.

Additionally, the rate at which the latency is increased relative to bytes transmitted is significantly lower for the RapidIO implementation, hinting at promising scaling capabilities.

4.2.2 Throughput

In the next graph, the throughput of the system against message size is presented. The figure refers to a one-to-one pair.

For each transaction size the benchmark was run with a roundtrip count of 20. The DMA Cell Size was set at 32MB, while the Circular Buffer length was 16. In other words, 512MB of reserved memory were cut at 16 chunks of 32 MB. This parameter combination was the most performing result, as it occurred from the circular buffer length and DMA cell size scan, presented later in the section.

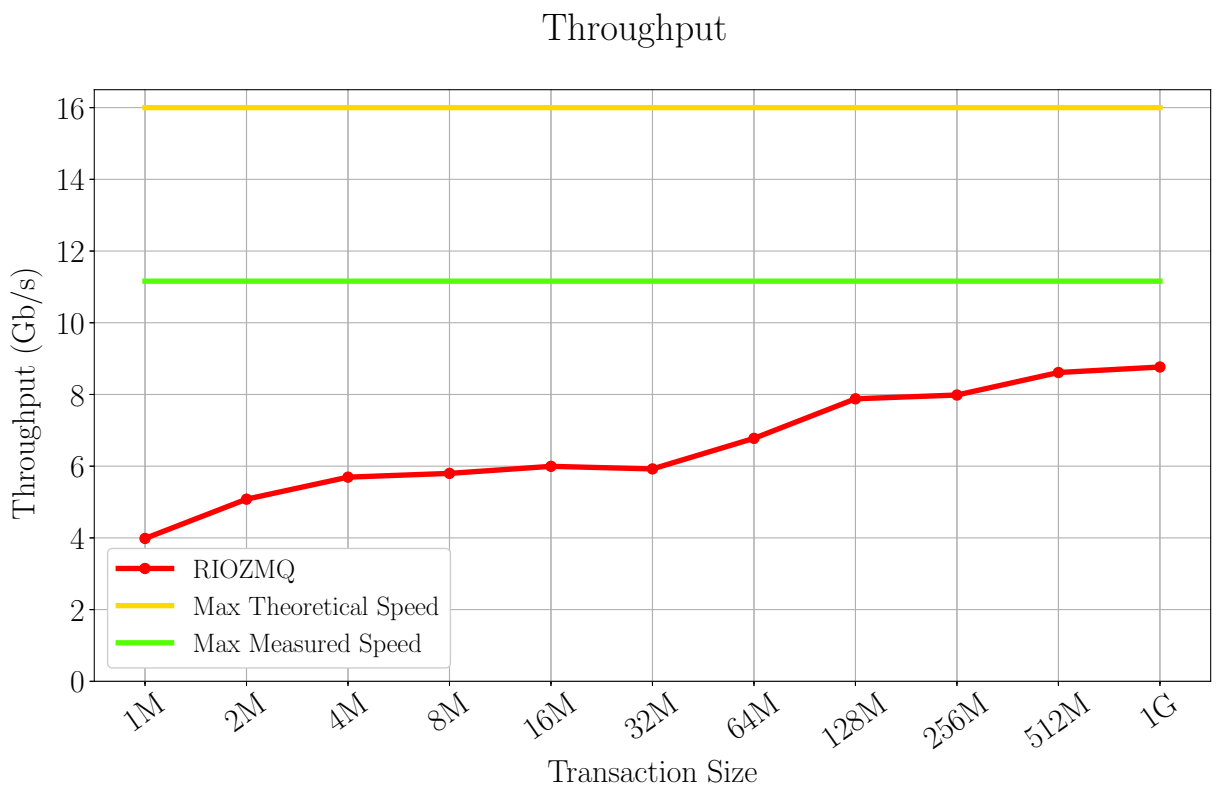

Figure 4.3: ZeroMQ Throughput over RapidIO

For every transaction, an overhead is introduced as part of the RDMA orchestration. This means that when a transaction is small, performance is heavily impacted by overhead. However, as size increases the impact of the overhead seizes to be so significant as the utilization of the line is better. This is also what is observed on the throughput graph above. We see that for large enough transactions (more than 512MB), throughput reaches a plateau of around 8.5Gbps. This is slightly more than half of the nominal link speed. However, as presented in chapter 2, after accounting from RapidIO and PCIe translations, and weighing the overhead of the kernel driver and the libraries, the expected practical speed should be around 11.1 Gbps. If this speed is regarded as a maximum, ZeroMQ over RapidIO can achieve around 77% utilization.

Throughput was also investigated in a many-to-one scenario. In the following scenario, a node is acting as the server, receiving transactions from multiple clients.

Each transaction was run with the same parameter combination as the previous

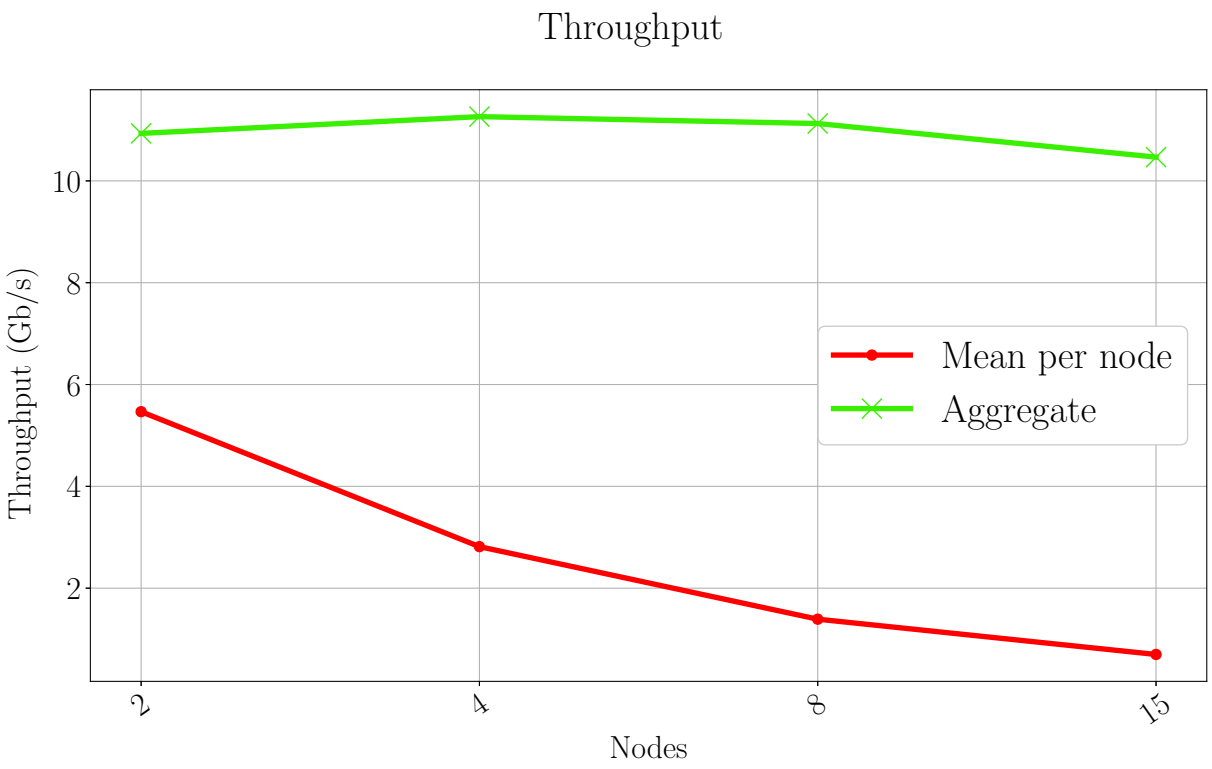

Figure 4.4: ZeroMQ Throughput over RapidIO - Multiple clients

throughput test.

When multiple nodes are communicating with one server, the nodes are fighting for bandwidth, on the link between the switch and the server. The red line we are seeing, is the mean speed across the reported speeds from all the clients. In other words, we see what the effective bandwidth *per client* is, in a many-to-one scenario. The green line is the aggregate speed, the sum of the speeds of all the nodes.

We can see that the system scales well, as the overall throughput of the system remains near constant, while the number of clients is increasing.

4.2.3 Circular Buffer Length

Graph 4.5 presents a scan of the circular buffer length parameter, which is part of the RDMA implementation for RapidIO.

For each circular buffer length, the benchmark was run with a roundtrip count of 20. The message size remained fix at 512MB, and the DMA cell size was

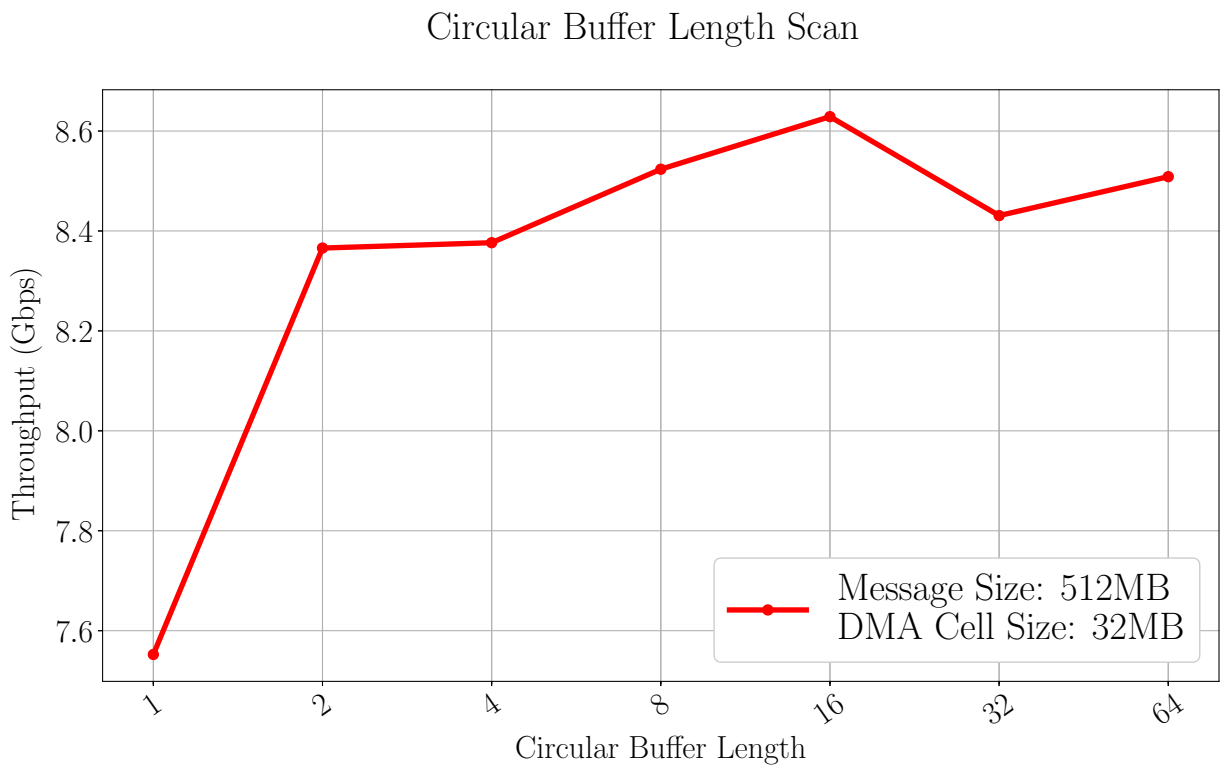

Figure 4.5: RapidIO Circular Buffer Length Scan

32MB for every occasion.

As can be seen from the graph, a significant increase in performance can be observed when increasing the circular buffer positions from one to two. This increase can be explained in the following way. Let's assume that the circular buffer consists of one position. A dma write operation is executed. The next one will have to wait, until the respective read operation is done, when the writer will be notified. This introduces a delay. This delay is instantly mended by adding another position to the circular buffer. Of course in this case, the third subsequent write operation will wait and so on and so forth. However, we see that for a length higher than two, the performance increase, if any, is considerably lower. The sweet spot appears to be between 8 and 16 positions. After that, it appears that the overhead added by position calculation starts hindering performance

4.2.4 DMA Cell Size

The plot depicted in 4.6 shows a scan of the DMA cell size, corresponding to one position of the circular buffer.

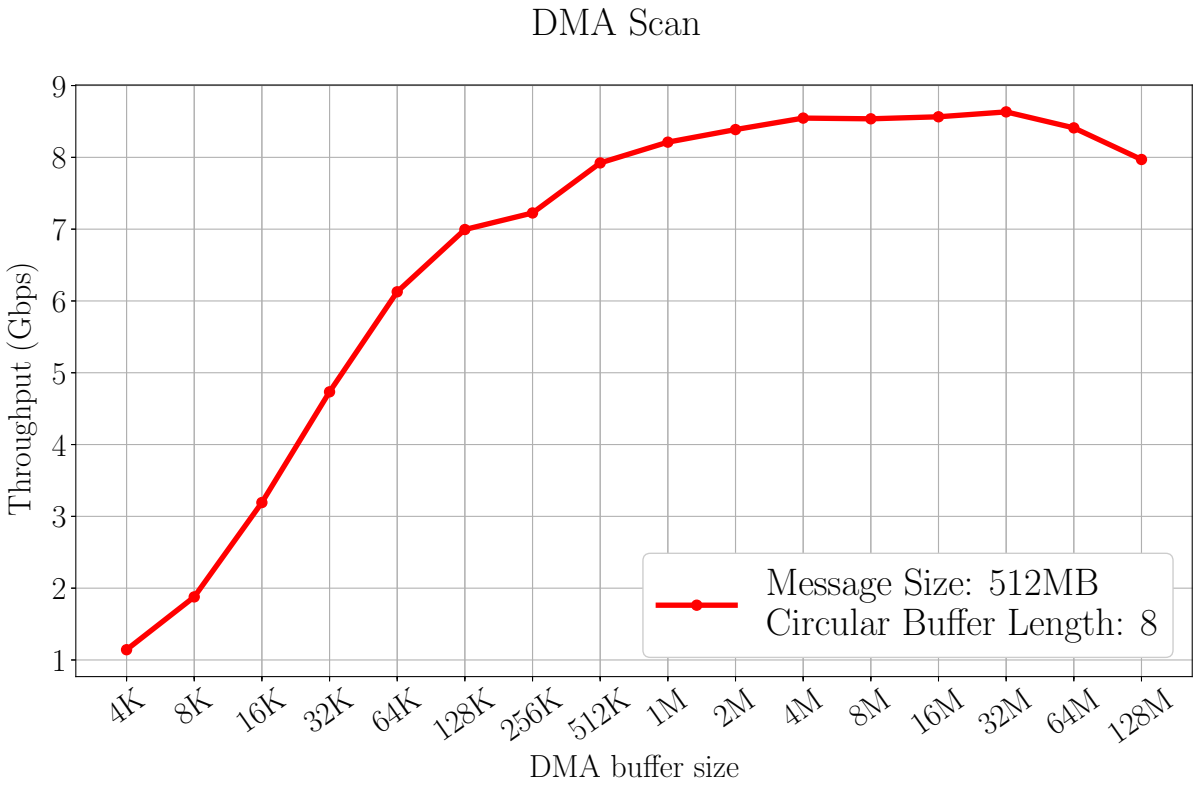

Figure 4.6: RapidIO DMA Cell Size Scan

For every DMA buffer size, the benchmark run roundtrip for 20 times. The message size here also remained fix at 512MB, employing a circular buffer of 8 positions.

As was expected, a larger DMA cell size yields better performance. For smaller cell sizes, the messages has to be cut into a higher number of chunks, introducing more overhead, and consequently hindering performance. The best outcome comes from DMA buffer sizes of 4, 8, 16 and 32 MB, with the last one being the best, but only slightly.

4.3 Breakdown Analysis

In this section an effort is made to understand the performance distribution across the system's components.

In order to do that, timestamps were taken in various locations across the critical path of the send and receive transactions.

4.3.1 Send

The results for the send operation and can be seen in figures 4.7 and **??**.

As is evident from the column graphs, the overwhelming majority of the time is consumed by the `dma_write()` operation. Apart from that, the `encode()` function also requires a lot of time. That happens because the ZeroMQ encoder class will do a `memcpy()` operation. As we can see, aside from the `dma_write()` and `encode()` operations, the impact of the others are trivial.

Conclusively, we cannot see any unexpected slowdowns.

4.3.2 Receive

The column graphs in 4.9 and 4.10 show the results for the receive operation.

Here the entry `in_event` is the one that takes the most time. This entry refers to the time elapsed between two subsequent `in_event()` calls. This time can contain the waiting time for a doorbell, doorbell handling, and the overhead from file descriptor operations. The `decode()` function also takes a lot of time as the messages get larger as it contains a memory copy operation, which requires resources.

Overall, this analysis produced two remarks. Firstly, the necessary file descriptor and doorbell operations introduce a notable delay. Secondly, the memory operations taking part in the encoder and decoder classes of ZeroMQ are an important bottleneck of the implementation.

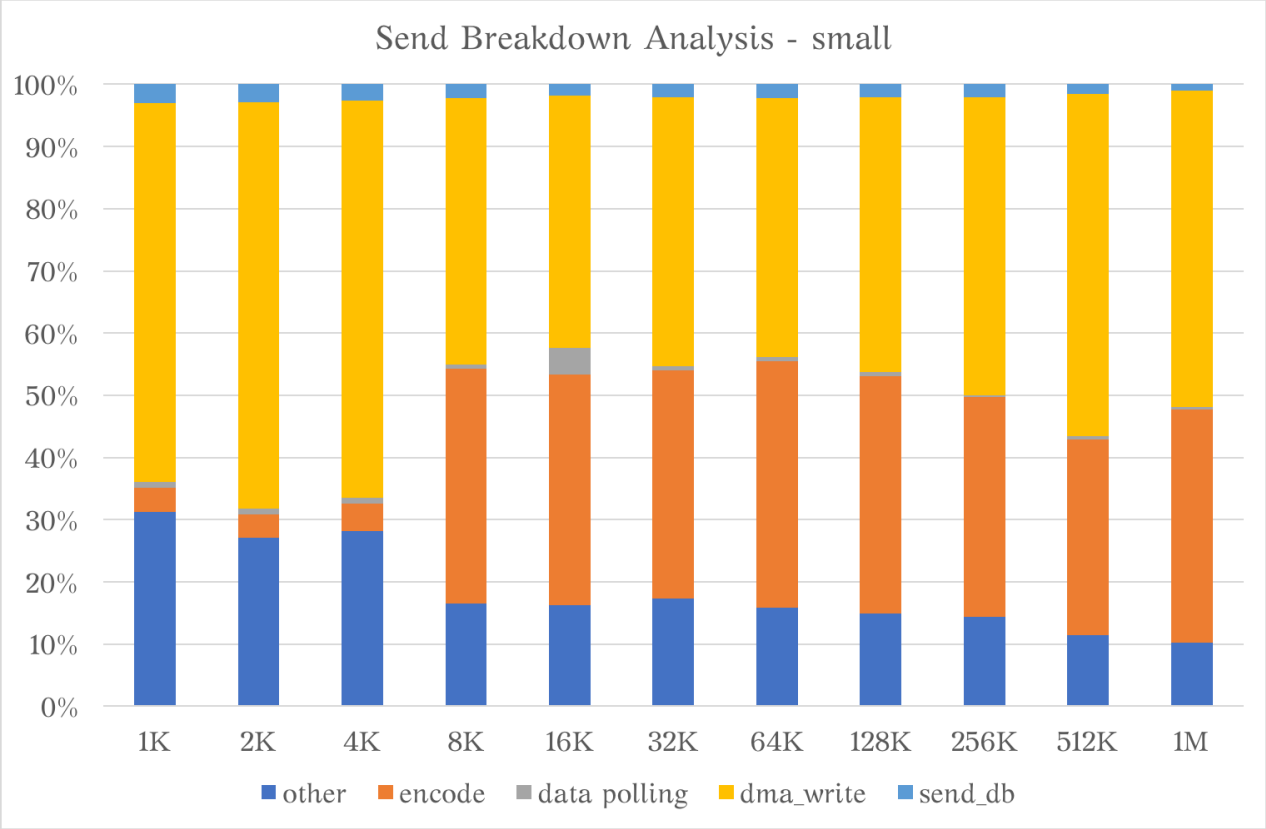

Figure 4.7: Breakdown analysis for the ZeroMQ sender [Small]

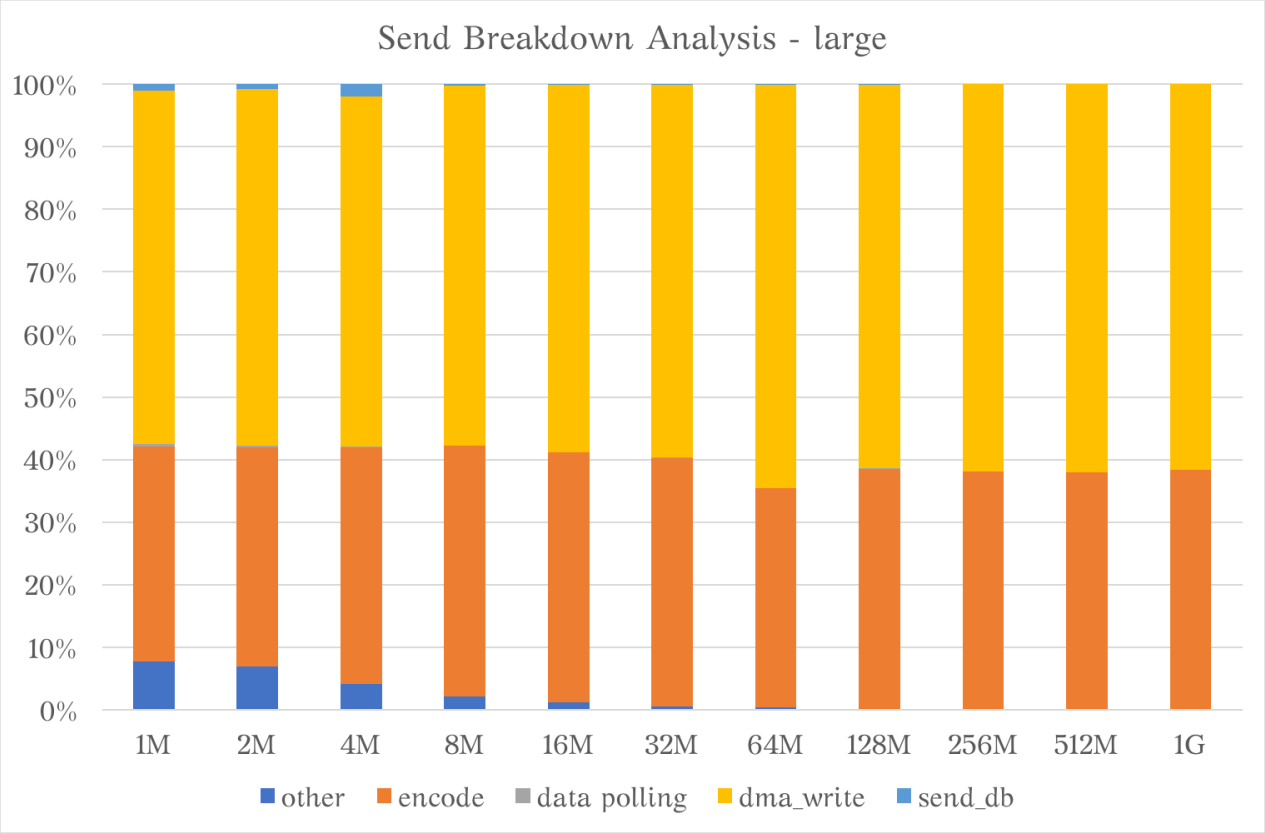

Figure 4.8: Breakdown analysis for the ZeroMQ sender [Large]

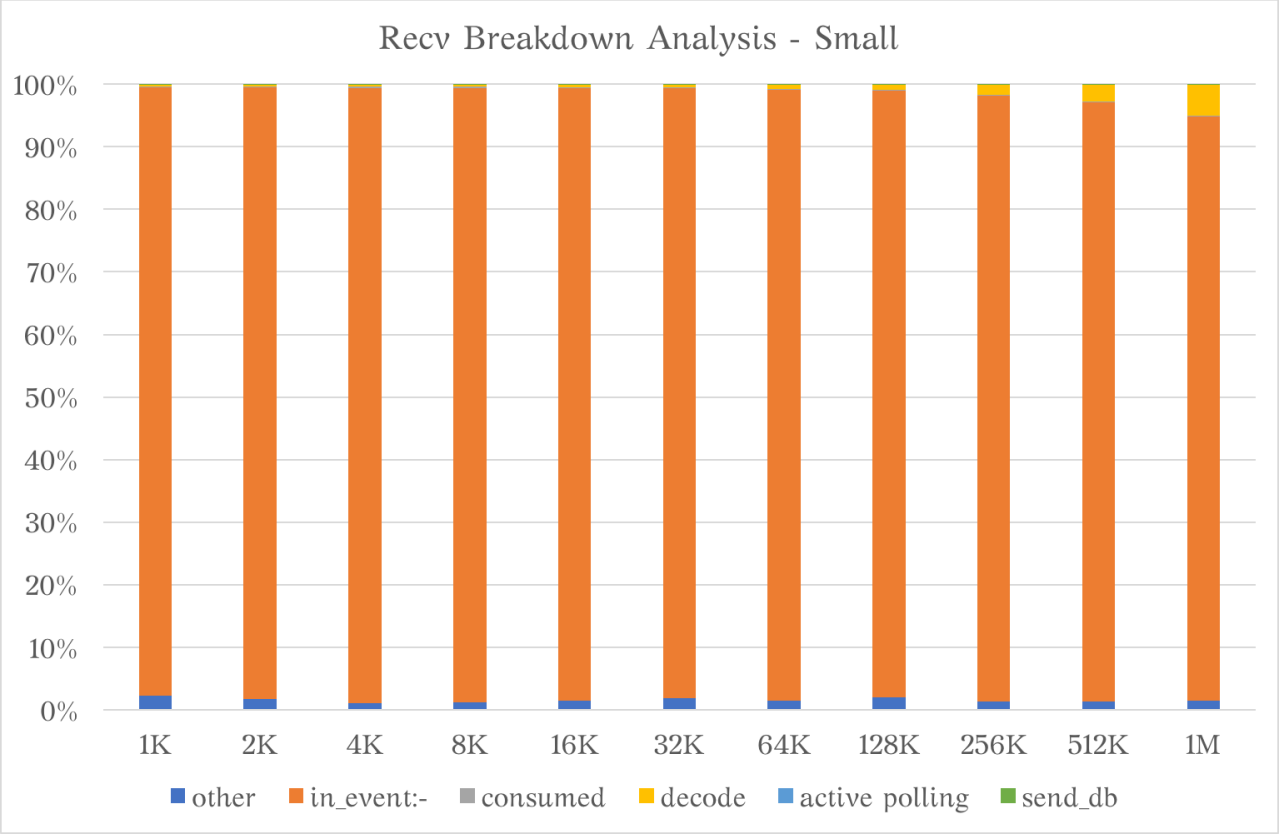

Figure 4.9: Breakdown analysis for the ZeroMQ receiver [Small]

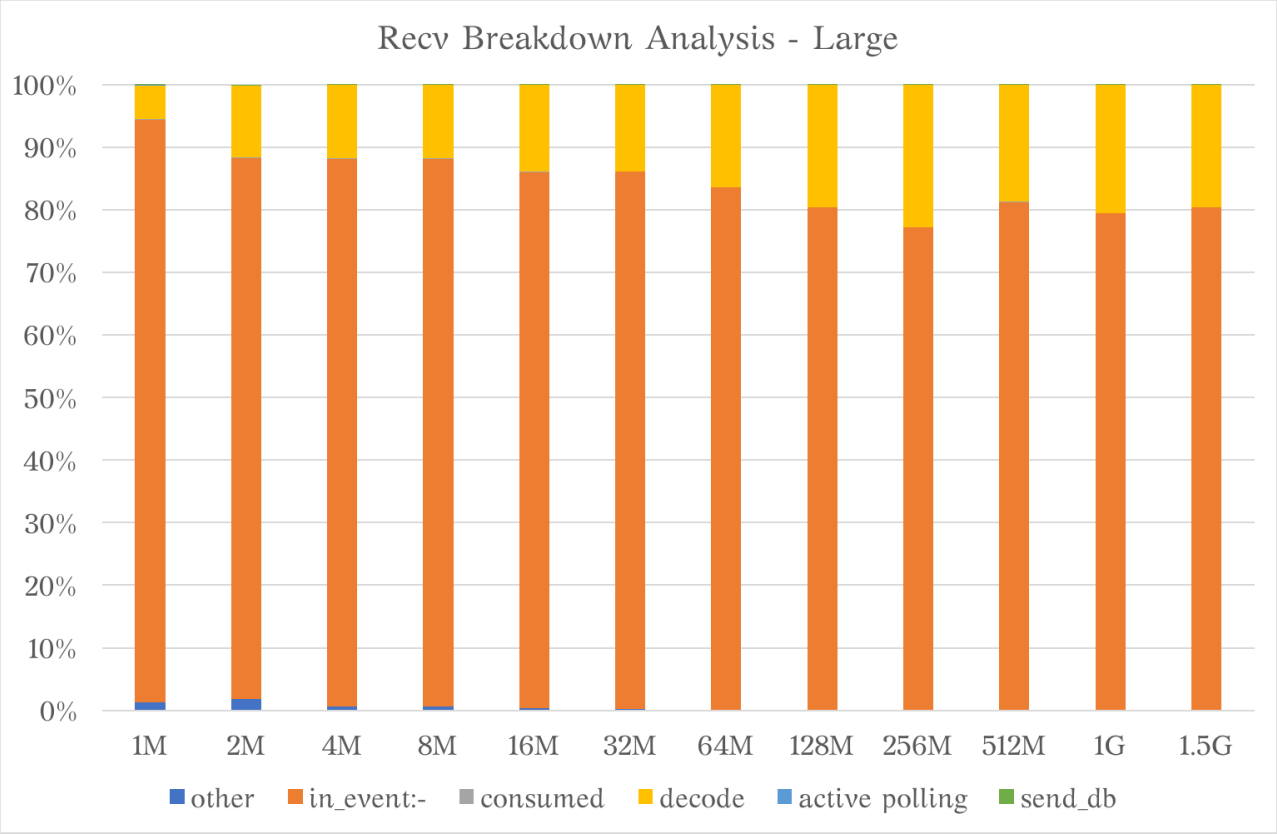

Figure 4.10: Breakdown analysis for the ZeroMQ receiver [Large]

Chapter 5

Prior Art

5.1 Introduction

In this chapter, work to the technologies is presented. The chapter first refers to RapidIO applications and it then talks about the use of ZeroMQ in various contexts, before finally citing previous diploma theses of similar content.

5.2 RapidIO

Work related to RapidIO is generally difficult to be found. Since its traditional application field is embedded systems, we assume that a big portion of research is proprietary. Research has been made regarding its potential use in realtime applications, as presented in [15], in 2004. In 2010 a paper about its suitability for a high-performance, heterogeneous embedded system was published [16].

In 2005 a Linux network driver implementation was introduced [17].

In 2016 RapidIO to PCIe bridge cards were designed, allowing for virtually any application [18]. The work presented in [**?**], introduces the porting of ROOT , a data processing framework targeted at physics data analysis and simulations, [20] and DAQPIPE, a benchmark application to test different network fabrics in preparation for the LHCb experiment [**?**] at CERN,[22], to employ the RapidIO transport, validating its suitability in a modern, HPC context.

5.3 ZeroMQ

ZeroMQ is a popular messaging system, used in a wide number of applications. For example, ZeroMQ was a leading candidate for employment on CERN's middleware solutions to operate accelerators [23].

More closely tied to this work, the ZeroMQ library has been ported to use other high-bandwidth transports, like VMCI [24], which provides communication channels between virtual machines and host. Another implementation supports SCIF [25], which offers a communication backbone between host processors and Xeon Phi coprocessors, and is RDMA-enabled.

5.4 RDMA-enabled interconnects

In the context of previous diploma theses that have been carried out in the Computing Systems Laboratory at NTUA, notable work with RDMA-enabled interconnects has been made. One group refers to the use of 10GbE NICs [26], proposing the SLURPoE RDMA protocol, which was further investigated in [27, 28] for virtualization.

Another group used the myrinet interconnect, a switched fabric interconnect technology that supports RDMA operations, for various applications such as Storage Networks [29, 30] and multi-processor interconnection [31].

It is evident, that RDMA-enabled interconnects offer promising qualities, like low latency, high bandwidth and scalability, and are of interest for many applications.

Chapter 6

Conclusions and Future Work

This chapter contains the conclusions for the work done in the scope of this thesis. A summary of lessons learned is presented, before discussing possible future work.

6.1 Concluding Remarks

As a result of this diploma thesis, an extension of the ZeroMQ messaging library to support the RapidIO transport has been implemented. It allows the use of ZeroMQ as a communication interface in a multi-node, distributed system. Since ZeroMQ provides an interface which is heavily abstracted from its implementation, the use of RapidIO in an HPC environment is facilitated. The coding effort which accompanies the adaptation of a new transport is lifted, as ZeroMQ allows for seamless, trivial choice of transport.

ZeroMQ is heavily programmed around socket interfaces. The implementation process allowed for the investigation of the paradigm incompatibility between the socket programming and memory coherency schemes. A simple mechanism to unify the two communication paradigms has been proposed.

The use of RapidIO has been investigated outside of its usual context, the embedded world. Performance measurements were taken in the context of a 16-node HPC setting, allowing for an evaluation of the protocol.

6.2 Future Work

As part of this thesis a Proof of Concept was created. However, the system's performance can be increased with a possible number of modifications and extensions. Additionally, certain aspects for further evaluation could be investigated. This section proposes potential directions for future work.

On the implementation level:

- The circular buffer implemented in the RapidIO engine of ZeroMQ gives way for optimization. The number of RDMA cells that are used per RDMA transaction could be calculated dynamically, depending on message size and reserved memory constraints.
- The circular buffer could be replaced by another data structure, like a queue. With the proper application of an appropriate algorithm the performance could possibly see an increase.
- The employment of zero-copy operations across the critical path could be investigated, further improving performance.
- The ZeroMQ parts that presented themselves as bottlenecks in the Performance Analysis section of the Evaluation chapter, are candidates for optimization. Not only in the context of the RapidIO transport, but for ZeroMQ in general.

On the evaluation level:

- In order to measure the performance of doorbells, we could get logs from the NIC, for the breakdown sections on the receive side.
- The system could be tested with more than 16 nodes. Further studying its scaling behavior would reveal more performance critical parts.
- A real-life benchmark could be run, like a physics simulation across a distributed system. This would stress the system both in terms of latency and throughput, evaluating the feasibility of its use in such settings.

Bibliography

[1] Mellanox Technologies, *Introduction to InfiniBand™ - White Paper* http://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf

[2] G. F. PFister, *An introduction to the Infiniband architecture*, High Performance Mass Storage and Parallel I/O:Technologies and Applications. New York, Wiley-IEEE Press, 2002, pp 616-632

[3] Intel Corporation, *Intel QuickPath Architecture* https://www.intel.com/pressroom/archive/reference/whitepaper_QuickPath.pdf

[4] *ZeroMQ* https://github.com/zeromq/libzmq

[5] Motorola Semiconductor Product Sensor *RapidIO™: An Embedded System Component Network Architecture* http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.93.4945&rep=rep1&type=pdf

[6] S. Fuller, RapidIO Trade Association, *RapidIO: The Embedded System Interconnect* John Wiley & Sons, Ltd, 2005

[7] RapidIO.org, *RapidIO™ The Interconnect Architecture for High Performance Embedded Systems - White Paper* http://www.rapidio.org/files/techwhitepaper_rev3.pdf

[8] G. Shippen, *System Interconnect Fabrics:Ethernet versus RapidIO® Technology* http://cache.freescale.com/files/32bia/doc/app_note/AN3088.pdf

[9] RapidIO.org, *RapidIO™ Interconnect Specification Version 4.0* http://www.rapidio.org/wp-content/uploads/2016/06/RapidIO-Specification-4.0.pdf

[10] PCI-SIG, *PCI Express® Base Specification Revision 2.1*

[11] RapidIO.org, *RapidIO™ Interconnect Specification Version 3.1* http://www.rapidio.org/wp-content/uploads/2014/10/RapidIO-3.1-Specification.pdf

[12] RapidIO.org *RapidIO Linux Kernel Driver*, 2016 `https://github.com/RapidIO/kernel-rapidio`

[13] RapidIO.org, *RapidIO Remote Memory Access Platform software*, 2016 `https://github.com/RapidIO/RapidIO_RRMAP`

[14] *Internal Architecture of libzmq - White Paper* http://zeromq.org/whitepapers:architecture

[15] D. Bueno, A. Leko, C. Conger,I. Troxel and A. D. George, *Simulative analysis of the RapidIO embedded interconnect architecture for real-time, network-intensive applications* , 2004, Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks

[16] W. Changrui, C. Fan and C. Huizhi, *A high-performance heterogeneous embedded signal processing system based on serial RapidIO interconnection*, 2010, 3rd IEEE International Conference on Computer Science and Information Technology **2** 611-614

[17] M. Porter, *RapidIO for Linux*, 2005 http://landley.net/kdocs/ols/2005/ols2005v2-pages-43-56.pdf

[18] Integrated Device Technology *Tsi721™ Datasheet*, 2016 https://www.idt.com/document/dst/tsi721-datasheet

[19] S. Baymani, K. Alexopoulos and S. Valat, *Exploring RapidIO technology within a DAQ system event building network*, 2017, IEEE Transactions on Nuclear Science **9** 2598 - 2605

[20] Rene Brun and Fons Rademakers, *ROOT - An Object Oriented Data Analysis Framework*, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86. See also http://root.cern.ch/.

[21] LHCb Collaboration, *The LHCb Detector at the LHC* http://inspirehep.net/record/796248/

[22] D. Cámpora, S. Valat, B. Vőneki, S. Baymani, *LHCB-DAQPIPE Wiki* `https://gitlab.cern.ch/svalat/lhcb-daqpipe-v2/wikis/home`

[23] A. Dworak, F. Ehm, W. Sliwinski, M. Sobczak, *MIDDLEWARE TRENDS AND MARKET LEADERS 2011* http://zeromq.wdfiles.com/local–files/intro%3Aread-the-manual/ Middleware%20Trends%20and%20Market%20Leaders%202011.pdf

[24] I. Kulakov, *VMCI extension for ZeroMQ* `https://github.com/Kentzo/libzmq/tree/vmci`

[25] A. Santogidis, *SCIF extension for ZeroMQ* `https://github.com/theWayofthecode/libzmq/tree/scif`

[26] Ν. Νικολέρης, *Σχεδίαση και Υλοποίηση μηχανισμού απευθείας απομακρυσμένης πρόσβασης στη μνήμη με χρήση προγραμματιζόμενου προσαρμογέα δικτύου 10GbE*, Διπλωματική Εργασία, Εθνικό Μετσόβιο Πολυτεχνειό, 2009

[27] Ε. Κοξύρη, *Ένταξη Σημασιολογίας Δικτύων Διασύνδεσης Υψηλής Επίδοσης σε Εικονικές Μηχανές*, Διπλωματική Εργασία, Εθνικό Μετσόβιο Πολυτεχνειό, 2010

[28] Σ. Ψωμαδάκης, *Δίκτυα Διασύνδεσης Υψηλής Επίδοσης σε Εικονικά Περιββάλοντα*, Διπλωματική Εργασία, Εθνικό Μετσόβιο Πολυτεχνείο, 2011

[29] Α. Νάνος, *Σχεδίαση Και Υλοποίηση Μηχανισμού Μεταφοράς Δεδομένων Από Συσκευές Αποθήκευσης Σε Δίκτυα Myrinet, Χωρίς Τη Μεσολάβηση Της Ιεραρχίας Μνήμης*, Διπλωματική Εργασία, Εθνικό Μετσόβιο Πολυτεχνείο 2006

[30] Κ. Βενετσανόπουλος, *Σχεδίαση και Υλοποίηση Οδηγού Συσκευής για τη Χρήση Προσαρμογέα Myrinet ως Αποθηκευτικού Μέσου Υψηλής Επίδοσης στο Λειτουργικό Σύστημα Linux*, Διπλωματική Εργασίας, Εθνικό Μετσόβιο Πολυτεχνείο, 2010

[31] Β. Λιασκοβίτης, *Χρονοδρομολόγηση Φωλιασμένων Βρόχων σε Συστοιχία Πολυεπεξεργαστών Συνδεδεμένων με Myrinet*, Διπλωματική Εργασία, Εθνικό Μετσόβιο Πολυτεχνείο, 2004