Bachelor's Thesis

Low Latency Byzantine Agreement using RDMA

Markus Becker

August 8, 2019

Institute of Operating Systems and Computer Networks
Prof. Dr. Rüdiger Kapitza

Supervisors:
Ines Messadi, M. Sc.
Signe Rüsch, M. Sc.

Statement of Originality

This thesis has been performed independently with the support of my supervisor/s. To the best of the author's knowledge, this thesis contains no material previously published or written by another person except where due reference is made in the text.

Braunschweig, August 8, 2019

_____

Abstract

Increased computational power and need for distributed services put the bottleneck of modern networked applications on the network layer. Among other applications, which would benefit from greater bandwidth and lower latency, Byzantine Fault Tolerance algorithms could gain from a more performant communication protocol because they commonly exhibit high network utilization. Improved performance of consensus protocols could increase their deployment in the future. New technologies offering performance increases require additional work to utilize their advantages. Remote Direct Memory Access is a communication protocol offering extraordinarily low latency and high bandwidth compared to traditional network protocols. To use it efficiently it is necessary to implement memory management and flow control on the application layer. We implement a copy free memory management structure and receive window flow control scheme in the Reptor Byzantine Fault Tolerance framework using the Hybster protocol.

Mr. Markus Becker
Matriculation Number: 4808448
Email-Address: markus.becker@tu-braunschweig.de
Course of Studies: Bachelor Informatik

Technische Universität
Braunschweig
Institut für Betriebssysteme und Rechnerverbund

Verteilte Systeme

Mühlenpfordtstr. 23
38106 Braunschweig
Deutschland

Prof. Dr.
Rüdiger Kapitza

Tel. +49 (0) 531 391-3294
Fax +49 (0) 531 391-5936
kapitza@ibr.cs.tu-bs.de
www.ibr.cs.tu-bs.de

Date: May 1st, 2019

Task Description of the Bachelor's Thesis

Low Latency Byzantine Agreement using RDMA

assigned to Mr. Markus Becker.

Motivation

Byzantine-fault tolerant (BFT) protocols allow mitigating a wide range of failures, thereby ensuring the availability and resiliency of a system. Yet, such protocols are considered costly in terms of message complexity and resource usage. The cost is to a large part caused by their high consensus latency on TCP/IP. A fitting solution is to use remote direct memory access (RDMA) to decrease this communication overhead.
RDMA is a technology that enables direct data movement between the memory of remote computers in a zero-copy manner, without the support of the operating system. Consequently, it helps to reduce the CPU load and to decrease the network overhead, which promises accelerated BFT systems. However, applying RDMA is challenging, since its performance is highly related to many low-level details associated with its resources and operation details, resulting in a whole redesign of a system.
The previous work *RUBIN*[1] is a solution to take advantage of RDMA counterparts without the need to rewrite the communication stack of Java-based BFT frameworks. However, it does not provide a communication buffer management scheme and RDMA-tailored flow control to ensure the efficient use of resources in a BFT setting.

Task Description

This thesis will focus on implementing an RDMA-tailored flow control and a buffer management scheme in the Reptor BFT protocol[2], relying on the existing RUBIN approach. In particular, the thesis needs to address the following steps:

- Analysis of the buffer management scheme of Reptor and the existing RDMA-based library
- Implement an RDMA flow control allowing replicas to exchange the status of the available resources
- Implement a zero-copy buffer management scheme into Reptor

[1] Rüsch, Signe, et al. "Towards Low-Latency Byzantine Agreement Protocols Using RDMA." Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops.IEEE, 2018.
[2] Behl, Johannes, et al. "Consensus-oriented parallelization: How to earn your first million." Proceedings of the 16th Annual Middleware Conference. ACM, 2015.

- Optional: implement additional optimizations, for example inline data
- Evaluation of the implemented extensions via a set of microbenchmarks regarding throughput and latency
  - Optional: evaluation via a replicated key/value store, or coordination service in terms of latency and throughput

General Remarks

The duration is 3 months.

The remarks regarding student theses at the IBR have to be considered.
(see `www.ibr.cs.tu-bs.de/kb/arbeiten.html`).

Task description and supervision

Prof. Dr. Rüdiger Kapitza: _____

M. Sc. Signe Rüsch: _____

M. Sc. Ines Messadi: _____

Thesis work

Markus Becker: _____

Contents

List of Figures

List of Tables

1 Introduction

With an increased need for low latency, large bandwidth transfers, both in academic and commercial applications, networking protocols focussing on performance become incredibly important. Network-based services are widespread, distributed machines running copies of the same software communicating to share resources or enable faster responses to users are common practice. These distributed systems are vulnerable to malicious attacks. This can cause their networked nodes to exhibit arbitrary behaviour which they need to be resilient against. One type of algorithm, that gives distributed systems resilience against nodes experiencing this kind of fault, are Byzantine Fault Tolerance (BFT) protocols. Those protocols often rely on many messages transferred over the network and are highly performance-critical [5]. Thus, efforts have been made to optimize the different BFT protocols themselves, or the framework they are built on top of, employing better parallelization or similar solutions [3, 14]. However, attempts to increase the bandwidth and lower latency of the network layer are less common. Most BFT frameworks are based on traditional network communication protocols like TCP/IP however, newer technologies are available, offering higher transfer speeds and lower latency.

One fairly recent technology, enabling faster network speeds, is Remote Direct Memory Access (RDMA). It allows networked computers to directly access remote memory to read and write and can be used to increase the performance of BFT protocols further. Some work has already been done applying RDMA for consensus protocols [23], however, few are as general as the RUBIN framework [26] made for the Java-based BFT framework Reptor [3]. Fully utilizing the advantages of RDMA requires the restructuring and optimizing of applications [8]. It is possible, that with further enhancements, the RUBIN implementation for RDMA in Reptor can achieve higher throughput of messages and result in overall faster processing of incoming requests. The most important missing feature is flow control, necessary to fully utilize RDMA without encountering data races destabilizing the system, as RDMA does not provide flow control like the Transmission Control Protocol (TCP) does [13].

The reason a technology like RDMA can be used to increase the performance of distributed systems is that their networked nodes need to frequently share data and send messages to each other [26, 8, 4]. Swapping out communication protocols is no trivial task. In case of TCP, a network protocol is built on the transport layer to achieve many features on top of the Internet Protocol (IP), which RDMA connections can support out of the box. However, due to the Direct Memory Access (DMA) aspect of RDMA, develop-

ers making use of this technology need to be aware of its constraints and limitations. To ensure the stability of the network communication of an application using RDMA it is necessary to implement a form of flow control.

The speciality of RDMA is that it alleviates load from the CPU by removing the need to copying data between buffers from user space to the memory area mapped to the network card [18]. This reduces overall overhead on the network protocol, as long as it can be seamlessly implemented. Not involving the CPU in data copies means it can perform other operations. Using RDMA is only reasonable if the overhead needed to make use of it efficiently is less than the overhead it saves. In fact, initializing a RDMA connection costs more than creating a TCP socket [8]. To begin communication over RDMA, both parties have to set up a comparably complex structure on their end to be able to receive and send data [8]. Before any data can be received or sent, both parties also have to have memory committed for those purposes. The RDMA data transfer itself can be initiated by the CPU in one system and is then completed by the network interfaces without the involvement of any further CPU operations [18]. There are different notification models available to further increase the effective bandwidth and lower the latency. Each additional optimization, that can be enabled by RDMA, needs to be handled by the application, making specific RDMA optimized applications practical, which better utilize the performance available through RDMA. The process of developing these applications, or porting an existing application to make use of RDMA, is, however, no trivial process.

Instead of developing a completely new BFT protocol or framework designed with RDMA in mind, which has been done before in protocols like DARE [23], the aim of this thesis is to extend the communication buffer management scheme and tailor a new flow control mechanism for use with RDMA in Reptor. This is to ensure efficient use of resources in a BFT setting. RDMA is also an ideal optimization for Reptor [3], because Reptor is bottlenecked by the network [4].

The state of the recent implementation of RUBIN [26] in Reptor led to a working Java NIO Selector prototype with several further optimization and safeties opportunities [26, 20]. It is necessary to implement a new form of structured memory management and proper flow control. The goal is to keep the RDMA specific implementation to the lowest layer possible. It also keeps the same design goals as RUBIN. This includes the ability to be placed into any Java application, using the same Selector interface, and enable it to make use of the advantages of RDMA. Additionally to furthering RUBINs development, there are also several optimizations for RDMA available which need to be propagated into Reptor's codebase.

1.1. Thesis outline

This thesis is structured in the following way:

- **Chapter 2** introduces the background necessary for this thesis. This includes the technology used in RDMA, the fundamentals of BFT protocols, as well as the software basis this thesis builds on top of.

- **Chapter 3** shows the analysis of the existing structure and outlines necessary changes to RUBIN and Reptor. We design the new memory management as well as the flow control scheme, describing the choices taken throughout the process.
- **Chapter 4** covers the implementation process in more detail. Shown are the exact memory management structures and their interplay as well as a worked example of the flow control algorithm.
- **Chapter 5** evaluates the new implementation in different settings through different scopes of benchmarks.
- **Chapter 6** mentions related work in this field.
- **Chapter 7** concludes this thesis, arguing the portability of RDMA and its state of implementation in Reptor.

2 Background

In this chapter, the technology and resources, necessary to pursue the aim of this thesis, are mentioned and explained. Most importantly the RDMA protocol. The basis of the implementation is the BFT framework Reptor [3] written in Java. Reptor has recently been patched and is already able to communicate using the RDMA protocol by Rüsch et al. [26].

2.1. Remote Direct Memory Access

RDMA is a message-oriented, zero-copy asynchronous communication protocol similar in function to DMA found in modern computers [18]. DMA enables transferring data without the CPU actively moving data around, freeing the CPU to work on non-I/O tasks. RDMA can be used to extend this paradigm to networked machines. Applying RDMA requires additional hardware [19, 17]. That is why it is not commonly used by end-users, instead, RDMA sees deployment in data centres and research facilities. RDMA both enables writing applications that need low latency and high bandwidth [18]. RDMA also requires additional work to make use of its advantages efficiently in software [8]. Traditionally the CPU is informed about the completion of DMA operations by an interrupt. RDMA offers similar functionality. Instead of consuming a larger amount of system resources to process a TCP/IP network stack, RDMA has been developed to reduce the load on the CPU and introduce as little latency as possible.

2.1.1. I/O buffering

In traditional communication protocols, like TCP [24], the CPU copies the data to be transmitted at least once, from the application memory into a temporary network buffer. The temporary network buffer, then, is consumed by the TCP/IP stack, which constructs a TCP packet using the data in the buffer, and initiates the transmit process in the network card employing the DMA controller. This means, an interrupt is sent to the CPU once the operation is completed, optionally freeing the temporary network buffer. These additional steps to sending a message, invisible to the application in user space, add overhead with each packet sent. RDMA, on the other hand, allows the user space application to directly write to and read from the buffers used by the network interface, without any necessary additional buffer copy steps.

Performance advantage of RDMA

RDMA aims to circumvent these buffer copy steps and provide a form of remote DMA, optionally also providing a notification on completion of the write or read event, analogous to the interrupt of DMA. To optimize performance, applications can be designed using RDMA to perform no buffer copies. This reduces load on the CPU to copy data

from one memory location to another. However, patching this behaviour in an existing application is not a straightforward process [8]. As doing so requires writing the data to be sent directly into the buffer used to send it.

At its most performant, RDMA can theoretically support transfer speeds from 10 Gb/s up to 56 Gb/s per port [18]. To fully utilize RDMA superior transfer speeds the user space application needs to be implemented in a specific way. The applications taking full advantage of RDMA have to be well optimized themselves, otherwise, the bandwidth will not present the bottleneck to warrant RDMA. Though the low latency of RDMA can be a factor in choosing RDMA too. But applications also have to be able to produce and consume large amounts of data in short periods of time.

It is, therefore, clear that the application needs to get data it wants to transmit written into an RDMA buffer as fast as possible. As it is not necessary to involve the operating system to copy data into device buffers any more.

Copy free memory management

The reason for trying to avoid additional buffer allocation and buffer copies, especially in garbage collected languages like Java, is the high performance cost of those operations. Depending on the implementation of the Java Virtual Machine (JVM), which runs the bytecode of any program written in a JVM language, garbage collection is rather costly [2]. To be able to collect unused references and free memory used by buffers, the number of active references to an object have to be counted. Different implementations for garbage collection either "stop the world" and analyse the memory used by a process to make decisions, others are iterative or even mixed [2]. This can lead to reduced throughput, delaying actions or even cause timeouts [2]. Allocating additional buffers, therefore, does not neatly replace the memory freed by the buffer just *unreferenced*. To free or reuse memory the JVM has to detect the last use of a reference first and then deallocate the memory. This means adding to the extra work needed to be done, unused memory can remain occupied by the JVM for some time.

However, the garbage collection algorithms are still seeing development promising higher performance, especially as they become more and more common in modern programming languages.

There are two ways to avoid all buffer copies using RDMA, differing in the order of operations. Either a buffer is allocated, then becomes a buffer RDMA can use and is then written to directly by the application, or a buffer the application has allocated and written to is then registered to be used with RDMA. The second option is rather easy to implement for an already existing system using traditional communication methods. This is because it is the traditional way to handle I/O buffering. Following that approach, all that has changed compared to TCP, from the data flow perspective, is that the temporary network buffer has moved into the user-accessible part of main memory. It suffers from worse latency [8] and higher memory usage, especially in garbage-collected languages [2]. The increased latency comes from the additional registration and de-registration that has to be done for each message buffer.

The reason the first approach, registering buffers for use with RDMA, and then filling and posting them, is advantageous, is because the registration and de-registration only happen once at the start and the end of the runtime of the application respectively. This means as long as the application can write the data it wants to transmit directly into the buffers used for communication, the additional overhead can be made up for, as shown in the study by Frey and Alonso [8].

The way to implement a system, which allows reuse of buffers in that way, without increasing the memory footprint immensely, is through memory management. Memory management allows an application to prescribe and regulate the way in which the usage of its memory is structured.

2.1.2. Requirements

To handle RDMA network traffic a special network setup is required. There are dedicated network interface controllers (NICs), which implement the RDMA interface, commonly using the physical InfiniBand connector. The RDMA network interface controllers (RNICs) does not have to use the physical InfiniBand connector, as there are implementations like RDMA over Converged Ethernet (RoCE) and iWARP which make use of existing (Ethernet) networks [19, 18, 16].

The way applications in user space interact with the RNICs is through an abstract interface called Verbs [18]. An instance of the Verbs interface implementation in Java is called jVerbs [27].

2.1.3. DiSNI Java library

Higher-level support of RDMA in Java sees current development [6]. One open-source library Direct Storage and Networking Interface (DiSNI) is actively updated and provides an interface for RDMA more fitting the Java paradigm and language model. It is written mostly in Java with a C interface between the Java functions and the operating system or the RNIC respectively. There exist two kinds of publicly accessible interfaces in DiSNI to extend with one's own functionality. They are each meant for either the server or client component [6]. DiSNI is released under the Apache License (Version 2.0). Building DiSNI only requires very few dependencies managed by Apache Maven, though it has a slightly uncommon build process, as one needs to build its Java and C library parts separately. According to its GitHub repository it runs on Java version 8 or higher [6]. We can make use of it because the codebase, we intend to use it with, is written in Java 8.

2.1.4. Connections

As a network protocol with additional guarantees, and possibly improved speed over TCP/IP, additional steps and preparations are necessary to initiate communication using RDMA. Any active RDMA connection resembles a one-to-one channel between two peers. Both of them have to allocate resources to support the connection. Initially a Queue Pair (QP) is prepared [18]. The QP does not have to be unique on the peer's side, and there can be multiple active QPs per connection acting as a filter or decoder. The QP is the container

for messages, ingoing and outgoing. It is also important for efficient communication to ensure that the QP does not enter an unrecoverable state. Such a state can, for example, be reached when an incoming message cannot be successfully received because the QP was not prepared to receive or the message cannot be handled for other reasons.

Common terminology

The way in which a QP is prepared for send and receive actions is by posting a so-called Work Request (WR). A WR consists of descriptors of at least one Memory Region (MR), which act similar to DMA, and can be read and written to without involvement of the CPU by the RNIC. The MRs used for these operations need to be specially prepared, in a process called pinning. Pinning is necessary to ensure that the memory representing the MR is accessible without causing a page fault. The MR can be accessed by use of a generated cryptographic key [16]. A key is also used to identify the MR to the remote peers [18]. A WR can consist of multiple MRs, thus a WR is actually associated with *Scatter or Gather Elements* (SGE), which allow for combining, and sectioning of, Memory Regions [18].

Operations

Different kinds of WR operations provide different kinds of notifications, aside from the direction of data transfer. The most fundamental operations are WRITE and READ. Both of which either directly write to or read from a region of memory on the remote side of the RDMA connection. That host is not notified and the entire data transfer is invisible to the application running, on both hosts. For example, after preparing for READ operations, entailing sending a key and memory location identification, the remote can read from that memory location at will, without causing any notifications locally. The WRITE operation works analogously, allowing the participant to write at arbitrary times, into parts of the shared memory locations, without causing notification about the operation on the other side. These operations are called one-sided. There are however more complex two-sided operations. The two-sided SEND operation is useful, because it allows polling a Work Completion Queue (WCQ) for Work Completion (WC) events on the receivers side [18, 25]. The execution of both kinds of operations are very efficient and fast compared to traditional network protocols, however, with the added complexity and overhead, the two-sided operations do perform worse than the one-side operations. Most importantly though, the operations themselves are far faster than the time it takes to establish and set up a connection. In certain applications, sending data to multiple peers only once, other network protocols, like TCP, can offer a lower time-to-first-byte, with 0.1 ms compared to 202 ms using RDMA [8].

2.2. Byzantine Fault Tolerance

Byzantine Fault Tolerance describes a property of a distributed system, as it is found in distributed and networked applications, in which the system is required to reach a consensus even though a certain number of participants are allowed to behave incorrectly.

Behaving incorrectly in BFT protocols is defined as the participant experiencing a Byzantine fault to be able to act in arbitrary ways. This extends the fault model further than just crash tolerance, as a malicious faulty member could pretend to behave correctly or lie about the information it has received from others. BFT is one of the most general fault models because of that, making Byzantine faults one of the hardest type to tolerate. The Byzantine Agreement Problem originated from the Byzantine Generals Problem by Lamport et al. [15]. Broadly, the Generals Problem is about communication participants, who cannot trust each other or their communication channel, who try to agree on a decision by sharing their *opinion*. They, importantly, have to be able to form a majority of *loyal* participants. The general assumption about their communication channel is, in practice, that sent messages do arrive, but can be arbitrarily delayed.

2.2.1. Failure model

There are two different failure models used in consensus protocols. Crash failures are moderately dangerous for a system. A participant experiencing a crash failure will not take part in any further communication, as their computation has halted. The cause of this can be a software bug or hardware failure. There are similar symptoms in distributed systems, when the network is experiencing an outage or messages sent are verifiably tampered with. The Byzantine fault model is more general. As Byzantine failures allow a participant to behave arbitrarily, it is a useful fault model for computer systems, which can be hacked or interfered with. Faulty behaviour includes, for example, sending different messages to different participants, delaying communications or lying about information concerning others.

To eliminate the possibility of bugs or incorrect behaviour of a correctly behaving participant, and provide a basis to define a formal protocol on, it is assumed that they execute deterministic state machines to reach their decision given a certain input. For the sake of the BFT protocol, all participants are then initialized with the identical state of the state machine. This process is called state machine replication, the participants, in a BFT consensus protocol, are called replicas.

BFT protocols are characterised by the number of failures they can tolerate and still reach a consensus within the correctly behaving participants. Assuming the consensus is reached by the majority, the total number of members n necessary, given the number of members who exhibit Byzantine failures f, can be expressed as a function of f. For the general Byzantine Problem the condition $n \geq 3f + 1$ holds [15]. This formula states a minimum number of replicas necessary to take part in the BFT communication to tolerate f failures. This means, in a system of four participants at most one Byzantine failure can be tolerated and the correct consensus will be reached, or the other way around, if a system has to be able to tolerate two faults at least seven replicas have to be deployed.

2.2.2. BFT stages

The formula is reached with the above assumptions, correctly functioning state machines will reach the same decision given an input, and they need to be able to share their decision. If they can form a majority, then a consensus can be reached. If a replica is not faulty, then it will also send the correct response, and behave correctly throughout all the BFT stages.

Ordering

The communication in BFT protocols often occurs in two rounds. First, the agreement stage. After a request from a client is received, the request is broadcast across all replicas and the order in which the replicas are processing the outstanding requests is agreed upon by all replicas. This is the ordering part of BFT protocols. Then the requests are executed in the so-called execution stage and each replica sends the result of its computation back to the client.

Checkpointing

In addition to ordering and execution of requests, two additional functions of BFT protocols are necessary. They are checkpointing and view-changes. Checkpointing is required because the network is often assumed to be unreliable [4], but also has other uses. Checkpointing allows a form of recovery from network-based failures, which are either random or have been otherwise resolved. Checkpointing also provides a perfect opportunity for garbage collection as it marks a clean cut in the protocol at which no references or memory of old data is necessary [4]. This involves the replicas sharing all the necessary data for new replicas to join in, and also creates a state, which allows recovered replicas to continue participating.

View-change

Another point of failure, which endangers the reaching of a consensus, is which replica is asked by the clients. Instead of contacting a random one, the client often refers to a publicly known leader. The leader is one of the replicas which receives the requests from clients and initiates the ordering process. During the execution of the BFT protocol, the replicas can request to change the leader if they suspect the leader to be faulty. Once enough requests to announce a new leader have been collected by a replica it will assume the leader position has been changed. This is called a view-change. The order in which the leader position rotates is usually predetermined.

2.2.3. Application

From the view of the client, one message is sent containing a request to the replicas. After the message has been processed, the client will receive a number of responses, one from each replica. At some point, the client decides that it received enough responses and finds the majority of the responses. It considers that majority to be the correct result. The condition $n \geq 3f + 1$, then, means that more than $\frac{2}{3}$ of the distributed system behave

correctly. The additional correct replica is necessary in case the client misses the last response from a correctly behaving replica. When the client performs the majority voting process on the responses, it will still be able to find a majority of correct responses.

This procedure is more complex than finding a consensus when the type of fault the system needs to tolerate is limited to crash faults, as those are generally characterized by $n \geq 2f + 1$, requiring a lot fewer resources when the expected number of faulty replicas is high.

For fully utilizing RDMA, we do also have to assume relative proximity between the replicas. This to some extent reduces the overall safety of the system as physical access to one might make it more likely to be able to physically access another. As well as the likelihood of faults caused by environmental effects. The distance of the replicas to the client is less important.

2.2.4. Deployment environment

It is also assumed that the replicas differ in all ways that enable contamination from one faulty replica to the other functioning replicas, thus byzantine faults occur one-by-one.

In practice, Byzantine fault-tolerant distributed systems are contacted by outside clients communicating with the leader replica to send his request, the replicas of the distributed application then communicate following the BFT protocol, and responses are usually sent from each replica directly to the client. Leaving the client to handle the voting on its own.

2.3. Reptor

Reptor is a framework written in Java, which has recently been enabled to communicate using RDMA to solve issues in BFT systems [26]. This thesis aims to extend the functionality of Reptor's RDMA network layer implementation. To enable the use of RDMA a software project called RUBIN has been patched into Reptor, closely following the implementation of the Java NIO Selector.

2.3.1. Choice of framework

The reason for extending Reptor with RDMA, is that Reptor's maximum performance depends heavily on the amount of bandwidth available to it [26, 20, 3]. It also benefits from low latency and a low memory footprint [26] and the gain in available processor time, if a communication protocol is applied, which alleviates the load on the CPU for sending a message.

Apart from the advantages Reptor would gain from RDMA, Reptor already is a good target to implement further optimizations on. The usual optimization target of BFT algorithms is the protocol itself. This means either reducing the necessary amount of communication rounds or simplifying the protocol in other ways. With the implementation of PBFT [5] and Hybster [4] in Reptor performing well [3], making use of the parallelization scheme COP [4]. Therefore, further optimizations can be reasonably targeted at the network layer.

2.3.2. Layout of Reptor

Reptor is structured in a way to run efficiently on multi-core processors. The implementation of the framework is rather complex, allowing for the stacking of layers as part of its configuration to change its behaviour and type of connection to peers. This stack of layers is shown in figure 2.1. To the outside, Reptor provides an interface to build any application to make use of its layered structure. The way the layers of Reptor interact is similar to the way data is wrapped or unwrapped and interpreted when moving down or up the conceptual Open Systems Interconnection (OSI) layers.

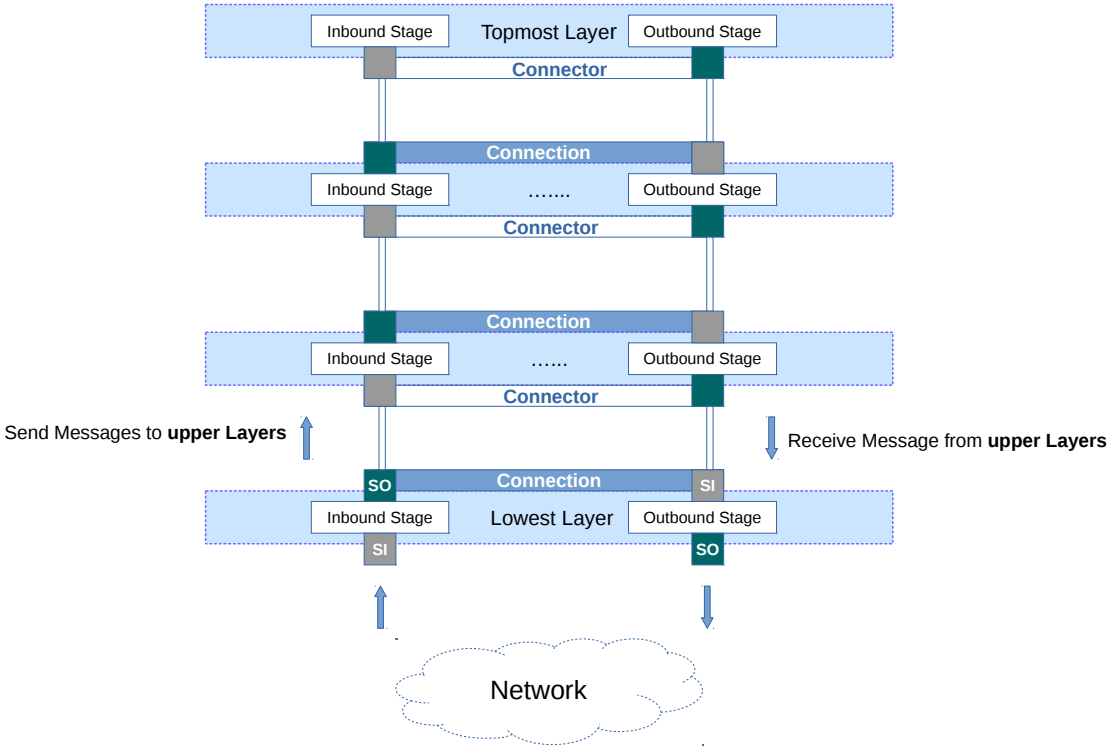


Figure 2.1.: Structure of Reptor consisting of layers, incoming and outgoing sockets as well as links connecting them [26, 3]

In an abstracted view on the semantic layers important to this thesis the stack of layers making up Reptor can be summarised beginning from the top. Most abstract is the implementation of a BFT protocol, built on top of methods provided by the Reptor framework. Those methods represent connective layers of Reptor, both wrapping and unwrapping, as well as, converting data from one format to another and storing it for transfer in different buffers and encodings, until the data reaches the network layer. The specifics of the network layer are oblivious to any protocol implementation and, for the most part, are considered a *blind* sink or source of data.

This is also mirrored in Reptor's structuring and use of data classes. Many of its serialised data representations are already designated to flow up through the layers of the stack as a network source buffer, or be transferred down the layers as a network sink buffer. For multiple reasons, inside of Reptor, caused by its design, it is helpful to think of each side of the stack (figure [2.1]) to be source and sink of data respectively.

Important for the focus of this thesis are the existing memory structures in Reptor. First of all the layers in the Reptor stack are connected to their direct neighbours by so-called links. The links handle the data flow and hand down references to buffers when sending data and up the stack for receiving data from the network. There are different kinds of links. Those which buffer and resize the messages can be more general, whereas links which are unbuffered need to be fine-tuned to the layers they connect.

The implementation of the network layer making use of RDMA used in this thesis is called RUBIN and provided by Rüsch et al. [26].

2.4. RUBIN

RUBIN is a software project aiming to make use of RDMA communication in the shape of a Java NIO Selector. As such, it is no extension of the Java NIO Selector, but rather a copy, intended to replace the Selector in the runtime, to more or less seamlessly replace the default communication model.

The goal is to not only enable Reptor to make use of the developed Selector implementation, but instead to be able to deploy the Selector into any Java application to enable RDMA communication. Especially BFT frameworks written in Java are considered a possible use case.

2.4.1. Java NIO Selector

The way the Java NIO Selector, and in turn RUBIN, works is by registering targets with one management object, called the Selector [22, 26]. The targets, also called channels, then can report capabilities. The application using the Selector is expected to query the Selector for the kind of capability it wants to access. This commonly is either for reading or writing. The kind of keys used as capabilities can be arbitrarily extended. The Selector then returns a reference to the first channel which reported to be able to perform the wanted capability.

Selectors have been a part of Java since Java version 1.4 [22]. They essentially multiplex between an arbitrary number of channels, but with each selection, additional information about that channels capabilities are conveyed by their selection key [22]. Selectors offer a very performant implementation and structure to handle multiple connections.

Another useful feature of Selectors in Java is that they support both blocking (i.e. synchronous) and non-blocking (i.e. asynchronous) selection of channels. This is extremely useful in concurrent high-performance use cases. The blocking calls can either block for as long as it takes to acquire a selection key, or provide a timeout after which the control flow can continue. The asynchronous calls always return immediately and either provide a selection key or return a null value.

More advanced control flow behaviour is also possible to implement through wakeup methods, which wake up blocking threads. The Selector is also concurrency safe, for the most part [22].

2.4.2. Implementation

There already exists the necessary glue code to make use of RUBIN within Reptor and multiple different test applications and unit tests have been written too. Reptor was developed with the Java NIO Selector in mind. With RUBIN being able to take the place of the Java NIO Selector, Reptor can use RUBIN instead.

RUBIN intentionally mirrors the implementation and behaviour of the Java NIO Selector so that Reptor, in particular, does not need to be changed to function. Instead of using TCP, default Java sockets, or even any part of the Java Network Interface, the new Selector extends the DiSNI library to use RDMA endpoints. Bypassing of the Java Network Interface offers a great performance benefit, as it does not even need to be initialized.

The use of the DiSNI library in Reptor is a rather idiomatic Java extension of its interfaces. With a differing philosophy behind Java sockets and TCP compared to RDMA the actual usage of those endpoints differs from how they would ideally be used. This is to be expected as Reptor at large was not designed with RDMA in mind.

3 Design

The main goals of this thesis are to design and implement a copy free send and receive pattern for the Reptor framework based on the Java NIO Selector implementation already using RDMA [26] in addition to increasing the stability of the system by revising a better flow control scheme. This is necessary to avoid QPs entering an unrecoverable error state. To achieve high throughput and low latency, buffer copies have to be avoided and even allocations of new buffers have to be reduced [8]. The time-to-first-byte for RDMA is worse, compared to similar communication methods, but long-living connections achieve up to a magnitude higher data transfer rates on average [8, 7, 18], this means reconnecting two machines, that entered a broken QP state, is also not an option for high-performance applications.

Targeting optimizations at the network layer is possible because the applications, running on the frameworks considered in this thesis, are highly optimized and bottlenecked by the network latency and bandwidth [4, 26]. The domain of many other optimizations is the protocol running itself or the framework on which the protocol is implemented. Such an optimization, available for implementation in Reptor, is zero-buffer-copy. Zero-buffer-copy is especially advantageous in combination with the introduction of RDMA in the network layer [26, 8]. In this context zero-buffer-copy means that throughout the lifetime of a message, it is never moved to another region of memory before it is sent.

The current implementation of memory management in Reptor is inefficient, when data flows through Reptor's layers, buffer copies are unavoidable in certain cases, which both slow down the system, as well as, increase the memory footprint of the entire application. More important, for the stability of the system, is that there is no guarantee that received data is not overwritten with another message before it is consumed. This also extends to sending data. Data, meant to be transmitted, might be overwritten, before it is physically transmitted. To minimize the risk of those events a large number of buffers needed to be used. Thus, the data race evident did not break the system immediately, however, frequent crashes are observable at almost all configurations. Buffer copies were done intentionally to further minimize the risk of overwriting data. Therefore, Reptor shows very high memory utilization. This is worsened by the requirement of large buffers being used, to allow receiving of large messages, as well as small messages, thus, a large amount of the memory would stay unused and wasted. As a rule of thumb, at its most performant configuration, the Reptor framework, running Hybster using RUBIN as a network layer, performed well for up to 11 seconds, after which it is very likely to crash. The crashes are caused by either a message being overwritten or a flow control failure in which a replica in the system is not prepared to receive a message being sent by another. As described

in chapter 2.1, this can happen either when the message is too large, when there was no receive request posted or if one of the replicas posted more requests than the hardware supports. This would lead to more replicas crashing than the protocol can tolerate failing and the system would eventually terminate.

This is why an efficient communication pattern between the replicas needs to be applied. A new form of memory management has to be implemented to enable full utilization of RDMA's advantages too.

3.1. Memory management

Because of the pre-existing structure of the Reptor and the BFT protocols built on top of it, it is necessary to design new management structures to aid in the implementation of zero-buffer-copy and an RDMA aware network layer. To minimize the overhead and amount of code needed to be refactored, the added structures need to be as similar to the default memory solution in Java as possible, which would allow reusing of existing code and minimize the risk to introduce additional bugs. However, it will be necessary to be able to access the introduced designs in a structured and controlled way to enforce zero-buffer-copy and ensure data safety properties are kept.

3.1.1. Smallest unit of memory management

The fundamental unit of the memory management introduced to Reptor in this thesis is a Managed Buffer component. Each Managed Buffer is a wrapper for a standard Java buffer. It is important that some calls have to behave differently to allow zero-buffer-copy and interaction with RDMA, without major problems. To allow multiple different implementations and further changes, an abstract API is defined in Reptor that allows for changes in behaviour while still allowing interoperability with the introduced memory structures.

The structure present in Reptor to allocate memory is a functional call that can allocate new memory on demand and returns the references to buffers without any further logic. Replacing this behaviour requires additional management structures. The management structures need to be able to create and return references to memory. For ensuring that buffers are not actively used by two separate parts of the system, uses of Managed Buffer objects also have to be managed. This can be done simply by counting the number of currently used references per buffer. The memory management structures also have to cover the same functionality as the existing implementation, in addition to being usable in the network layer. The new design also has to establish the necessary properties needed for use with RDMA. This could also be done with only limited refactoring by changing the implementation of the functional handle used. A design like that would be harder to reuse and maintain and would probably be harder to implement and more prone to bugs. There are also concerns whether a functional interface for memory access and management would be sufficient to provide the guarantees which can easily be

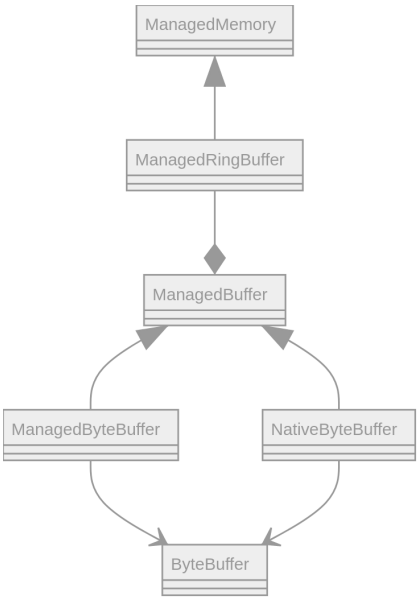


Figure 3.1.: Simplified class diagram showing the classes designed to manage access to memory.

implemented using object-oriented code. It is technically possible to do so, but the size of a completely functional implementation would likely outgrow the structural approach using an object-oriented paradigm.

In conclusion, the smallest unit of the new memory management design is a Managed Buffer. The Managed Buffer exposes an API identical to the standard Java buffer. To deploy them in Reptor, it is going to require additional structures built on top of them to properly replace the current design, and allow for the necessary control over the use of memory.

3.1.2. Management structures for memory

Even before considering the structure, in which the Managed Buffers need to be stored and how they are going to be used, the layout of the management structures themselves needs to be designed. While every message needs at least, ideally exactly, one instance of a buffer, it is important to note that, we want to be able to use different management structures for different peers and, more importantly, different structures for transmit and receive directions. This is because every management structure is going to behave as the source of buffers for that part of Reptor.

Scope of management structures

The pre-existing implementation enabled almost all layers of the Reptor framework to allocate memory at will. This allows for the allocation of arbitrarily large buffers, using a single call to a lambda object, from anywhere in Reptor. To minimize the amount of refactoring and changing of behaviour, multiple levels of abstraction on the concept of

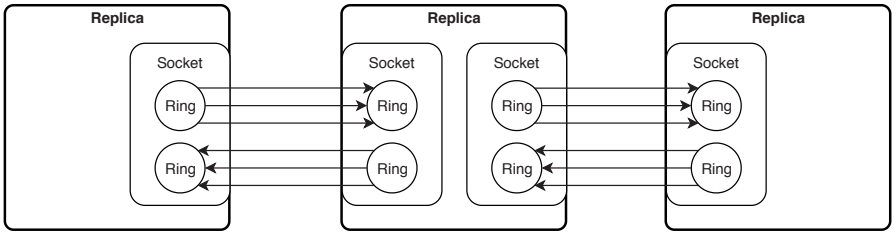


Figure 3.2.: Sockets of replicas, each with send and receive buffer rings for each RDMA connection to any other replica.

memory allocation need to be constructed. The design presented in this thesis calls for four different levels of abstraction along with their differing scope 3.1. These are the Managed Buffers at the bottom and the control structures, called Managed Memory, on top of the buffers. For use with different Sockets, factories will have to create unique Managed Memory instances. To reduce the memory footprint, the highest level is formed by Singletons for overviewing the memory allocation process.

The standard Java buffer is to be replaced by the Managed Buffers. They have to allow writes and reads of the underlying data, as if they need to replace the Java buffers in Reptor. Importantly, the internal buffer of a Managed Buffer must never be replaced by another. The internal buffer must not be able to grow and they must not make the internal buffer accessible by any other way, to provide necessary safeties.

Building on top of the foundation of Managed Buffers, structures can be designed to cycle through already allocated Managed Buffers and allow Reptor's layers to access memory without major changes. These rings of Managed Buffers are the solution to the controlled access to memory. Iterating through a ring of buffers is able to replace the existing memory management, provided there are enough buffers available and the buffer ring can count uses of each buffer object. This will require changing the parts of Reptor that allocate memory and those which consume the data in them, to notify the ring of buffers. To improve performance and enable zero-buffer-copy, the buffers of the buffer ring also have to be able to be registered with RDMA, which means they, and their pinned memory region, have to be able to be linked to each other. This also limits the amount of useful buffers in any system to be used with RDMA as the hardware, specifically the RNIC, has an upper limit of posted requests. This can be circumvented by batching in the network layer. Apart from that, the only limitation to the size of the ring is the memory addressable by the Java Virtual Machine.

It is preferable to use the buffer ring sequentially, in essence, to minimize occurrences in which the next buffer in the ring would still be in use while there exist available buffers further along in the ring. Though this cannot completely be avoided, it can be minimized by allocating a buffer ring for each consumer and producer. The figure 3.2 shows this applied to the network layer. In the network layer, both sending and receiving represent a consumer and producer, respectively. The aim of this design, and necessary to achieve

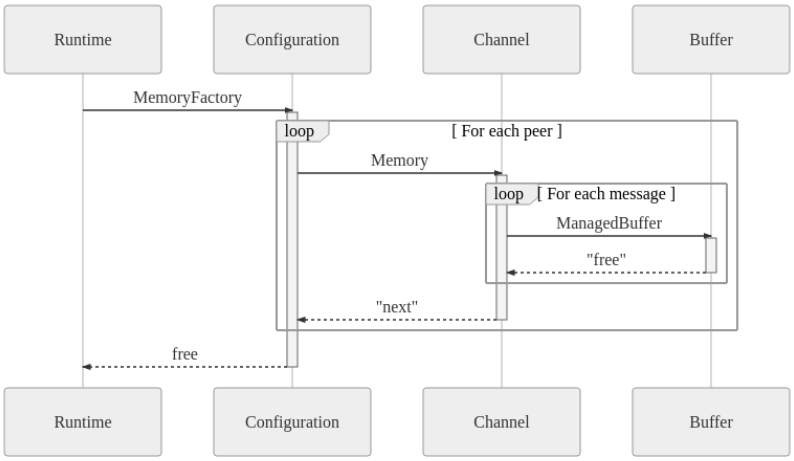


Figure 3.3.: Lifetime of buffers.

zero-buffer-copy, is to propagate the use of these buffer rings further up the framework, Reptor. This lets different buffer rings semantically become sending and receiving buffer rings, with identical behaviour.

For the use with RDMA, each replica should have its own separate memory source for receive buffers. Otherwise, multiple channels could post receive requests for the same buffer without knowing and, thus, overwrite data. There are solutions to circumvent this problem and use the same memory factory for all peers and still ensure that no two sockets have access to the same buffer to receive data into. In this thesis, we were able to use the separated sources of memory for each socket's receive buffers to do further optimizations.

Memory reuse and safety

The lifetime of a single buffer extends for longer than the lifetime of the message in the system, as the buffers are kept in a buffer ring. To make sure data in buffers is never overwritten before it was consumed or is otherwise not useful any more, the references to the buffers have to be counted. Simply, on obtaining a reference to a buffer a counter is increased, and after the buffer was used a free call has to be invoked, not asimilar to languages without garbage collection. This buffer lifetime is shown in figure 3.3. Once the buffer is freed, by a user, the counter tallying its usages is decreased. If the counter is decremented to zero the buffer's state is also reset to default values so it can be reused as if it was just allocated and registered with the RDMA channel. There is no enforcement, in the code, that prohibits recent owners of memory to continue using it, after they marked it as freed. This design was chosen to increase performance, as accessing the memory can be done without authenticating or identifying the user, which would have been required otherwise. The contents of the freed buffers are not reset, which may have security implications if parts of the framework are not trusted, for this thesis, however, we assume the layers of Reptor having access to the buffers are trusted.

This implementation should keep the memory footprint of the network layer of Reptor constant, as it can reuse the same buffers over and over again. The protocol on top is still able to allocate arbitrary amounts of memory. Assuming the protocol does not contain a memory leak and its memory utilization does not grow arbitrarily large, given that the system running Reptor with a BFT framework has enough memory available, this memory management scheme will be sustainable. It could be argued that, exposing Reptor's new memory management to applications running on top of Reptor might be advantageous, for further memory usage optimization.

3.2. Flow Control

The design of the memory management, described in chapter 3.1, does have some functionality to aid in ensuring that buffers are not incorrectly reused or overwritten. However, in the network layer of Reptor, changes need to be made to ensure that buffers are freed in time to be reused. One way to ensure this, is by implementing flow control. Reptor was not developed with a network layer in mind, which requires an inherently different kind of flow control, for both send and receive, compared to TCP/IP. RDMA and TCP both offer reliable connections, but RDMA operates on completely different verbs than TCP does [18] and requires more setup and initialization [8]. The work required, to enable Reptor to send and receive data using RDMA, has been done before [26]. RUBIN, however, was incomplete in terms of flow control and lacked stability, among other features that need to be designed and implemented to realise its use in a protocol.

3.2.1. Necessity of Flow Control

The fundamental difference, and the reason why RDMA requires additional flow control, is that the CPU is not involved in the actual data transfer taking place. The RNIC has to be configured to be able to write arriving data into RAM before such data arrives. The state of the two peers, taking part in an RDMA connection, is stored in a QP. After the QP is initialized between the two peers both of them can post send and receive requests. RDMA also supports more performant direct reads and writes to memory. This thesis will only make use of the RDMA operations with notifications, which allows the software using RDMA to be asynchronously notified of completed WRs, both send and receive. To avoid an unrecoverable QP state, the receiving side of a transaction always has to have a receive WR posted, with sufficient pinned memory, to successfully receive the data the sender is providing. The registration of such memory blocks, as well as, the posting of the WRs, is required to be done by the CPU ahead of time.

Notably, for the sake of optimizations on the receiving end, is that the same WR, describing the same MR, can be posted multiple times. The limiting factors are that all RNICs have an upper limit on the amount of queued up WR they can queue on each connection. They are also limited by the speed at which the software using RDMA can consume and act upon the arriving data and fulfilled WRs. The maximum amount of data transferred in a single request is 2 GB per message, but may be limited by other factors [25].

3.2.2. Existing Flow Control

The existing structure in Reptor is employing a Java NIO Selector. This is either the default Java NIO Selector for the Reptor implementation using TCP, or RUBIN for the Reptor implementation using RDMA. RUBIN's Selector acts on the WC events asynchronously emitted by the RDMA connection [26]. The completed requests have a `wr_id` used to determine the buffer, which just received the data. The data is then copied into a separate buffer provided by the Reptor layer above. The buffer copy step was necessary because RUBIN needed to avoid overwriting received data at all cost. To replenish the posted WR

queue, all receive WRs are posted again, when possible. This is done to minimize the risk of not having a receive request posted when the peer, sharing the RDMA connection, wants to send. The exact timings are protocol and implementation-dependent, but, in this application, the problem arises because one of the participants in the communication sends messages faster than the receiving end can process and *repost* the requests. Thus, applying RDMA in an already existing application can require flow control, not unlike TCP's sliding window, in which both parties are aware of the receiving capabilities and capacities of their peer.

It is clear that, in an application developed with RDMA in mind, with very predictable data flow and communication, and few and large data chunks, this overhead can be minimized or completely resolved, which is when the application of RDMA is most advantageous.

3.2.3. Requirements of new Flow Control

The changes intended to be implemented in this thesis are those necessary to enable stable communication between peers to avoid entering an unrecoverable QP state. This needs to be done while adding as little overhead as possible. To keep the code portable, changes will be limited to the lowest layer, when possible, and change as few methods of the network layer, as possible. As before, the implementation also needs to be interchangeable with the default Java NIO Selector. Different benchmarks and applications built on top of Reptor have different configurations concerning the stack of layers, which need to stay compatible with the new flow control. It is, therefore, important that, the changes made to any message, or any additional communication, are completely invisible to the layers above, as they should be able to consider the network layer a sort of black box. Only the links between the layers should be aware of implementation-specific details.

The baseline exhibits a race condition between the sending peer and the receiving peer. Notably, those roles change over the lifetime of the RDMA channel, as the protocol using the Reptor framework can enqueue reads and writes between any replicas in arbitrary order and relative magnitude. It is to be expected that protocols will present asymmetric behaviour over short periods, in which one peer predominantly enqueues sends and the other has to be able to receive large amounts of data or numerous messages in succession. If, at any time, the receiving party is slower at preparing receive requests, then the sending party can exhaust the QP, and an unrecoverable error state can be reached. This is the reason why a flow control scheme inspired by the TCP receive window needs to be realized and implemented. It is not necessary to implement the entire flow control functionality of TCP, for multiple reasons. Our goal is to solve the problem described above, commonly known as *back pressure*, we do have guarantees which TCP implementations do not have, as well as fewer requirements than the entire TCP network stack has to meet. Most importantly, unlike IP, the reliable connected QP guarantees that each message is transmitted at most once, without corruption and in order, in a one-to-one cardinality with another reliable connected QP. There are other Queue Pair modes with fewer guarantees and, thus, a lower overhead, they are no reliable connections, which we assume to

have. All of those guarantees given by the implementation of RDMA are features the TCP stack has to implement in software, which we do not want to implement in software here and instead use the faster hardware implementation [18, 19], that the reliable connection mode of RDMA provides.

Ideally, any added flow control scheme does not decrease the overall performance of the system. Also, as this flow control scheme is applied to RDMA, some additional constraints have to be taken into consideration. First of all, the hardware responsible for executing the RDMA operations has limitations which cause a faulty QP state when exceeded. If the new flow control scheme intends for any batching to occur, then the maximum number of outstanding and completed WRs queued in the RNIC has to be taken into account. This means that, if flow control schemes call for higher batch numbers, or allow arbitrarily large batches, then they cannot be used for this purpose. Similarly, additional logic will be required, likely to contain some state, which has to be stored in memory of the system running the application. Assuming this state needs to be uniquely stored for each connection to a replica, then the number of peers a replica needs to connect to, using the flow control scheme, is limited by the amount of memory available.

3.2.4. Exploration of solutions

There exist very basic mechanisms which can often be used to implement a working flow control mechanism resolving the problem of back pressure as described in the context of TCP. Though simpler to apply than the solution settled on, they come with drawbacks of their own which makes them inapplicable for this specific problem.

Buffering

The naïve solutions to back pressure include the buffering of messages to process and consume them at a later point, when the constant influx of messages from the sending side reduces or stops. This only works if enough messages can be buffered to survive the period of time, the sender is bombarding the receiver with messages, giving this method an upper bound based on the memory available to the system, as well as the processing power of the system. If the receiver is not able to process all outstanding buffered messages before the next period of severe load ensues, the amount of buffered work will pile up and cause unbounded memory use in the long run. Another reason, buffering cannot be considered the solution for this problem, is that the other goal outlined in this thesis is achieving zero-buffer-copy for sending and receiving messages. Buffering could mean keeping a possibly arbitrary amount of messages in memory. This either will require a very large memory footprint even when most of the memory is not actively used and just reserved for peak network traffic times, or a complex adaptive algorithm has to be developed and implemented.

Dropping

Another often-used solution, to the problem outlined, is the dropping of messages if the receiving side is not able to process the amount of incoming traffic. There are again multiple reasons this is not a viable solution in our case. Most importantly, the dropping of messages through RDMA is only possible in the sense that the framework on top is not processing the messages, because the network interface controller is programmed to process the incoming message without the involvement of the CPU. This would mean that the sender would not even be aware that the receiving software decided to drop the message. This would, in the long run, increase the network traffic and does not even solve the problem of the race condition, described above.

A version of dropping could be implemented by using direct writes and reads, which RDMA supports. The notifications, provided by RDMA send operations, however, are crucial to the design of the Selector at the core of RUBIN.

Stop-and-wait

Stop-and-wait describes the behaviour of the participants communicating using the stop-and-wait flow control scheme. Each participant is aware that it is allowed to send a single message, once it does so it has to wait for the receiver to acknowledge that the message arrived and that it is now able to receive another message. Stop-and-wait is commonly implemented in reliable connections, where one can assume that the message will at some point reach the receiver, and the receiver will be able to send the acknowledgement.

Stop-and-wait, however, does not provide the necessary performance, as Reptor's performance is highly dependent on low latency network traffic [26]. Stop-and-wait roughly doubles the latency, because it necessitates the acknowledgement to be sent in response. Stop-and-wait excels in its simplicity. The approach of notifying the peer about ones own ability to receive messages can be used in our case. To reduce the overhead on each message one needs to prepare and notify the peer about the ability to receive multiple messages in a single message. This way the overhead of the additional message can be divided across the number of messages which can be sent based on that information. This proposed scheme is most performant, the higher the number of messages a peer can prepare for and the faster that additional message can be sent.

That scheme could be further optimized by not necessarily requiring an additional message containing only flow control information, as the additional data required can be appended to the normal messages sent at a lower cost than sending an additional message. This can be taken even further by looking at the information available to both peers taking part in the communication, leading to the receive window flow control scheme.

Receive window

The solution, reduced to the fundamental construct, is visualised in figure 3.4. Similar to the TCP receive window, both peers will keep track of two additional numbers and interchange information about their state with every message. The window represents the number of available buffers, ready to be used to receive buffers, this number is computable

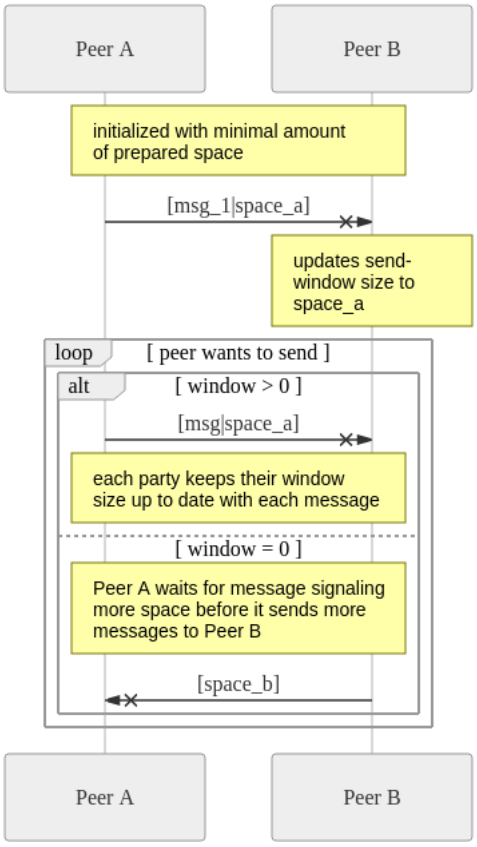


Figure 3.4.: Example sequence diagram of the worst case: one peer predominantly sending, applying flow control.

locally at any point. The state of the peers is described by two integer numbers. First, their knowledge about the window size of the remote peer. To keep in sync with the remote's actual window size, this number needs to be decreased, with every message sent to the peer, and updated once the peer prepares more buffers. Secondly, they need to keep track of the knowledge the remote peer has. Because RDMA offers a reliable connected channel, the local peer can count the number of messages the remote has sent. This allows calculating the remote peer's knowledge about the local peer's capabilities. This second value will slowly drift out of sync, due to the posting of new receive WRs, which grow the window size. The first part is rather trivial, if at any point the local peer knows about the size of the window, updating the window size with each message is simple. The second part, the assumption that the remote peer is also keeping track of the window size, is necessary, to know when to update the remote's state, if their knowledge goes too far out of date.

This means additional rounds of communication, just for flow control, are not necessary. Only if the remote peer's knowledge is outdated, a single message, just containing flow control data, needs to be sent. Those flow control messages do not need to be acknowledged either. Furthermore, the additional information about the window size can also be appended to the handshake step of Reptor because any added information can be seamlessly removed. This means, replicas only have to be able to receive a single message and can then adapt to their remote. Importantly, only a few bytes have to be appended to each message. Assuming that those values can be kept reasonably small, when encoded for transmission, and the messages themselves are far larger than the additional bytes sent, then the overhead, on the network, per message sent, should be negligible.

In the bigger picture, the memory management structures from chapter 3.1, especially the Managed Buffers, as well as the buffer rings, can be used to implement this behaviour. Mapping each Managed Buffer to a message, the buffer ring can keep track of the buffers currently prepared to be used for receiving, using the reference counter. Upon receiving data, the Managed Buffer can be used in Reptor, like any other buffer, with the difference that it needs an additional call, when the data has been processed, to mark the buffer as freed. This way, the receive window is represented by a chunk of Managed Buffers, from the buffer ring used for receiving, indicated by non-zero reference counters.

A more realistic bidirectional conversation using this flow control paradigm, and a more complete picture, is shown in the sequence diagram 3.5. A complete analysis, and worked example, is shown in 4.3.

Sliding window

TCP, which the receive window flow control scheme is inspired by, does enable further functionalities grouped under the term sliding window, which contains the receive window scheme, among other things. This is necessary, because TCP builds a reliable and ordered transport layer on top of the not reliable and unordered internet protocol [24]. Some assumptions about our system influence the choice on what can be omitted from the TCP flow control implementation. The replicas will all prepare the same size of buffer

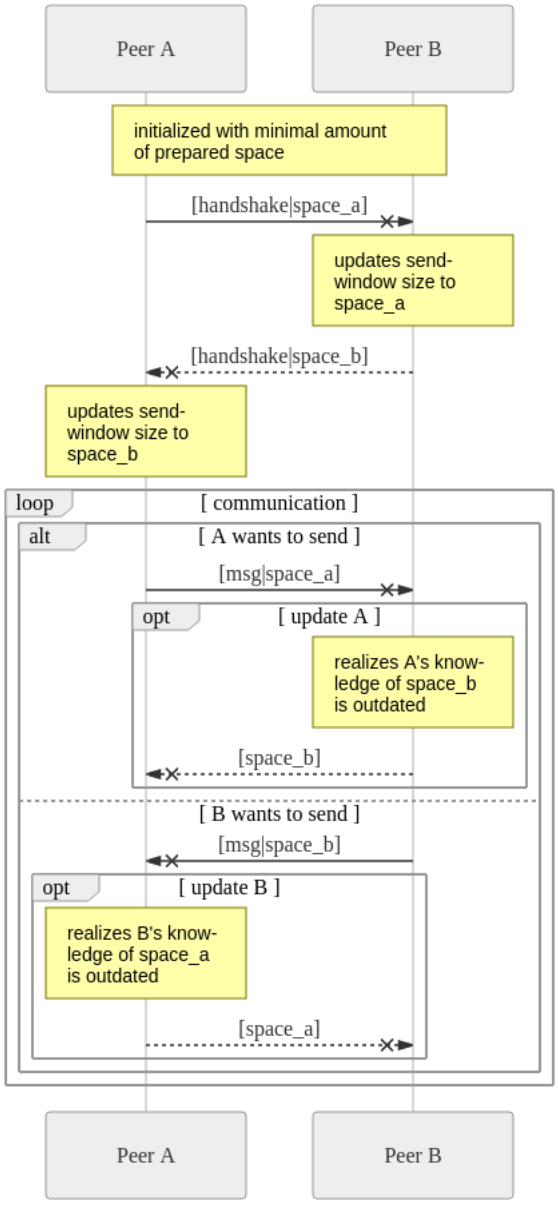


Figure 3.5.: Complete messaging scheme.

ring, and all buffers are the same capacity. Further, replicas, experiencing Byzantine Failure, attacking the flow control implementation of another replica will only cause that replica to be unable to communicate with the faulty replica. This means, a failure in the flow control will only cause the affected channel to crash or stop working, rather than all channels the replica has.

Most of the left out features can be omitted because RDMA, in a reliable communication mode, already provides what TCP has to achieve through flow control, on top of IP. Also, if one of the QPs, taking part in a communication, enters an unrecoverable failure state, the other will do so too. This removes the possibility of half-open channels. Because TCP packets can be dropped or may not reach their target, they need to be able to be resent, as TCP aims to provide a reliable transmission. This adds additional communication because the state of transmitted packets needs to be queried. Implementing receive window for RDMA can be further simplified because transmissions are error-free. This removes the need for negative acknowledgements in the transport layer too.

3.2.5. Scope of implementation

Implementing the presented flow control scheme in Reptor requires some changes to be made, not only to the network layer, but will also necessitate some changes to be propagated into Reptor's higher layers, as well as the Selector implementation of RUBIN. First of all, because of the buffering nature of TCP, and the way the old implementation of RUBIN was integrated with Reptor, there is currently no intended way to signal Reptor that the sending of a message failed. The closest to this would be using the Selector to disable sending for the channel that is currently unable to send, due to the flow control state. It is currently assumed that the socket will always be able to send, which means those changes need to be both applied to the Selector and Reptor's higher layers.

3.3. Security Analysis

Memory regions are read from and written to directly. A compromised machine can still read and write to all registered parts of memory, which it was granted access to, at will. To develop secure software one has to assess this as an attack vector and inherently not trust any memory which was registered for RDMA. An additional risk, considering the layout of RDMA communication, is that the authentication, used by the remote to initiate one-sided operations, could be guessed [28]. The permission management is done by splitting the authentication across two keys, a local key, used for identifying local memory regions, and a remote key, which authenticates the remote to write to and read from the memory region. Using the two-sided RDMA SEND the rkey does not have to be used, because the sender does not know where the data will be written to [25]. This means that even if the rkey is not shared, a remote peer, connected by RDMA, could feasibly guess rkeys [28], and directly read from and write to memory it was not granted permissions for. The problem of the rkey's security is that they are predictable [28], and side-channel attacks on modern RNICs are possible too. This is not by design and could be resolved in future iterations

of the hardware, as the attack is purely based on the implementation of the RNICs. It is also feasible that future RNICs could allow disabling one-sided operations entirely, to rule these kinds of attack out.

On the other hand, the permissions structure of the RDMA enables forbidding the entire communication with a remote peer. Additionally, all network traffic over InfiniBand is securely encrypted [16] and, according to Mellanox Technologies [16], provides better overall security than standard high-performance networks. Each message is paired with the hardware address, queue number and sequence numbers combined to form two CRCs, making up a packet. This security partially depends on the low latency of messages and CRC used, which makes it increasingly difficult to fake a packet before a real one is sent. Under these assumptions, RDMA can be used to share memory which can be trusted [23].

This does not contradict that memory received or read through RDMA cannot be trusted. Because if a host is exhibiting Byzantine faults, it remains capable of sending arbitrary data and read all memory regions it has access to. Though the right to do so can be revoked by the remote party. Assuming that trusted shared memory can be built and RDMA can be used for BFT protocols, this still leaves an attack vector when using RDMA if the application using it trusts data received or read with it inherently. In our case, if we can assume that only RDMA SEND operations are used, the risk of misusing the rkey diminishes.

4 Implementation

The implementation was done in Sun's Java 8, based on the Reptor framework including the changes undertaken to enable the network layer to use an RDMA Java NIO Selector [26] called RUBIN. Different approaches were considered and the majority of them taken to investigate their feasibility and potential. The design chosen in chapter 3 is implemented in logical steps with unit and integration tests at each stage.

4.1. DiSNI Upgrade

The most popular library to make use of RDMA communication in a Java application is DiSNI. This library was used before to implement RDMA in Reptor, because Reptor is written in Java. DiSNI provides an open-source implementation of the abstract Verbs API written in a combination of C and Java [6]. The reason an implementation of the RDMA Verbs API is preferable, is because they are closest to the actual hardware functionality the RNICs provide to be used. Circumventing the default Java Network Interface also enables running at ultra-low latency because the Java Network Interface of the Java Virtual Machine does not have to be initialized, which removes some additional overhead [27].

4.1.1. Existing implementation

The implementation of the network layer using RDMA in Reptor made use of the Java library DiSNI version 1.4 [26, 6]. An effort was made to upgrade all of the projects applications to use the, as of the time at the beginning of this thesis, newly released version 2.0 of DiSNI. Since then another version bump has taken place, to version 2.1 on the 20th of June 2019. The porting of the project to version 2.0 of DiSNI was only partially successful.

4.1.2. Porting Reptor to DiSNI version 2.0

Separated unit tests and applications were able to be ported over, even the implementation of the Java NIO Selector built on top of DiSNI [26] was ported successfully.

There were two different strategies to change the Selector implementation, making use of DiSNI, following different philosophies. First, to allow easier upgrading of DiSNI versions in the future, and be closer to idiomatic Java, it was attempted to write a sort of wrapper around the DiSNI library. The idea was to only minimally change DiSNI and build a kind of framework around it, which the Selector implementation of RUBIN could make use of. This fundamentally succeeded but would have required large amounts of the Selector and Reptor to be changed to integrate it properly and was put aside in the interest of time. The more serious strategy was then to directly change DiSNI's source code and keep the Selector implementation identical. This meant both the *old* changes to DiSNI needed to be transferred over in addition to reverse engineering parts of DiSNI

to add backwards compatibility between version 2.0 and 1.4 after the fact. Having DiSNI undertake a major version change meant this was no small task, as even the way the code was structured changed from version 1.4 and 2.0. Finally, a version which passed the Java compiler and successfully ran unit tests was achieved through this *direct patch* strategy.

However, the entirety of Reptor was not able to function with the replaced Selector. This could be due to a different implementation of DiSNI on the native side (written in C) or, because the effort to port Reptor to DiSNI 2.0 was undertaken before the implementation of flow control, some data race, previously harmless but now causing Reptor's SocketChannels to misbehave, could be the culprit. It is reasonable to plan to undertake the version upgrade of DiSNI in Reptor in the future, as stability and support for more functionality should increase by the upgrade. The limited testing and benchmarking done with DiSNI 2.0 using the slightly changed Selector implementation did not seem to benefit noticeably, however, no memory profiling or long term tests were undertaken.

4.2. Zero Buffer Copy

The buffering and general memory allocation solution, predating the work presented in this thesis, can be separated into send and receive components. On the sending side, the implementation made use of a functional object `buffac`, which on call with the argument `bufsize` allocated a DirectByteBuffer using `ByteBuffer.allocateDirect(bufsize)`. This simple solution provided each outgoing message with a buffer to serialize into. The functional object of the direct allocation was also provided to each network buffer, and other network classes, to allocate additional memory alongside an already allocated buffer they can use. This function to allocate was used to replace the inherited buffer, with another buffer of greater capacity, in case the data outgrew the initial buffer. Finally, before being sent using the SocketChannel, the contents of the message buffer were copied into the only send buffer prepared in the SocketChannel and its corresponding send WR was posted and executed.

The receiving data flow is based on the array of ByteBuffers in the SocketChannel, which are registered for receive WRs, and a sublist of the WRs are regularly posted. The Selector notifies the framework and protocol at large about the event of an incoming message, the Work Completion Queue is then polled for Work Completions. The first completed WC can then be queried for its id which relates to the buffer, which holds the data of the completed data transfer. The contents of this buffer are then copied into a ByteBuffer, provided by the caller, and a reference to the buffer is returned. Depending on the link and configuration of the stack either the newly created ByteBuffer, used to call the read routine, or the returned buffer are then used to serialize the data into an object used by the protocol at large. The encoding, the data is serialized into, is Reptor's own PushMessageEncoding. There is no guarantee that the data held by the returned ByteBuffer was not changed by subsequent RDMA receives before it is serialised. There was also the possibility that some configurations attempted to resize or flip the ByteBuffer returned by the SocketChannel, which would interfere with the RDMA request posted. The problem arises because the

buffers have to be registered to be used for sending, and incoming data is written into the registered buffer location. There is no guarantee that during a resize or flip the actual memory location of a buffer will not be changed by the JVM. This means that, the physical memory location of the buffer may not be the initially registered one, after a resize or flip.

Another notable feature is batching of requests posted by the SocketChannel. The pre-existing implementation posted the receive requests with indices between 0 and `BATCH_NUMBER` − 1 including. However, if the total number of buffers was equal to that number all receive requests were posted. This re-posting was undertaken when receiving a message which was the fifth to last message posted. How far ahead this re-posting is of exhausting the total number of outstanding receive requests is more impactful for the performance evaluation.

4.2.1. Implemented memory management structures

The new memory management structure following the analysis and design in chapter 3 is itself structured in multiple layers managing access to a buffer container. The entire structure implemented is shown in table 4.1.

The unit of the memory management is the ManagedBuffer class. To be thought of as an extension of the ByteBuffer in almost all ways. An instance of a ManagedBuffer is either a ManagedByteBuffer or a NativeByteBuffer, both of which expose the same API as a ByteBuffer would. ManagedBuffers are not able to be extensions of ByteBuffers, which obstructed the possible simplicity of implementation. Part of implementing the new memory management, therefore, entailed changing the signatures of Reptor's functions, to not take ByteBuffers but instead ManagedBuffers as arguments. That is why the two separate classes, ManagedByteBuffer and NativeByteBuffer, implementing the ManagedBuffer interface exist. After changing the signatures, the NativeByteBuffer can be passed as a ManagedBuffer and will behave identical to a ByteBuffer, while the ManagedByteBuffer is implemented to support RDMA and check for correct usage. This means trying to flip or grow it will cause an exception to be thrown which allows for providing safety of some properties at compile time.

The ManagedBufferRing acting as the buffer ring described in chapter 3 can be implemented succinctly following the design outline. Internally storing the ManagedBuffers in an array with a counter to index them in order. It has some additional functions to access the ManagedBuffers memory directly to register them with the SocketChannel to map one buffer ring one-to-one onto the buffers used by the SocketChannel for send and receiving respectively.

There are more classes either built on top of the ManagedBufferRing or exhibiting similar functionality that are grouped by the ManagedMemory interface. ManagedMemory is a rather simplistic interface. All it requires is that a class is capable of producing buffers, with no additional required properties. This could be extended upon in the future, structuring the memory management even further by having different ring structures implement different interfaces to mirror their exact behaviour and properties. One very generic

class, aiding in replacing the old memory management with the new one, is the FunctionalMemoryConverter. It can convert lambda objects into the new memory management structures and vice versa.

It is also obvious that configuration classes will instantiate connections to multiple peers. They will, therefore, will require classes akin to factories for structures like the ManagedMemory, which the ManagedBufferRing is an instance of. Responsible for this operation are instances of the classes SingleRingFactory and MultiRingFactory, with different implementations of the ManagedMemoryFactory interface. Also, to keep a smaller memory footprint, there's the option to use Singletons of those factories to ensure that each runtime only has access to a fixed, and constant, memory pool for message buffers. For the use without RDMA, both send and receive factories can be used to drastically reduce the amount of memory allocated.

4.2.2. Extending ByteBuffer

Ideally, the ManagedBuffer class could have been an extension of the ByteBuffer class. But the way ByteBuffers are implemented in Java 8, having exactly two classes extending the abstract ByteBuffer class, in the DirectByteBuffer and HeapByteBuffer classes makes it impossible to extend the ByteBuffer class without making additional efforts and leaving "Standard Java". This means that the ByteBuffer class will not be extended, which would result in a cleaner refactoring process with fewer method signatures changed and less boilerplate code written. It, instead, will be necessary to implement the ManagedBuffer classes as wrappers around DirectByteBuffers, which can interact with the RDMA library DiSNI.

The reason extending the ByteBuffer class would be impossible, is that the constructor of the abstract ByteBuffer class itself is package-protected, and even DirectByteBuffer and HeapByteBuffer are not directly instantiable, but instead need to be allocated by a static call to ByteBuffer allocateDirect or allocate methods. Some ways to circumvent this would be to use reflection to access protected symbols in the Java Standard Library. This would reduce performance because the compiler is able to optimize the ByteBuffer classes, especially, because it can be sure that only two kinds of extensions are possible to exist at runtime. Another way to extend the ByteBuffer class would be to create a class in the same package the ByteBuffer class exists in, which is against Sun's Java license agreement. Even then the community seems uncertain if such an undertaking would be successful [10].

4.2.3. Buffer Solution

The implementation of the classes described in chapter 3.1 strongly depends on the specifics of the ManagedByteBuffer class. The goal to achieve zero-buffer-copy is influenced by the earlier paper implementing RDMA in the Reptor framework [26] as well as other works arguing the requirement of zero-buffer-copy for efficient use of RDMA [7, 13, 8]. Although with small message sizes there is an argument to be made that a `memcpy` is faster than the overhead presented by the memory management necessary to support zero-buffer-copy

Class	Scope	Function	RDMA	Type
MemoryRingFactorySingleton	Runtime	Abstract class containing reference to one ManagedBufferRing	N/A	Singleton and MemoryFactory
SendMemoryRingFactory	Runtime	Singleton instance of MemoryRingFactorySingleton for sender ring	No	Singleton and MemoryFactory
RecvMemoryRingFactory	Runtime	Singleton instance of MemoryRingFactorySingleton for receiver ring	No	Singleton and MemoryFactory
ManagedMemoryFactory	N/A	Defines interface for obtaining access to rings	N/A	MemoryFactory
MultiRingFactory	Peer	Each instance is able to generate new ManagedBufferRings	No	MemoryFactory
SingleRingFactory	Channel	Each instance contains reference to one ManagedBufferRing	No	MemoryFactory
ManagedMemory	Channel	Defines methods for obtaining memory	N/A	Memory
FunctionalMemoryConverter	Function	Wraps function in anonymous class conforming to this structure	Yes	Memory or MemoryFactory
ManagedBufferRing	Channel	Manages ManagedBuffer in a ring structure	Yes	Memory
ManagedBuffer	Buffer	Defines ByteBuffer inspired interface	N/A	Buffer
ManagedByteBuffer	Buffer	Implements ManagedBuffer avoids buffer copy and is RDMA aware	Yes	Buffer
NativeByteBuffer	Buffer	Implements ManagedBuffer behaves like ByteBuffer	Yes	Buffer

Table 4.1.: Memory management class structure and inheritance model.

[8]. The memory management implementation is using buffers of a fixed size in a buffer ring, which are reasonably above the critical buffer size at which those considerations have to be taken into account.

The reason for implementing the ManagedBuffers the way presented here is that they can override and control certain operations to guarantee compatibility with RDMA and the implementation of zero-buffer-copy. The entire class diagram is shown in figure 4.1 with per class descriptions in the table 4.1 including the intended scope and RDMA awareness. The capacity of the internal ByteBuffer of any ManagedBuffer has to be equal to the buffer sizes used in the SocketChannel to allow registration of the buffers. For this reason, the Singletons at the top of the memory *stack* are initializing their buffer size with the static constant `BUFFER_SIZE` in the SocketChannel class and the length of the internal ring to `BUFFERS_NUMBER`. This way it can be guaranteed that the ManagedBuffers and even the ManagedBufferRings stored inside the singletons are able to interoperate with the SocketChannel. This `BUFFER_SIZE`, therefore, presents an upper capacity limit on any message to be sent using the framework. It also prohibits classes in the network layer to attempt to *grow* the buffer they are provided with. Though, this is no more limiting than the original RUBIN implementation as the outgoing messages were also limited by the capacity of the single send buffer in the SocketChannel.

The new implementation to transmit messages presents a way to achieve zero-buffer-copy with a constant memory footprint used for the send buffers. The functional allocation call was replaced with ManagedBufferRings and factories creating ManagedBufferRings. The exact number of unique buffer rings is dependent on the configuration, though in general, it follows the scope shown in table 4.1, with one transmit ring per channel. Replacing the call to the functional object is the invocation of the next method of the ring. In the simplest case, just the next buffer in the ring is returned. Before this can be done, the number of references to the buffer are manually counted. This also requires the owner of the buffer to invoke free on the ManagedBufferRing with the ManagedBuffer. Only currently free buffers are returned, otherwise, they are skipped. If the entire ring is in use, that means every buffer is occupied, an exception is thrown giving the choice of how to handle this state to the caller. On creation, the ByteBuffers backing the ManagedBuffers in the ManagedBufferRing, also replace the SocketChannels ByteBuffer array for send buffers, each ManagedBuffer is then assigned its corresponding send WR. Most of the network layer can then remain unchanged, with just the behaviour of the NetworkSinkBuffer changed substantially to avoid buffer copies, among other operations breaking the relationship between the backing ByteBuffer and the already registered send WR, stored alongside it, in the ManagedBuffer. In the final calls of the network layer, the ManagedBuffer is unwrapped into the ByteBuffer and the send work requests, which is subsequently posted by the SocketChannel. The limit of the ByteBuffer is then changed directly to allow interoperability with the old implementation. This also allows the existence of multiple buffer rings for sending, as all of their backing memory can be pinned and send WRs created. Those send WRs can then be posted once the buffer is to be sent.

To keep the memory footprint small, as few buffer rings as possible should be used, as allocating each of them constantly increases the used memory. Because of this, and the manual reference counting, the whole runtime can also safely share access to the same ManagedBufferRing using the SendMemoryRingFactory Singleton, to reduce the used memory and memory allocation calls.

The receive data flow direction is implemented similarly. However, the existence of exactly one ManagedBufferRing per SocketChannel is important. This is taken care of by the structure in which the ManagedBufferRings are created (ManagedMemoryFactory → SingleRingFactory). Analogous to the registration of the send buffers, each SocketChannel can be provided with the ByteBuffers of a ManagedBufferRing to replace its receive buffers, before registering them for RDMA receives. This can be further simplified by allowing the array of ByteBuffers allocated and registered in the SocketChannel to be outwardly represented by a MangedBufferRing. Using the buffers allocated at the initialization of the SocketChannel, a ManagedBufferRing can be created referencing those buffers wrapped in ManagedBuffers. Then, when returning the buffers to travel up through the stack of layers in Reptor a reference is counted and when the message is finally marshalled into a PushMessage the reference is released and the buffer is free to be posted again.

Though the benefits of an implementation like this have to be weighed, accounting for the overhead of the additional memory management structures and the reduced time spent allocating memory and garbage collecting, this implementation allows for easier integration with RDMA.

Future work could include the use of different SocketChannels for differently sized messages or even a different communication method or channel for sending flow control messages, as they pose the exception for may of the consideration here.

4.3. Receive Window

The receive window, inspired the flow control in TCP, described in chapter 3.2 fundamentally mirrors the receive window implementation found within the TCP protocol, with a few omissions, due to guarantees and features RDMA provides. It is used for the exact same reason TCP has flow control. The receiving side of any communication has to have buffers prepared to receive the entire message before they can process the incoming buffers. This creates the back pressure. If there are no buffers prepared, because the sender is faster at sending messages than the receiver is at consuming and preparing buffers, a system can enter an error state or messages are lost.

Instead of requiring the protocol built on top of the Reptor framework to be aware of the possibility of flooding its peers, the implementation presented in this thesis is entirely contained in the network layer. Most of the logic required to maintain a receive window can be moved to the lowest layer of the send and receive calls. In the case of Reptor, modified to use a Selector using RDMA sockets, this means the implementation of the receive window logic can be moved inside the SocketChannel class.

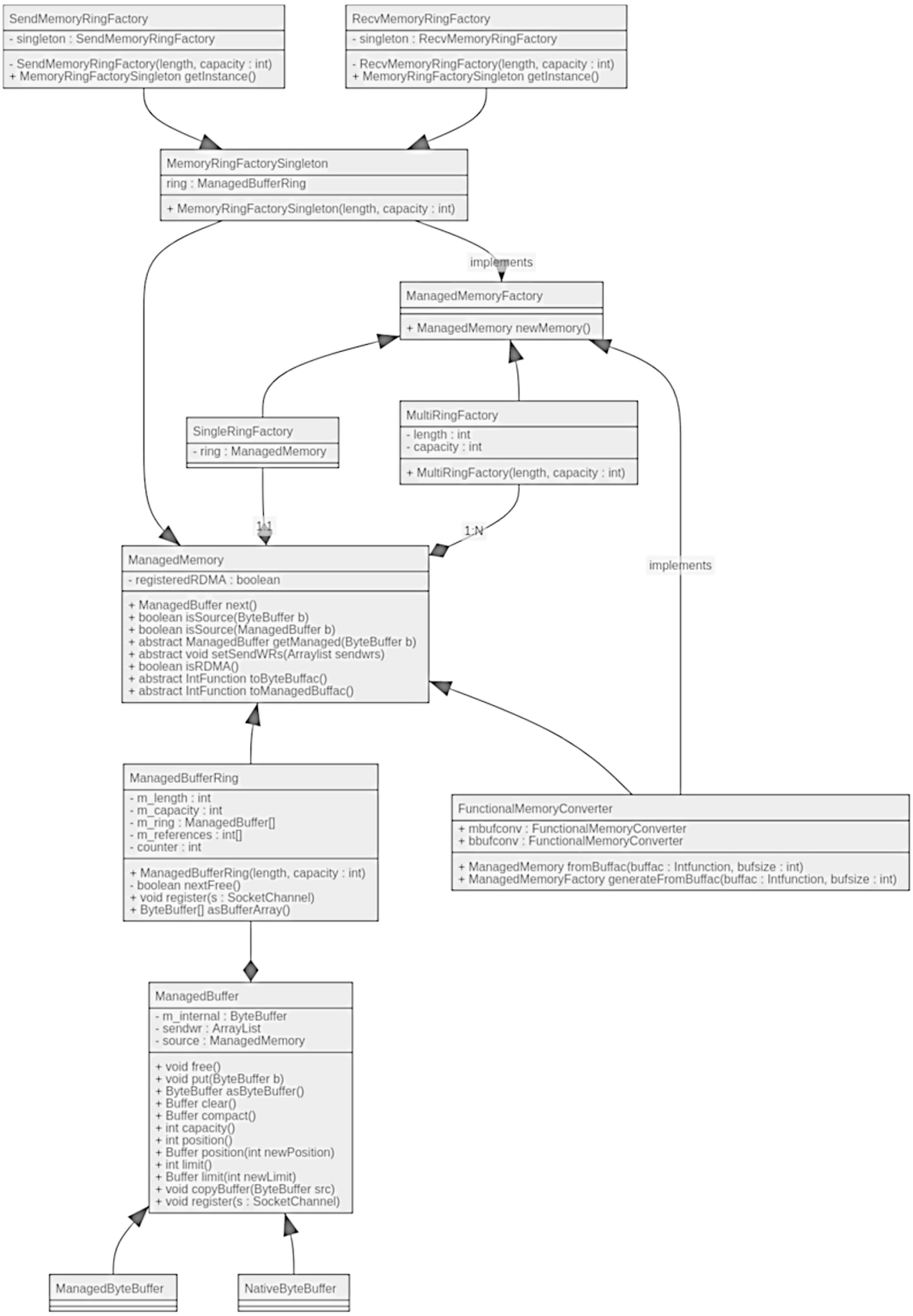


Figure 4.1.: Complete class diagram showing the classes managing access to memory and their composition.

As mentioned in the design chapter, this goal can only be achieved to some part, without changing small parts of Reptor and RUBIN. Specifically, the Selector of RUBIN needed to be changed to allow signalling about the current flow control state, that is whether or not a SocketChannel is allowed to send or not. It already was used to determine whether a SocketChannel has data to read, so adding this was not especially difficult. Reptor and the different layers within, as well as the protocol running on top, did also have to be able to either react, or the layers needed to handle this new possibility for layers above themselves. This meant to either escalate the exception when trying to write when the Selector is unable to select a writeable SocketChannel, or to buffer the messages until the SocketChannel is allowed to send again. To test the difference between the two strategies, both have been implemented in Reptor, with the escalation of exception strategy being the one that was used for further development because of the zero-buffer-copy target.

First of all, there are some defining characteristic values of the network layer which determine the behaviour of each connection. Defining the maximum amount of memory used by the network layer is the number of buffers reserved for the buffer ring `BUFFERS_NUMBER`. Not impacting the actual flow control implementation but effecting its performance due to latency and total transfer duration for each message is the capacity of each buffer. Due to the memory management implementation, the capacity of each buffer needs to be the same for each connection. There could, however, be multiple different connections between two replicas which use different buffer sizes.

The flow control scheme is based on the fact that both replicas are attempting to replenish message buffers as soon as possible. The method with which this is done is by posting the receive requests for a batch of buffers, sequential in the buffer ring if the current number of usable buffers approaches a threshold. This defines two further characteristic variables: `BUFFERS_BATCH`, the number of buffers in a batch, and `BATCH_OVERLAP`, the threshold for when to post another batch.

We also assume that throughout the communication rounds all non-faulty replicas use the same or at least compatible values for these variables. This mostly concerns the buffer capacity, using a larger one to send or a smaller one to receive will cause a faulty QP state.

4.3.1. Flow Control algorithm

The diagram shown in figure 4.2 shows an example communication following the flow control structure implemented, alongside notes showing the client-side calculations taking place. As described in 3.2 the state of each peer regarding the flow control can be reduced to a tuple of numbers characterizing the state of the remote peer and a method for querying the local window size. Using the same naming scheme as the diagram, the first numerical variable is `pp` short for "peer prepared". The `pp` variable is kept equal to the number of buffers the remote replica is prepared to receive. Every time a message is sent to that replica `pp` is decremented to reflect the consumption of that buffer. The second numerical variable is `pk`, short for "peer knowledge". This variable is updated to keep in sync with the `pp` value of the remote replica. To do this `pk` is decremented with each incoming message. This is very trivial compared to the algorithm necessary to implement a similar feature in TCP, because our RDMA QP is a reliable connection, and each message is guaranteed by the network protocol to be sent exactly once.

The part of the communication structure that relieves the back pressure of the system, is the following condition: a peer is only allowed to send if its `pp` is above a certain threshold, that is that the peer knows that it has to not exhausted all prepared buffers of the remote peer. In the actual Java implementation `pp` and `pk` are already adjusted by the threshold, changing the condition to $\texttt{pp} - \texttt{threshold} > 0$. Also, in combination with the Selector structure, this means the SocketChannel has to report not to be writeable when this condition is met.

Once a message is exchanged, the current number of locally prepared messages (`pm`), alongside all necessary information to recon-

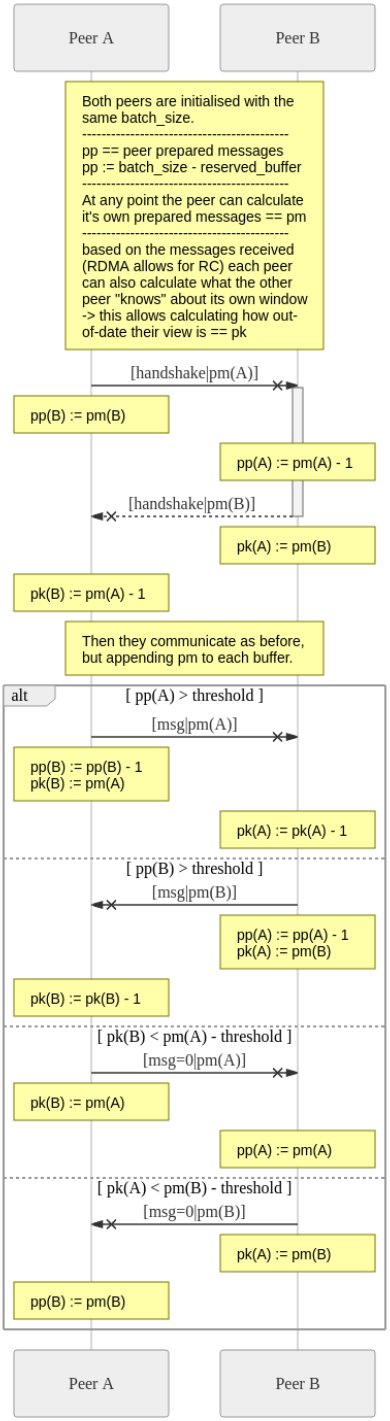


Figure 4.2.: Sequence diagram showing complete implementation of flow control

struct the original buffer before the additional information was concatenated, are appended to the buffer. This flow control information is received on the remote peer, which will update `pp := pm`, and the peer sending the message updates `pk := pm`. If any peer becomes aware that the remote peer is about to starve, that is `pk` approaching the `threshold` value, it sends a flow control message, initiating the update process. The only static requirements of the network layer for this flow control scheme to function, is that the `BATCH_OVERLAP` of batches, as well as the size of batches, and in extension the number of buffers available, are larger than the threshold chosen. The threshold is necessary to allow the exchange of flow control messages if a peer has starved of *slots* for normal messages. For the sake of this thesis the threshold is a static value unrelated to any other variable, generally set to five. Arguably determining a more performant threshold value adaptively could increase throughput and overall performance.

This implementation also requires no additional communication rounds and the additional information appended to each buffer is only 5 bytes long. The necessary information to be transmitted is the number of prepared buffers, in addition to whichever information is necessary to *reconstruct* the original buffer. In the current implementation, 4 bytes are used to transmit the limit of the original buffer. The limit of a buffer in the network layer represents the end of the written data. As the flow control data is appended to the end of the buffer, information about the position of the original limit would be lost. As an integer in Java is always 4 bytes long, the limit is appended as its 4 byte long representation. The number of prepared buffers is represented by a byte. The theoretical maximum number of preparable buffers is hardware dependent. In the case of our machines, it is 16351. A byte is not enough to accurately represent all integers between 0 and 16351 which means the byte is to be interpreted as a code word. And accurate representation is also not required, as the number transmitted has to convey an upper bound the remote is allowed to send. The simplest code would be $\texttt{min}(\texttt{pm}, 255)$. As we allow the machines to share magic numbers before starting to communicate, an arbitrary linear multiplier can be chosen. Even if only an approximation of the prepared buffer number can be sent, because both the local replica as well as the remote then update their variables to the interpretation of the number, the local replica will be aware of the synchronisation status of the remote. With the default code, sending the amount of prepared buffers, it can be experienced, that one replica sends 255 messages, then waits for the update from the remote and sends another 255 messages. This cycle can repeat multiple times. Even though the receiving replica may have many buffers prepared, it could just takes too long to update the remote before it exhausts it's allowed window.

An implementation using an integer to transmit the accurate number of prepared buffers, instead of the clamped version from 0 up to 255, was tested. It was observable that fewer control messages needed to be sent, but no measurable impact on performance was caused. We hypothesize the 8 bit codeword is reasonably powerful enough to map to the field of possible values. As a compromise, using a short typed value could be investigated on its impact.

There is a large space of possible implementations using this flow control scheme, which could be investigated further. Additionally, using direct writes and direct reads for reporting one replicas state to a remote replica could be used to separate BFT messages from flow control state. This could reduce the latency on control messages and, thus, reduce the amount of time a replica is waiting to be allowed to send again. Writing the local flow control state to the remote has been done by Dragojević et al. [7], and in Reptor it could even help simplify the interface between flow control and memory management.

4.4. Testing

Along with the implementation of the memory management classes, multiple unit tests and preliminary benchmarks were written. Existing unit tests can be used to check for the correct behaviour of the classes implementing ManagedBuffer, as well as the ManagedMemory classes. Structurally more abstract classes, like the Singletons and the FunctionalMemoryConverter, are also tested for their functionality and correctly entering the failure states.

A larger portion of the work done introduced multiple new exception classes and behavioural changes to catch-clauses on already existing Exceptions. This was necessary because of the stacked structure of Reptor, which allows for almost arbitrary configurations and, more importantly, arbitrary layers to be neighbouring each other. Having ManagedBuffer, ManagedMemory and other introduced memory management classes throw custom exceptions enables safety that all edge cases and failure states are handled, because they can be checked at compile time.

Initial benchmarks, determining the time saved by using the ManagedBufferRing and reusing the ByteBuffers, as compared to allocating new direct ByteBuffers, clocked in at about 7 times faster.

4.4.1. Expectations

The amount of performance increase is strongly dependent on the utilization of the CPU. In reference to the diagram of figure 4.3, zero-buffer-copy is the most performant if the idle time of the CPU is close to zero. In a networked protocol, this means zero-buffer-copy can only increase the performance based on the weakest link present in the network, as if a peer is faster at consuming messages than the remote peer is at sending, the time idling could be used allocating and copying buffers without any deduction in Rounds Per Second (RPS) achieved. This is why synthetic benchmarks show a speed increase unrealistic to expect in a real application.

4.5. Discussion

This subchapter will be used to discuss the use and availability of RDMA, as well as some assumptions made throughout the design process.

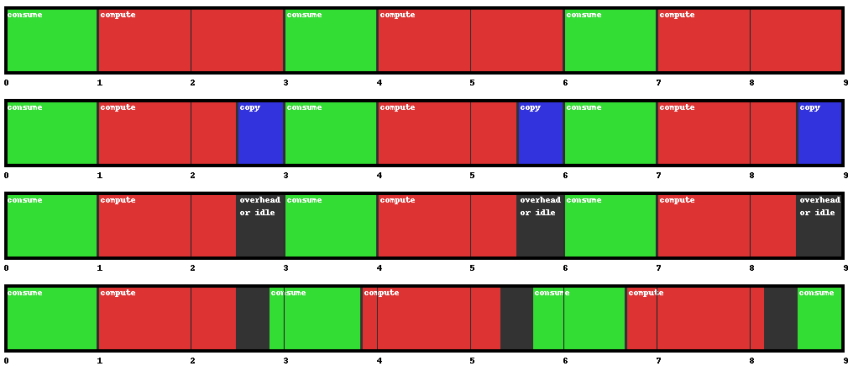


Figure 4.3.: Diagram showing how full CPU utilization is needed to benefit from zero-buffer-copy. The network speed needs to be the bottleneck.

4.5.1. Availability of RNICs

InfiniBand is a technology which currently only sees widespread application in data and research centres. Other protocols and implementations, like the Internet Wide Area RDMA Protocol (iWARP) and RDMA over Converged Ethernet (RoCE), suffer when used with low-end network components or require a special setup [19, 18]. The fact that they can operate in existing Ethernet networks is favourable in some applications, enabling RDMA in consumer, and non-specialised, networks. iWARP, in particular, is interesting because it delivers RDMA services over TCP/IP. According to Mellanox Technologies [19], this dooms iWARP, as it forfeits RDMA's core advantages, high throughput, low latency, and low CPU utilization, by doing so.

iWARP's goal is to enable RDMA over existing hardware and infrastructure. It makes few assumptions about the network and its packets can be routed like any other TCP packet. Thus, iWARP involves the entire TCP stack which, Mellanox Technologies [19] claim, prohibits true and scalable RDMA operations. To transport RDMA iWARP stacks multiple network layers on top of the TCP/IP layer, whereas RoCE, it's direct competitor, uses UDP/IP frames with fewer different technologies stacked on top [19]. Instead of using RDMAP, a separate RDMA protocol, Direct Data Placement (DDP) and Marker PDU Aligned framing (MPA) on top of TCP as iWARP does, RoCE only applies the IBTA Transport Protocol [19]. This defines one of the larger differences between the two protocols. iWARP, building on top of TCP is guaranteed reliability, whereas RoCE using UDP has to gain reliability through another way. This is commonly achieved through converged Ethernet which transforms the Ethernet connection itself into a reliable connection, but also requiring special hardware.

RoCE and iWARP are compatible in software. This is because both implement the RDMA Verbs interface through the Open Fabric Alliance Stack. They can either be run as a kernel implementation, enabling larger portability, or using the hardware of RNICs, profiting from increased performance. They are each considered the standard for RDMA by different companies and institutions [19].

4.5.2. RDMA for consensus

Touched on, in the section Security Analysis of chapter 3, while RDMA provides many security features, some of its prerequisites to function efficiently have to be evaluated on their security impact. First of all, as shown by other work in this field, like DARE by Poke and Hoefler [23] or FaRM from Dragojević et al. [7], RDMA can be used to create trusted environments and even aid in the development of RDMA protocols [1]. From the requirements of the different RDMA implementations in 4.5.1, one has to assume that replicas connected through RDMA are not only in physical proximity to each other. To keep the reliability of the connection between replicas, and to not hurt the latency and throughput, the replicas have to be connected to the same network switch or local area network, at best. This provides a single point of failure, which needs to be considered when deploying an RDMA based distributed service.

5 Evaluation

The system and integration benchmarks were run on machines with 2 sockets using Intel® Xeon® CPU E5-2430 v2 @ 2.50GHz with 6 cores per socket and hyper-threading, totalling 24 threads. The system has 16 GB DDR3 RAM available across 4 DIMMs. RDMA is used through OpenFabrics Enterprise Distribution (OFED). OFED is an open-source software for RDMA and kernel bypass which is loaded as a kernel module [21]. The version used is OFED-4.2-1.2.0. The RNIC used for RDMA communication is a Mellanox MT27520 Family ConnectX®-3 Pro connected through PCIe Gen 3.0 for a theoretical maximum data throughput of 8 GT/s, however, the combined throughput over the InfiniBand lanes is at most 56 Gb/s [18], with 14.0625 Gb/s per lane [17].

The used system is running 64-bit Ubuntu 14.04.5 LTS Trusty Tahr using the 3.13.0-147-generic kernel, the operating system is not virtualized.

For benchmarking and evaluating logfiles, new python scripts were written which can be used to interpret, analyse and visualise arbitrary logs written by Reptor. These are used alongside existing benchmarks and tests to collect data for this chapter.

5.1. JUnit benchmarks

JUnit is a Java framework for writing repeatable tests [12]. The project makes use of JUnit version 4.12. There are multiple unit tests written using JUnit probing for proper behaviour of the implemented memory management structures and, to some extent, also the flow control.

JUnit 4 supports the creation of benchmarks as well as tests when combined with the JUnitBenchmarks plugin [29]. That combination was used to benchmark the performance of the memory management implementation. Most specifically, the ManagedByteBuffer and the ManagedBufferRing, as all other classes are either based on either of those or are not vital for system performance. For example, the FunctionalMemoryConverter can be run once, on start-up, the Singleton classes are identical to the ManagedBufferRing, except for one additional getInstance call.

The benchmarks, written for the ManagedByteBuffer and ManagedBufferRing, aim to compare their performance to the native Java ByteBuffer. As almost all of Reptor will interact with the ByteBuffer wrapped inside the ManagedByteBuffer, the interesting metrics to take are the overhead required to create the ring structure, the time save gained by reusing memory instead of reallocating and the reduction in time spent running the garbage collector.

Type	Iterations	Write	Round [s]	Dev. [s]	GC calls	GC time [s]
Native	10^3	No	0.09	0.04	10	0.03
Ring	10^3	No	0.08	0.02	12	0.03
Native	10^5	No	9.62	0.56	1142	3.21
Ring	10^5	No	0.10	0.02	14	0.05
Native	10^5	Yes	10.6	0.72	1142	3.48
Ring	10^5	Yes	0.10	0.02	14	0.04

Table 5.1.: Results of JUnit performance benchmarks of ManagedBuffer and ManagedBufferRing compared to Java's native ByteBuffer performance. One iteration corresponds to either 1000 `ByteBuffer.allocateDirect` or `ManagedBufferRing::next` calls.

The benchmarks, designed to measure those metrics, are averaged across at least 20 runs. All benchmarks are run single threaded. Because the round times of a single allocation and a single ring initialization are so low, the smallest benchmark starts comparing 1000 ByteBuffer allocations of 1 MB to fully iterating through a buffer ring of length 1000, of buffers with the same capacity. This is repeated 20 times. In this benchmark, a round of native buffer allocations take 90 ms ($\mp$ 40 ms), with 10 garbage collection calls, spending 30 ms in garbage collection. The ring buffer benchmark has a round time of 80 ms ($\mp$ 20 ms) with 12 garbage collection calls, also spending 30 ms in garbage collection code. The buffer ring implementation always seems to be slightly faster. This can be explained by the JVM keeping necessary prototypes loaded in memory to reduce access and instantiation times, this, however, does not fully explain why it is not capable of doing the same for the ByteBuffer benchmark.

To gain more accurate results and to investigate how the ring performs over a longer system runtime, and to rule out that the Java compiler optimizes certain parts of the buffer rings code away, if the buffer is never written to, two other types of benchmark are conceived (see B). First, the amount of iterations through the ring are increased to 100 times, also multiplying the number of native ByteBuffer allocations by 100, to be able to compare results. To avoid compiler optimizations, *messing up the results*, an additional type of benchmark obtains the buffer, from either the ring or a direct allocation, then writes to it and frees the buffer again. This behaviour simulates the filling of a message buffer for sending with RDMA.

The results of the benchmarks shown in table 5.1 are very positive. Most notably, the benchmark allocating new ByteBuffer objects spends 87 times longer in garbage collection code and jumps into garbage collection about 81.6 times more often. This performance advantage will only increase the longer the buffer ring stays in use compared to the allocation of new buffers. Also, the small difference between the benchmarks, whether or not the buffer is just allocated or also written to, shows that the compiler does apply some form of optimization to the buffer ring structure, if the buffer is not written to.

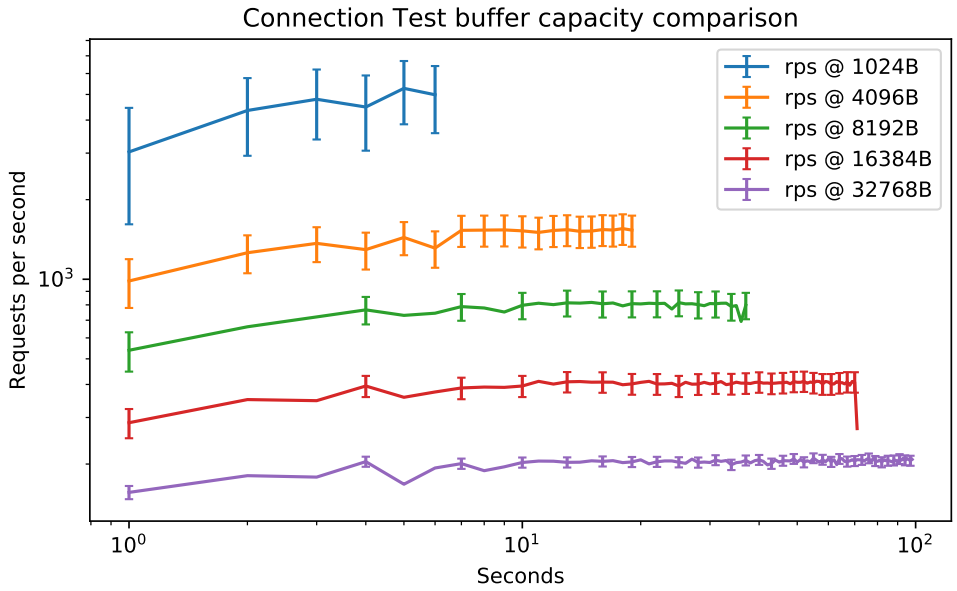


Figure 5.1.: Distributed connection test evaluation on the development branch. With the server running on beagle1, and the client running on beagle4. Restarting both for each new benchmark and compiling both the client and the server for new buffer sizes.

5.2. Connection test

There exist multiple development versions with differing behaviour to compare. There exist some test applications written in Reptor, both to test its own function, as well as, to measure its performance it more realistic environments. For benchmarking of BFT protocol implementations, deployed in a replicated state, there is the replicated system benchmark 5.3. For a simpler setup, which can be launched distributed across different machines more easily, and performs a simple ping-pong conversation, the connection test can be used. The connection test is an application built on top of framework. There are separate server and client implementations, which, after a message has been received respond as quickly as possible, entering a loop. This is continued for up to 99 seconds, and the achieved RPS for each interval are logged. The messages also contain a counter, enabling both the client and server to check that messages arrive intact, and in the right order.

In the most advanced branch, the flow control scheme ensures no crashes because of the RDMA Queue Pair. It can be experienced that one of the peers receives a buffer filled with zeros, which stops the connection test server from serving any more messages. The distributed connection test benchmarks were taken on beagle1 and beagle4.

The distributed connection test runs were undertaken with 7000 buffers per buffer ring at different capacities (figure [5.1]). They are batch posted by the SocketChannel in groups of 1000, and batching is done 300 buffers before the current batch runs out. Out of the

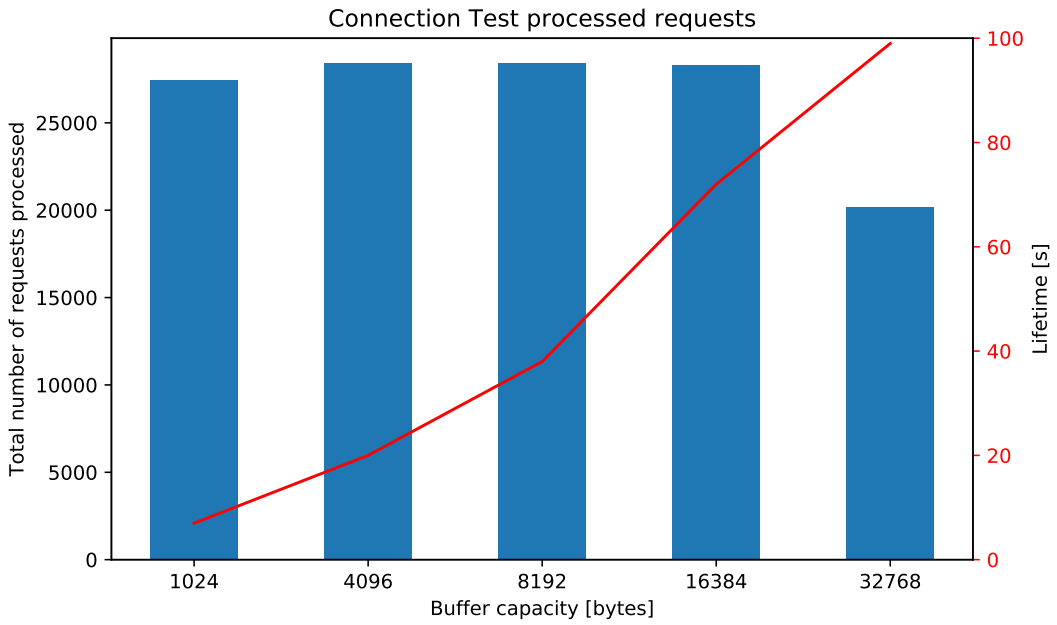


Figure 5.2.: The columns show the total number of requests processed throughout the connection test. Shown by the red line is the lifetime of the connection test at the specific buffer size.

7000 buffers, five buffers remain reserved for control messages. Over the first six seconds, an average number of 4505 RPS is reached using 1 KB buffers. The numbers were chosen to produce representative results, the performance of the implementation as well as the stability is, however, heavily influenced by the numbers chosen. Some, especially with a greater total number of buffers, perform far better, even better than the implementation without the flow control using the same configuration. Higher RPS are achievable dialing the total number of buffers as well as the buffer size down 5.1. In terms of stability, the system gains stability by increasing either the number of buffers or the capacity of the buffers, as seen in figure 5.2.

This configuration is also chosen to force control messages to be sent, as, on average, every second the server sends a control message to the client to update its counter, as it otherwise would stop sending. This imbues confidence that the flow control algorithm works, especially because the benchmark runs for far longer and uses more buffers than a single batch or even the entire buffer ring. Also noteworthy is that more than the five buffers reserved for flow control messages can be sent, as those five are replenished as soon as another batch is posted. It is observable that, this kind of benchmark still is not stable. The symptom occurring is, that seemingly dependant on the buffer size, at some point a buffer filled with zeroes is received. This is strange, as all buffers are appended at least the flow control information. This causes the flow control implementation to send

flow control messages, as the reported prepared size is zero. The buffer can even be ignored, but the benchmark will stop at that time because a response is waited for. The new error type is not crashing any replica, not even the channel but halting the transmission of new messages after the empty buffer has been received. It is also obvious, looking at the logs, that the end of execution is not caused by some sort of deadlock of the flow control mechanism, as both replicas exchange flow control messages and update their state successfully. They recover in terms of flow control, and would be ready to send again, if the network layer received another message to send. This is a rather unsatisfying interpretation, but Reptor was not designed with the possibility of failing to send or having to resend a message, which now causes existing benchmarks to halt. The solution to this is to buffer the messages as they come in. This is not an option when also applying zero-buffer-copy.

In this benchmark, the baseline RDMA implementation far exceeds it in performance with an average of 279525 RPS served. This, however, exhibits the data race inspiring the implementation of flow control. The RDMA baseline also is likely to crash with RDMA QP Exceptions, running the system benchmark.

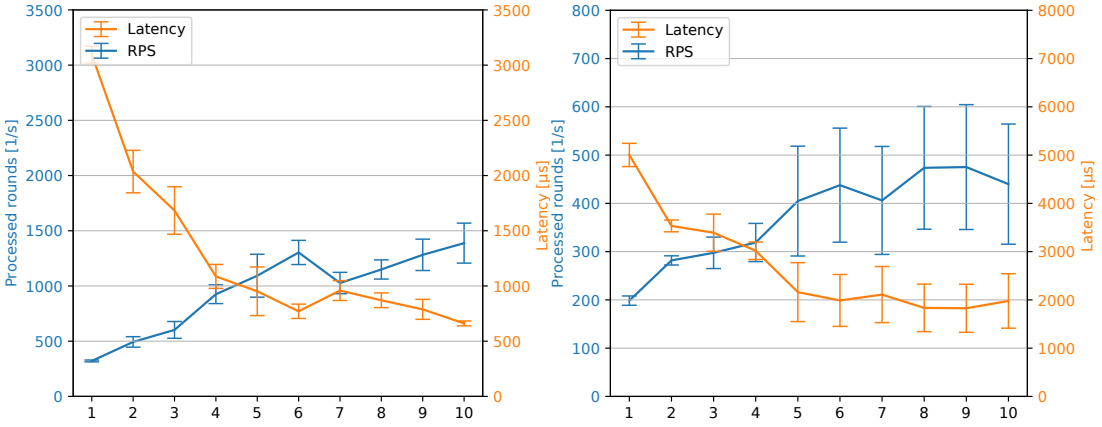
5.3. System benchmark

Similar to the Connection Test 5.2, the replicated system benchmark, also called python benchmark, has differing results depending on the version of Reptor used.

The most stable version, implementing both zero-buffer-copy and flow control, using the most simple settings, running locally, achieves 484.6 RPS with standard deviation of 16.3 (figure [5.3b]). This is compared to 1153 RPS, with an average standard deviation of 91, using the RDMA baseline version, with identical settings (figure [5.3a]). The system benchmark can be configured to start distributed too, however, the results here show the local benchmark results. It is replicated, because it spawns a number of replicas when launched. A client can then send messages to the replicas, to receive responses. The system benchmark, in particular, also launches clients, which send requests with zero values, to receive responses from the replicated system.

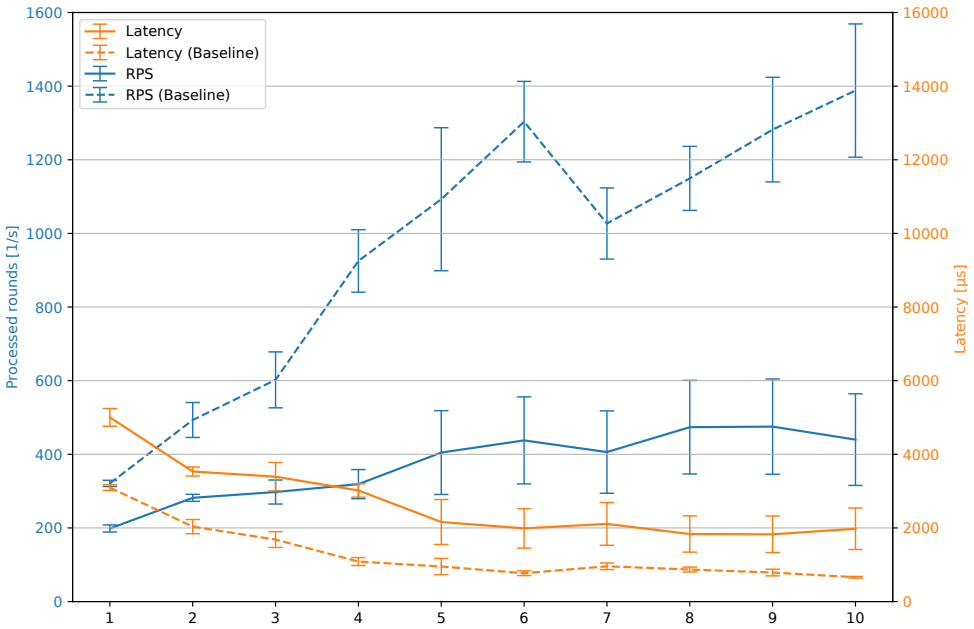
During the benchmarks of the RDMA baseline implementation, on each run of 10 seconds, at least one replica crashed, due to a unrecoverable QP state. These crashes are not evident at all in the new implementation.

The benchmarks do not use the parallelized configuration COP [3]. The benchmarks use one client and run the PBFT [5] protocol. It is possible to serve many clients. The reason the benchmarks do not use the parallelized configuration, is because part of its scheduling behaviour seems to break the one-to-one mapping from the receive buffer ring to the socket channel, over the lifetime of the benchmark, which causes an RDMA memory fault. The measurements in 5.3 show that the metrics of RPS, as well as, latency, have worsened in response to the implementation of flow control. The RPS achieved has decreased to 42%, and the latency roughly doubled. After letting the system stabilize for a few seconds, a stable latency of around 2 ms is reached. This is likely due to the introduced overhead of flow control.



(a) Baseline implementation using RDMA

(b) New implementation with flow control and zero-buffer-copy using RDMA



(c) Comparison overlaying RDMA baseline

Figure 5.3.: Replicated system benchmark result plots for the baseline using RDMA and the new implementation. Averaged across 10 runs with otherwise identical configuration. It is observable that the time saved in memory management through zero-buffer-copy is not outweighing the overhead added by the flow control.

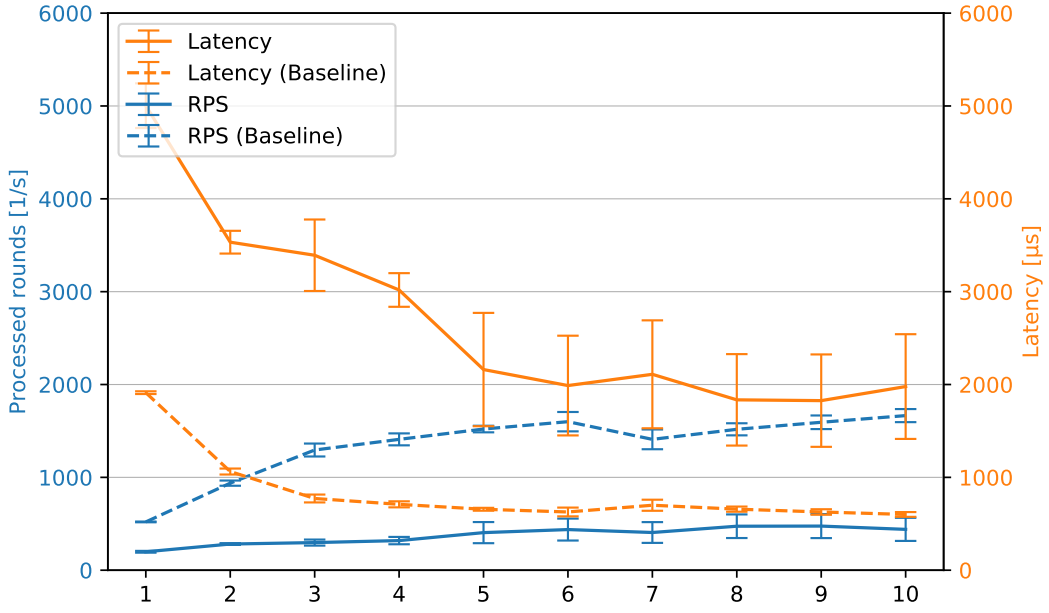


Figure 5.4.: Replicated system benchmarks comparing the new RDMA implementation to the baseline Reptor implementation using TCP. The benchmarks are configured identically otherwise.

These results using RDMA can also be compared to the original Reptor implementation, using the Java Network Interface based on TCP. The diagram 5.4 shows a similar trend across both implementations, but it is clear that the RDMA implementation of the network layer still does not utilize the strengths of RDMA. A possible reason for these results could be a performance oversight in the flow control implementation. Though, keeping the flow control scheme performant, was an active goal of this thesis, and best efforts were made to reduce overhead where possible. Another reason for this could be that the TCP Reptor benchmark is capable of sending smaller sized packets, whereas the RDMA implementation is limited to sending same sized buffers, which have to be at least as big as the largest packet that could be sent by the benchmark. Thus, more data needs to be transferred over the network in the RDMA benchmarks.

5.4. RDMA optimizations

There are further optimizations available. They can either be enabled through the DiSNI library or by changing how the DiSNI library is used in the SocketChannel and, in extent, Reptor.

5.4.1. Doorbell batching

RDMA itself provides the option to batch notifications for completed work requests. This means, an arbitrary number of requests can be completed and the notification about their completion can be sent at once, saving bandwidth, thereby, freeing the data channel for

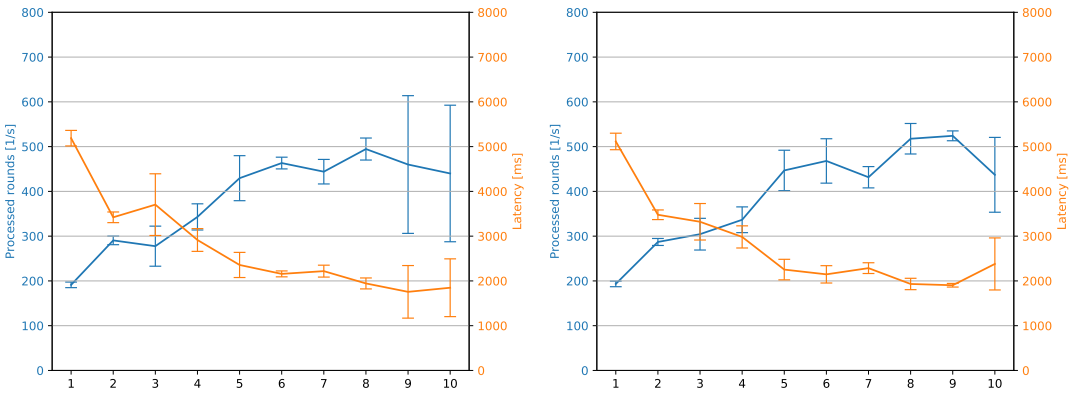
application data [25]. Enabling this optimization does interfere with the flow control implementation that determines the number of messages a peer is allowed to send to its remote. This could be circumvented by changing the flow control scheme to take this batching of notifications into account. The BFT protocol running might need to be synchronized with the network batching too, especially, if there is no timeout. As a result of testing batching of notifications, it seemed only possible to choose the number of buffers reserved for control messages as the batching size for notifications, and only if the batching size of the SocketChannel class was a multiple of it. A similar problem is encountered by another optimization method. The posting of work requests is always done by posting a list of work requests with the RDMA library. This allows for posting multiple work requests at once, instead of just a single request at a time. The latter is, however, more alike the behaviour of conventional communication protocols, thus, we post work request lists of length one, containing a single work request instead of multiple, because the network layer cannot expect any properties from the BFT protocol and a timeout approach was tested but did not deliver a working solution. The performance that could be gained, stems from the overhead added by the memory-mapped I/O (MMIO) step of posting a send request. The complexity of the MMIO step does not grow with the number of messages sent, thus, *batching* in this sense also can be used to improve the performance further. The posting of multiple WRs at once has further advantages, summed up in the so-called *Doorbell* method [13]. The current implementation for sending messages, called *WQE-by-MMIO*, is more portable but is inferior in performance [13].

The doorbell method is used for posting the receive requests for each buffer batch by the SocketChannel.

5.4.2. Message inlining

RDMA has the feature of inlining messages. Inlined messages reduce the amount of communication necessary to transfer the data between the RNICs. Therefore, a better performance, most notably, due to reduced latency, should be achievable [25]. This, however, comes at the cost of reducing the number of CPU cycles available to the application. Inlining messages works by using the CPU to read the data instead of the NIC [25], which saves an extra PCIE DMA transaction. It therefore is only reasonable to enable inlining for smaller message sizes, as otherwise the time saved, by avoiding the PCIE DMA transaction, does not make up for the cost of involving the CPU.

We tested the performance impact of message inlining. The averaged results across ten benchmarks each, comparing no inlining whatsoever to inlining all messages, are shown in the diagrams of figure 5.5. Inlining is only supported up to a certain size of message but involves the CPU more [25]. Enabling this feature is rather simple, as we do send buffers of identical capacity with each message, because of the zero-buffer-copy approach, this means, that either all messages, or no messages, will be inlined. Inlining didn't show a notable difference in the benchmark results, with both throughput and latency falling within margin of error.



(a) No inlining

(b) Full inlining

Figure 5.5.: Measuring performance impact of inlining messages using the replicated system benchmark. These benchmarks are identical to the system benchmarks shown in 5.3b, except that the buffer capacities are reduced to 512 bytes. This is to allow direct comparison between inlined buffers and sending the buffers normally.

6 Related Work

The advantages of RDMA over TCP inspire multiple exercises to make use of it. This can be done in a more general approach like in this thesis, providing a framework to work in, to transparently make use of RDMA. Other works choose an approach that includes developing new technologies based on RDMA, which fully utilize RDMA's features as part of their design. Both strategies are useful to uncover the capabilities that RDMA can offer in practise.

In the space of RDMA consensus protocols the focus of previous work has been to develop protocols which utilize RDMA communication methods out of the box. As to fully utilize RDMA, an argument can be made that, the communication protocol needs to be designed with RDMA in mind. Some properties of RDMA can even be used to simplify the protocol itself, as shown by Aguilera et al. [1].

6.1. DARE

DARE, proposed by Poke and Hoefler [23], is developed to make use of what RDMA has to offer. Instead of just accelerating the communication rounds inside the BFT protocol, they also made use of RDMA to speed up more processes related to the replication process itself. Their consensus protocol, due to the consideration of RDMA as the communication protocol from the beginning, is able to fully utilize RDMA and avoid the complications outlined in this thesis. To circumvent the network delay their communication scheme is synchronous.

6.2. FaRM

FaRM, short for Fast Remote Memory, is a distributed computing platform capable of running large scale networked key-value stores at incredible performance [7]. Their approach places the entire key-value store across the main memory of all participating systems. All the participants of their network form a shared address space of 2.8 TB of DRAM, separated into 2 GB chunks addressed by a consistent hashing algorithm. To ensure consistency they use transactions across the shared memory regions with logs and perform regular backups to disk in case of a crash.

They also provide a programming model which uses lock-free reads for maximal performance where possible. It could be used to not only implement key-value stores but also other memory intensive distributed services. The approach they use to relieve back pressure applies two strategies. First, they poll the end of buffers posted for receives, which are known to change when the write is complete. This allows them to use RDMA writes without notification. Their send procedure is similar to the one presented in this the-

sis. Each sender knows the location of a head pointer, similar to the number of prepared buffers, of the receiver and never writes over it [7]. The sender needs regular updates of the receivers head pointer to be able to keep sending. Their approach allows the receiver to directly write its own head pointer position into the senders memory with RDMA.

6.3. RDMA for Agreement

Aguilera et al. [1] have shown that, using the specifics of RDMA, a consensus protocol with tolerance $n \geq 2f + 1$ can be devised. Though this metric is also achievable through other means, like mapping Byzantine faults to crash faults by providing additional safeties like SGX [4], they make use of the shared-memory and dynamic permissions [1]. Their goal is to create a resilient and performant consensus protocol. Using the dynamic permissions of RDMA, they can prevent Byzantine processes from overwriting memory. To do so they apply a model, very close to the one we have implemented and devised, which shares similarities with DARE and Paxos [1, 9]. The Message-and-Memory (M&M) model both allows for high performance through asynchronous access as well as robustness to failures [1].

6.4. Hidden cost of RDMA

Frey and Alonso [8] present the technology that is RDMA as the specific tool it is. Strongly influencing the decisions made throughout this thesis, their benchmarks and general guidelines have shaped parts of the flow control and memory management scheme. Similarly arguing the necessary structure to utilize the feature set and advantages of RDMA, Kalia et al. [13] lay out general guidelines to achieve high performance, through the use of RDMA. More closely related to the topic of this thesis, Jalili et al. [11] show the requirements for high performance state-machine replication. An instance of such high performance state replication making use of RDMA has already been mentioned in DARE 6.1.

7 Conclusion

Using a faster communication protocol is only as efficient as the additional logic it requires to make it work. Any communication protocol for networked computers is going to require some form of flow control, as reserving memory for user space applications to receive data into, or the memory the operating system or network card require to initially receive data into is going to present a bottleneck. One that, if not handled properly, will mean that messages will be lost. In the case of RDMA, this buffer does not lie within the network card or operating system, but the same problem exists. There are multiple possible flow control algorithms, with different scopes. Just for solving the problem presented by back pressure, many algorithms and naïve solutions can be used. The performance of each solution also heavily depends on the application and application's communication model. From simple one way file transfers to BFT algorithms, different solutions come with advantages and disadvantages in performance, memory footprint, ease of implementation and safety.

To achieve the highest performance the system has to utilize its subsystems to its fullest. This includes technologies like DMA, which allow the CPU to perform work, whilst other parts of the system perform I/O actions independently, and in parallel. Data copies consume processor cycles without advancing the logic of the software, the user of a system wants to run. RDMA presents the networked version of this strategy, offering an opportunity to minimize the amount of buffer copies, freeing the CPU for other tasks.

RDMA is a technology, which if combined with a flow control scheme efficient for the application, can offer a performance increase for almost all types of applications. If the hardware and conditions necessary for its application are present, it not only speeds up the data transfer itself, but also relieves stress on the CPU. This can be taken to an extreme if the communication protocol is developed with RDMA in mind, with its asynchronous nature. At this point, it seems that RDMA, however, can not fully replace traditional communication protocols like TCP/IP, not only due to the necessary hardware but also the required flow control which needs to be designed and tweaked for each use case.

One limiting factor in measuring the performance of RDMA in a networked protocol is that any free CPU cycles can always be used to copy buffers. This means that if the CPU is able to do the buffer copies in between the intended task the system has to perform the performance gain of avoiding any buffer copies becomes unnoticeable, making utilizing RDMA to it's fullest only possible in network bottlenecked use cases.

This means that RDMA, as such, does not have to be the universal communication protocol, like TCP, as its performance strongly depends on the use case at hand.

The results, found in this thesis, signal that further integration of RDMA in the BFT framework Reptor is required to utilize all of its advantages at highest efficiency. The implemented memory management and flow control should be able to form a basis for future work, further optimizing the interplay of Reptor's layers to make use of RDMA's feature set.

Bibliography

[1] M. K. Aguilera, N. Ben-David, R. Guerraoui, V. Marathe, and I. Zablotchi. The impact of rdma on agreement. May 2019.

[2] AZUL Systems. *Understaning Java Garbage Collection*, 2014. URL `https://www.azul.com/files/Understanding_Java_Garbage_Collection_v4.pdf`.

[3] J. Behl, T. Distler, and R. Kapitza. Consensus-oriented parallelization: How to earn your first million. In *Proceedings of the 16th Annual Middleware Conference*, pages 173–184. ACM, 2015.

[4] J. Behl, T. Distler, and R. Kapitza. Hybrids on steroids: Sgx-based high performance bft. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 222–237, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4938-3. doi: 10.1145/3064176.3064213. URL `http://doi.acm.org/10.1145/3064176.3064213`.

[5] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, Nov. 2002. ISSN 0734-2071. doi: 10.1145/571637.571640. URL `http://doi.acm.org/10.1145/571637.571640`.

[6] DiSNI. Direct storage and networking interface, 2019. URL `https://github.com/zrlio/disni`.

[7] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, 2014. USENIX Association. ISBN 978-1-931971-09-6. URL `https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi{\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{c\global\mathchardef\accent@spacefactor\spacefactor}\accent19c\egroup\spacefactor\accent@spacefactor}`.

[8] P. W. Frey and G. Alonso. Minimizing the hidden cost of rdma. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 553–560, June 2009. doi: 10.1109/ICDCS.2009.32.

[9] E. Gafni and L. Lamport. Disk paxos. In *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, pages 330–344, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-41143-7. URL `http://dl.acm.org/citation.cfm?id=645957.675967`.

[10] M. B. (https://stackoverflow.com/users/16883/michael borgwardt). Extending bytebuffer class. Stack Overflow. URL https://stackoverflow.com/a/624508. version: 2009-03-08.

[11] P. Jalili, M. Primi, and F. Pedone. High performance state-machine replication. pages 454–465, 06 2011. doi: 10.1109/DSN.2011.5958258.

[12] JUnit. junit4, June 2019. URL https://github.com/junit-team/junit4.

[13] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, 2016. USENIX Association. ISBN 978-1-931971-30-0. URL https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia.

[14] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. *SIGOPS Oper. Syst. Rev.*, 41(6):45–58, Oct. 2007. ISSN 0163-5980. doi: 10.1145/1323293.1294267. URL http://doi.acm.org/10.1145/1323293.1294267.

[15] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[16] Mellanox Technologies. *Security in Mellanox Technologies InfiniBand Fabrics Rev 1.0*, 2012. URL https://www.mellanox.com/related-docs/whitepapers/WP_Secuirty_In_InfiniBand_Fabrics_Final.pdf.

[17] Mellanox Technologies. *ConnectX®-3 Pro VPI Single and Dual QSFP+ Port Adapter Card User Manual Rev 1.5*, December 2015.

[18] Mellanox Technologies. *RDMA Aware Networks Programming User Manual Rev 1.7*, May 2015. URL https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[19] Mellanox Technologies. *RoCE vs. iWARP Competitive Analysis*, February 2017.

[20] I. Messadi. Low latency byzantine agreement using rdma. Master's thesis, November 2017.

[21] OpenFabrics. Ofed for linux. https://www.openfabrics.org/ofed-for-linux/, 2019.

[22] Oracle. *Selector (Java Platform SE 8)*, 2019. URL https://docs.oracle.com/javase/8/docs/api/java/nio/channels/Selector.html.

[23] M. Poke and T. Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 107–118. ACM, 2015.

[24] J. Postel. *Transmission Control Protocol*, September 1981. URL https://tools.ietf.org/rfc/rfc793.txt.

[25] RDMAmojo. ibv_post_send(), 2019. URL https://www.rdmamojo.com/2013/01/26/ibv_post_send/.

[26] S. Rüsch, I. Messadi, and R. Kapitza. Towards low-latency byzantine agreement protocols using rdma. In *Proceedings of the 1st Workshop on Byzantine Consensus and Resilient Blockchains*, BCRB'18, Luxemburg, jun 2018. URL https://www.ibr.cs.tu-bs.de/users/ruesch/papers/ruesch-bcrb18.pdf.

[27] P. Stuedi, B. Metzler, and A. Trivedi. jverbs: Ultra-low latency for data center applications. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 10:1–10:14, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2428-1. doi: 10.1145/2523616.2523631. URL http://doi.acm.org/10.1145/2523616.2523631.

[28] S. Tsai and Y. Zhang. A double-edged sword: Security threats and opportunities in one-sided network communication. *CoRR*, abs/1903.09355, 2019. URL http://arxiv.org/abs/1903.09355.

[29] D. Weiss and S. Osiski. JUnitBenchmarks, August 2013. URL https://mvnrepository.com/artifact/com.carrotsearch/junit-benchmarks/0.7.2.

A Contents of the CD

The CD included with the thesis is structured into the following folders:

- **`ba-rdma-reptor`**

  This folder contains the documents related to the thesis document itself as well as configuration files, log files and benchmark results. It also contains a digital copy of this thesis. The ba-rdma-reptordirectory also contains documented scripts used to start, evaluate and visualize the benchmarks shown in this thesis.

- **`disni-hybster`** (git bundle)

  The bundle disni-hybster is a git repository. There are multiple branches available showing different implementation efforts and philosophies. The two branches developed during this thesis are `wrapper_patch` and `direct_patch`. The direct patch is what was used for further development throughout this thesis on the `rdma-hybster` repository. The philosophy was to port changes from the pre-existing RUBIN implementation directly on the DiSNI version 2.0 source.

  Before this was undertaken we aimed to instead change RUBIN into a wrapper around the DiSNI library, starting from version 2.0 to make upgrading DiSNI version easier in the future. Due to time constraints, this was stopped in favour of the direct patch attempt.

- **`rdma-hybster`** (git bundle)

  Contains the implementation of RUBIN in Reptor extended throughout this thesis. The branches map to different stages of development:

  - `master`: This branch represents the baseline from the previous implementation of RUBIN in Reptor.
  - `markus-becker-2019-ba`: This branch contains the progress of implementing the DiSNI wrapper in Reptor.
  - `markus-becker-2019-ba-direct`: This branch is the most stable version of the implementation presented in this thesis.
  - `markus-becker-2019-ba-direct-dev`: Additional performance tweaks and more finely granulated commits to document the implementation process and optimize performance.

B JUnit benchmark

```
package reptor.distrbt.common.data;


import com.carrotsearch.junitbenchmarks.AbstractBenchmark;
import com.carrotsearch.junitbenchmarks.BenchmarkOptions;
import org.junit.Before;
import org.junit.Test;

import java.nio.ByteBuffer;

@BenchmarkOptions(benchmarkRounds=20)
public class MemoryManagementBenchmarks extends AbstractBenchmark {

    public static final int FULL_REPETITIONS = 100;
    public static final int DEFAULT_LENGTH = 1_000;
    public static final int DEFAULT_CAPACITY = 1024 * 1024; // 1 MB

    protected ManagedBufferRing allocRing;

    @Before
    public void init() {
        allocRing = new ManagedBufferRing(DEFAULT_LENGTH, DEFAULT_CAPACITY);
    }

    @Test
    public void fullAllocationRing() {
        for (int j = 0; j < DEFAULT_LENGTH; j++) {
            allocRing.next();
        }
    }

    @Test
    public void fullAllocationNative() {
        for (int j = 0; j < DEFAULT_LENGTH; j++) {
            ByteBuffer.allocateDirect(DEFAULT_CAPACITY);
```

```
        }
    }

    @Test
    public void manyFullAllocationsRingVariable() {
        ManagedBuffer current;
        for (int i = 0; i < DEFAULT_LENGTH * FULL_REPETITIONS; i++) {
            current = allocRing.next();
            current.asByteBuffer().put((byte)1);
            current.free();
        }
    }

    @Test
    public void manyFullAllocationsRingAnonymous() {
        for (int i = 0; i < DEFAULT_LENGTH * FULL_REPETITIONS; i++) {
            allocRing.next().free();
        }
    }

    @Test
    public void manyFullAllocationsNativeAnonymous() {
        for (int i = 0; i < DEFAULT_LENGTH * FULL_REPETITIONS; i++) {
            ByteBuffer.allocateDirect(DEFAULT_CAPACITY);
        }
    }

    @Test
    public void manyFullAllocationsNativeVariable() {
        ByteBuffer current;
        for (int i = 0; i < DEFAULT_LENGTH * FULL_REPETITIONS; i++) {
            current = ByteBuffer.allocateDirect(DEFAULT_CAPACITY);
            current.put((byte)1);
        }
    }
}
```