Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

MASTER'S THESIS IN INFORMATICS

Optimizing Network Latency for Transactions

Philipp Fent

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

Master's Thesis in Informatics

Optimizing Network Latency for Transactions

Optimierung der Netzwerklatenz für die Transaktionsverarbeitung

Author:	Philipp Fent
Supervisor:	Prof. Dr. Thomas Neumann
Advisors:	Dr. Viktor Leis Alexander van Renen, M. Sc.
Submission date:	May 15, 2018

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Garching,	
Location, Date	Signature

Abstract

Transaction processing over the network is usually only limited by latency, since typical workloads do not saturate the network bandwidth, but need to transmit many small messages as fast as possible. Since conventional systems are not sufficiently optimized for low latency operations, in this thesis we designed a messaging library, which effectively transmits transactions over network interfaces with latencies as low as possible.

As a key component, we use user-space remote direct memory access (RDMA) over Infiniband, because its latencies are roughly ten times lower than kernel based networking. Instead of using RDMA send and receive primitives, we implemented an optimized messaging library, which directly writes into remote message buffers. This approach reduces overhead and allows to efficiently check for new incoming messages with optimized polling routines.

Using our library, we achieve round trip times as low as 2.5 µs, which is over 30 % lower than using send and receive and a 13 fold increase in sequential transaction throughput compared to standard TCP. In comparison to competing state of the art RDMA transaction processing systems, our library achieves a 0.5 µs or 18 % lower round trip time on similar hardware, while still providing reliable message delivery. Based on our work, new, high performance distributed transaction processing systems can be implemented.

ZUSAMMENFASSUNG

Das größte Problem für netzwerkbasierte Transaktionsverarbeitung sind Netzwerklatenzen. Die Ursache dafür ist, das bei derartigen Anwendungsfällen nicht die gesamte Netzwerk Bandbreite ausgenutzt werden kann, sondern das Ziel ist, viele kleine Nachrichten möglich schnell zu verschicken. Nachdem herkömmliche Systeme nicht ausreichend auf geringe Latenzen optimiert sind, haben wir im Rahmen dieser Arbeit eine Programmbibliothek entworfen, die es erlaubt Transaktionen mit kürzest möglichen Umlaufzeiten zu übermitteln.

Als zentrale Komponente verwenden wir User-Mode RDMA über ein Infiniband-Netzwerk, da die damit einhergehenden Latenzen ca. zehnmal geringer sind, als Netzwerkkommunikation über den Kernel. Auf Basis dessen haben wir eine speziell optimierte nachrichtenbasierte Kommunikation implementiert, die direkt in die Netzwerkpuffer des Kommunikationspartner schreibt, statt RDMA Send- und Receive-Primitiven zu verwenden. Dieser Ansatz hat deutlich weniger Overhead und erlaubt es, eingehende Nachrichten mit speziell optimierten Routinen zu detektieren.

Mit unserer Implementierung erreichen wir Paketumlaufzeiten von 2.5 µs, was ca. 30 % schneller ist als die Verwendung von Standard RDMA-Nachrichten und verglichen mit TCP einem ca. 13 mal so hohen sequentiellen Durchsatz entspricht. Im Vergleich zu ähnlichen Transaktionsverarbeitungssystemen, die RDMA verwenden, hat unser System ca. 0.5 µs geringere Paketumlaufzeiten auf vergleichbarer Hardware, ohne die Zuverlässigkeit der Nachrichtenübermittlung zu beeinträchtigen. Aufbauend auf unserer Arbeit können hochperformante, verteilte Transaktionsverarbeitungssysteme implementiert werden.

CONTENTS

List of Figures

List of Tables

Listings

CHAPTER 1

INTRODUCTION

By eliminating traditional bottlenecks piece by piece, modern in-memory databases are getting fast [11]: In-memory data structures can easily process multiple million reads and writes per second [12], which is mainly caused by the ability to fit large datasets completely into main memory, where traditional database systems need to store their data on disk. Random accesses on hard drives can quickly take several milliseconds, which caused physical storage to be the major bottleneck. Now, storage is getting increasingly faster, with solid state drives (SSDs) and non-volatile memory (NVM) becoming available throughout the industry.

The many eliminated bottlenecks on local databases shift the responsibility towards interconnecting multiple systems. Databases are often run in a networked cluster, either to guarantee high availability through redundancy or because of sheer data size, which does not fit a single machine. Compared to single node systems, operations over the network are slow, because faster networks were neglected, while other bottlenecks where a priority. Raw bandwidth of networks can usually be scaled reasonably well, e.g. using link aggregation or more expensive network equipment, but high-speed TCP operations are usually CPU bound, as receivers become computationally overloaded [19].

For distributed online transaction processing (OLTP) systems, high network latencies thrash performance, since subsequent operations are often dependent on previous ones. Although network bandwidth is limited, raw serialization times of messages only make up a minor fraction of the total latency, because the size of each transaction is usually small. Most of the latency is actually processing overhead, which caused many researchers to focus on avoiding network traffic altogether. Networks supporting RDMA reduce this overhead instead, which significantly reduces overall latency and allows systems to scale better.

In this work, we will take a look on how to optimize latencies for networked transactions. We will also answer the question, how an implementation for best low-latency transaction processing over RDMA networks should be designed.

METHODOLOGY

To answer these questions, we designed a high-performance system with the available tools for RDMA communication. Since software supporting RDMA is already available, we first analyzed the current implementations and reasoned about their effectiveness. Additionally, we constantly measured the different approaches and categorized their usefulness. From the best approaches we then created our own implementation, which fits transaction processing best. Afterwards we evaluated it using transactional workloads and suggest more applications of similar techniques.

CHAPTER 2

NETWORKING AND REMOTE DIRECT MEMORY ACCESS (RDMA)

When running databases on multiple machines, we need a way to coordinate the individual nodes. This is fundamentally different to running multiple processes on a single machine, since distributed systems cannot directly access other system's memory. Instead, distributed systems use network interface cards (NICs) to send messages via a network.

The traditional way to communicate between systems is to use the socket API, which defines a set of network agnostic operations to send data to other processes or machines. The socket API defines several types of sockets, which implement different semantics: Unreliable, unordered, and message based data transmission using `SOCK_DGRAM` and reliable, ordered, and transmitting a stream of data using `SOCK_STREAM`. Reliable connections are the go-to solution for databases, because they typically are built to store data consistently. Out of band detection for lost or corrupted unreliable messages adds complexity, which increases overhead and latency.

Stream oriented transports are suitable for transmitting big messages, but unsuited for low latency messages. To reduce message header overhead, the network layer batches small chunks data to be transmitted together. This behaviour actually adds unnecessary latency for transaction processing. The `SOCK_SEQPACKET` socket type with reliable message semantics would be the best fit in the socket API, but is not widely available. Similar semantics can be implemented with stream sockets, using features of the underlying transmission control protocol (TCP), but this turns sockets into a “leaky abstraction”.

The network layer, on which sockets operate, is usually shared between all processes on a system, thus every connection must go through the kernel to coordinate access to the hardware. As a result, in high performance workloads more time is spent on context switches, than on doing actual work, which causes the network performance to be slow. Sockets also have a long history of supporting multiple different mechanisms to solve similar problems, which came up long after the API was created. One of the most intriguing examples is the difference between `select`, `poll` and `epoll` with options set via `fcntl` and `setsockopt` to achieve non-blocking I/O and handle multiple connections. Since almost all software actually uses different combinations of these calls, supporting the full socket API is not a viable solution when only concentrating on processing transactional workloads.

Systems that emulate the socket API using RDMA are available[1], but only support a subset of all operations and are brittle when used in complex applications. Previous experiments[2] quickly uncovered several limitations regarding the API and had more problems overcoming them, then with actually creating a performant implementation. In our work, we decided to abandon the socket interface and implement our own messaging interface. With this decision, we can focus on making use of proper message semantics and provide a fitting abstraction for RDMA based messages.

2.1 RDMA WITH INFINIBAND

One of the main aspects of RDMA is to eliminate the system call overhead for network communication, similar to how shared memory eliminates it for domain socket inter-process communication (IPC). RDMA requires special NICs, connected to the PCIe bus, having direct memory access (DMA) to the host system. Using those cards, special commands can be sent to a remote device, which effectively allows DMA over the network.

[1] E.g. librdmacm's rsocket `https://github.com/linux-rdma/rdma-core`

[2] `https://github.com/pfent/rdma_tests`

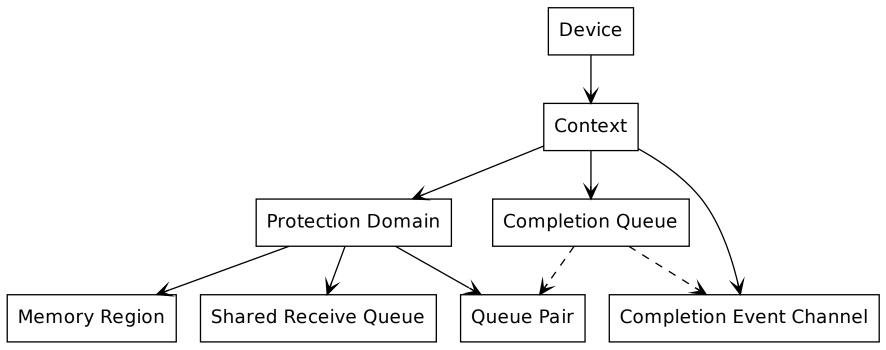

FIGURE 2.1: Libibverbs resource hierarchy: The figure shows in which order resources need to be initialized before messages can be sent.

2.2 LIBIBVERBS

The interface for RDMA are so called verbs, as defined in the RDMA Protocol Verbs Specification [7]. The userspace library implementing those verbs on Linux is called libibverbs. This library is the most low-level library for RDMA, allowing access to the raw performance of RDMA.

In the setup of an RDMA connection, there are several resource interdependencies, which are displayed in Figure 2.1. The dependencies force programs to call a sequence of verbs to allocate and create those resources. First, a device needs to be opened with a call to `ibv_open_device()`, which provides a usable handle to perform actions on the device. Next, this context can be used to create a protection domain using `ibv_alloc_pd()`, which registers memory regions using `ibv_reg_mr()`.

Memory regions specify the access permissions for RDMA operations, similar to `mprotect()` and lock virtual and physical memory, so relocation cannot happen while the NIC asynchronously accesses those regions. Furthermore, the base resources are also used to create several queues: A queue pair using `ibv_create_qp()`, consisting of a send and a receive queue and corresponding send and receive completion queues (`ibv_create_cq()`). The normal queues can be used to trigger network operations using work requests, whereas the completion queues signal the asynchronous process of work requests with work completions. The completion queues can either be polled for events or one can set up an event channel, which supports asynchronous callbacks for new messages.

As outlined in Table 2.1, RDMA connections usually require separate completion queues, which might be problematic, if one wants to check multiple connections for received messages. To natively support this, multiple receive queues can also be replaced by

TABLE 2.1: Libibverbs resource usage: Some resources only need to be acquired once for each device, others for each connection

Per device	A context Protection domain(s) Memory region(s)
Per connection	Setup information, e.g. via a traditional socket A queue pair, consisting of: • exclusive send queue • (shared) receive queue • (shared) send completion queue • (shared) receive completion queue

a shared receive queue (`ibv_create_srq()`), which can be used as a single point to receive messages.

Afterwards, the queue pairs can be connected to a remote queue pair, by issuing a series of calls to `ibv_modify_qp()`, which transition the queue from reset (`RESET`) to initializing (`INIT`), then to ready to receive (`RTR`) and finally to ready to send (`RTS`). In this process, various parameters for the queue pair need to be configured, but usually it is acceptable to use the maximum the hardware allows, which can be queried using `ibv_query_device()`.

2.2.1 WORK REQUESTS

The actual communication over RDMA happens via so called work requests. Those requests are messages to the NIC, posted to a queue in a queue pair and used to trigger asynchronous processing of the requested work.

A basic, simplified structure of a work request is shown in the listing below. Only fields which are relevant for memory access operations are shown. Actual work requests are more involved, since there are manifold types of work requests.

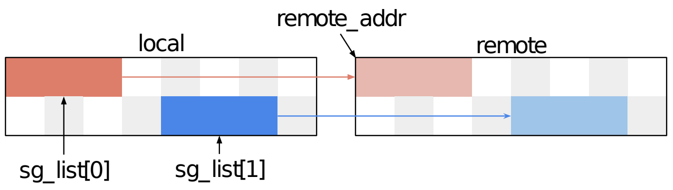

FIGURE 2.2: Scatter / gather operations: Multiple entries in `sg_list` allow to selectively modify memory, while leaving intermediate values unchanged.

LISTING 2.1: Simplified work request structure

```
 1 struct ibv_send_wr {
 2     uint64_t             wr_id;        // Identifier
 3     struct ibv_send_wr  *next;         // List of work requests posted together
 4     struct ibv_sge      *sg_list;      // Array of fragments of local memory
 5     int                  num_sge;      // Size of that array
 6     enum ibv_wr_opcode   opcode;       // Type of the work request
 7     unsigned int         send_flags;   // Flags, altering behaviour
 8     uint32_t             imm_data;     // Only valid for *_WITH_IMM work requests
 9     struct { // Actually a union, but for clearness, only the active part is shown
10         uint64_t         remote_addr; // Target remote address
11         uint32_t         rkey;         // Identifier of the remote memory region
12     } rdma;
13     // Additional members omitted
14 };
```

The work request structure consists of multiple fields: The `wr_id` of the work request identifies it locally, i.e. a work completion also contains this id. Work requests can also be chained to a linked list via the `next` pointer with a terminating `nullptr`. Multiple different fragments of memory can also be combined in a single work request as scatter / gather entries in the `sg_list`. The name is a bit confusing, since this member is not actually a list, but a pointer to an array with `num_sge` entries. The first entry in this list specifies the base local address, which corresponds to the specified remote memory. Subsequent entries then operate with an offset on the remote, which is equal to the offset to the first entry (cf. Figure 2.2). This effectively allows operations with holes. `opcode` specifies the type of work request, which the structure contains. `send_flags` specify optional flags, which influence the local processing of the work request.

The actual data to be sent is contained in `imm_data` (for types `*_WITH_IMM`) and the following `rdma` data structure. Different types of work requests have different active fields in a union (line 9), but for the sake of brevity we only look at one instance. In our case, `remote_addr` identifies the memory location of the remote system, this work item operates on, and `rkey` identifies the RDMA memory region this address belongs to. This mapping is needed, because virtual memory addresses are not necessarily unique and multiple processes could have the same address registered with libibverbs. The length of data manipulated in the remote memory is implicitly defined as the sum of individual lengths in the `sg_list`.

The `opcode` field specifies, which type of work request the structure contains. The types can be classified into two basic classes of work requests corresponding to the initiating system: outgoing and incoming. Outgoing work requests are rather diverse with read, write, send, and atomic work requests, while the only incoming operations are receive requests. Please note, that send work requests have slightly confusing names: The `ibv_send_wr` structure can contain all possible types of outgoing work requests, while a send work requests is one specific type, consisting of a `ibv_send_wr` with `opcode` send.

Read requests are a special case, since the flow of data is actually reversed. Read work requests specify a remote address to read from and a local address, where the read data is then put into. When the read has been finished, the NIC generates a work completion in the send completion queue and the initiator can use the data. The following listing shows a simplified code fragment to read data from remote memory via an RDMA read.

LISTING 2.2: Usage example of an RDMA read

```
ibv_sge local_addr = {
    .addr = 0x560268049000,
    .length = 8
}
ibv_send_wr workRequest = {
    .opcode = IBV_WR_RDMA_READ,
    .send_flags = IBV_SEND_SIGNALED,
    .sg_list = &local_addr,
    .num_sge = 1,
    .rdma.remote_addr = 0x7ffce0847000
};
ibv_post_send(queuePair, &workRequest, &error);

while(ibv_poll_cq(&completionQueue, 1, &workCompletion) == 0)
    wait();
// *local_addr now contains the data from remote system's *remote_addr
```

Write work requests send data to the remote and instruct the remote NIC to write to a specific memory address. The requests therefore need to specify local data source and remote destination address. To avoid race conditions, the local memory should not be modified, while the write request has not been completed. It can be reused immediately, when the "inline" flag has been set, which sends the data synchronously. However, this operation is only supported up to an upper limit, typically 512 Bytes. Otherwise, the same polling for work completions mechanism as with read requests needs to be used, before the sending memory can be reused.

TABLE 2.2: Supported work requests per connection [15]

	RC	UC	UD
Send / Recv	✓	✓	✓
Write	✓	✓	
Read	✓		
Atomic	✓		

Send work requests also send data to the remote, but do not specify the destination and therefore need to be handled on the receiving side. They only operate with corresponding receive requests and are described together with them later.

Atomic operations are the most advanced work requests, which support compare-and-swap and fetch-and-add operations. Those operations behave similar to the instructions executed on the CPU, thus are effectively a read plus a write. However, they both require memory fences on the remote end, with a more involved communication between the remote NIC and main memory subsystem. Also, both operations only operate on exactly one 64 bit unsigned integer. Those atomic operations are useful for specific applications, such as network-global counters, but for most use-cases they require multiple round trips and therefore are hardly suitable for low latency. Usually, they also decrease network throughput, since memory fences are rather expensive in comparison to normal read and write operations.

Receive requests are the only incoming work requests They do not cause any actions on their own, but act as tokens to be consumed by incoming messages. In the receive requests, the corresponding memory address for send work requests are specified, allowing them to be processed.

2.2.2 CONNECTION TYPE

While work requests are used to implement communication, they can only be transmitted between queue pairs, which can be connected differently. The choice of the actual connection type, unreliable datagram (UD), unreliable connected (UC), or reliable connected (RC) might affect the overall performance of the implementation. Another difference is, that not all connections support all work requests. Supported requests for each connection type are shown in Table 2.2.

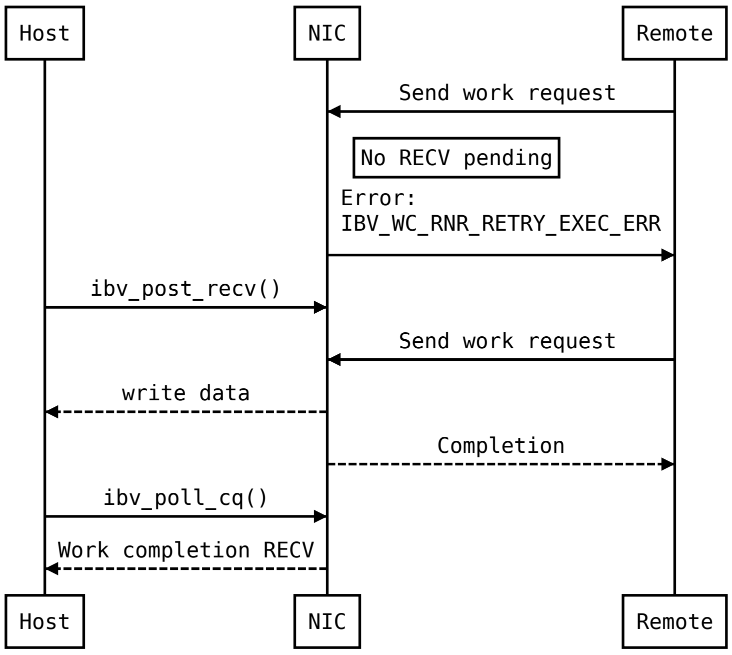

FIGURE 2.3: Send and receive requests: An incoming send message needs to be matched with a receive request, or the transmission will fail.

2.2.3 USAGE

Using work requests has some important pitfalls as displayed in Figure 2.3: Receive requests actually need to be in the receive queue, before any send can be accepted. Since the NIC does not buffer any data, each incoming send needs to be matched with a corresponding receive, otherwise an error message is generated, usually "retry counter exceeded".

The receive requests also need to be prepared to store arbitrarily large data. When an incoming send is bigger than the corresponding receive request, a "local length error" is generated. A common way to work around this behaviour is to only send fixed size messages, or limit messages to a maximum size and include the actual message size in the immediate data field.

When a receive request was consumed, a work completion is generated and placed in the associated receive completion queue. The receive request behaviour strongly depends on the incoming message:

- Send work requests only copy the incoming data to the specified memory address.

- For send work requests with immediate data, the generated work completion additionally contains a valid immediate data field.
- Write work requests with immediate data also consume a receive request, but do not use the specified memory address. Instead, they only generate a work completion with an immediate value, which is the only way to receive the immediate data.

To summarize, setting up memory regions and protection domains is rather expensive and should only be done once. In contrast, work requests are rather cheap and can be recreated for each message, but should ideally also be reused when possible. Communication with multiple peers need a separate queue pairs and completion queues for each connection. In this case, shared receive queues and completion event channels are a nice way to process messages with reduced CPU overhead. Their suitability for low latency application is discussed in Section 3.2. For message types, send and receive work requests are a nice fit for message oriented communication, but are quite hard to use reliably. Read and write messages are much easier to use correctly, but require additional work to operate with message semantics.

CHAPTER 3

USING RDMA EFFICIENTLY

RDMA and its performance implications have already been studied in various papers [8, 13, 15]. However, there still exists non intuitive hardware behaviour and hidden latency pitfalls. To uncover those, we took a detailed look at the the libibverbs library and measured the latency behaviour of various primitives. The goal of this chapter is to understand the RDMA subsystem and how software and hardware play together.

3.1 LIBIBVERBS WITH C++

Libibverbs is a traditional C-style library, while most modern high performance applications to some degree use C++ and the accompanying paradigms. The C-style API already caused headaches and several hours of debugging work, because of the completely manual resource management with interdependent resources (see Section 2.2).

Within the scope of this thesis, we mitigated some of the flaws of libibverbs by designing and implementing better abstractions for C++ programs. We built libibverbscpp as an attempt to improve the usability by providing common C++ abstraction on top of libibverbs, with no computational overhead. The library has been published separately[1].

Libibverbs' design has some major downsides, regarding its object-orientated design. Although it is quite well designed within the capabilities of C, it still lacks some basic

[1] `https://github.com/pfent/libibverbscpp`

design principles, like information hiding, encapsulation with separation of concerns, and type safe polymorphism.

3.1.1 Separation of Concerns

The lack of information hiding is mainly caused by the absence of a concept of private members in C structs. This results in situations, where the user of the library sees every implementation detail, which makes it difficult to keep writable and read-only members apart.

Libibverbscpp tries to improve this situation by providing zero overhead abstractions over the C-style API. Improvements on the object-oriented design were straight-forward, since information hiding can easily be done via private members and appropriate setters and getters. E.g. libibverbs exposes immanent device attributes as writable, although there is no meaningful way to modify them. By only providing getters for the C++ version, this immutability can instead be correctly represented by the available operations.

3.1.2 Polymorphism

Furthermore, libibverbs makes heavy use of union polymorphism, e.g. depending on the operation code of a work request or the type of an event, a different set of union members are valid. This can easily lead to programming errors with corrupted data and undefined behaviour, since no errors or warnings are emitted when accessing a currently invalid part of the union. This makes libibverbs rather confusing, since the behaviour of various of the approximately 70 global functions changes, depending on which union members are valid. Which functions are callable in the current context is not immediately clear.

Similar to visibility restrictions, polymorphism is a common feature of C++ and was quickly implemented by creating class hierarchies. E.g. the hierarchy displayed in Figure 3.1 was used for work requests. With this approach, each class can only selectively offer the methods, which are valid for this specific work request, without the cognitive overhead of checking the operation code each time. Since this implementation is static polymorphism, without virtual functions, the performance of this approach is still exactly the same as directly using libibverbs.

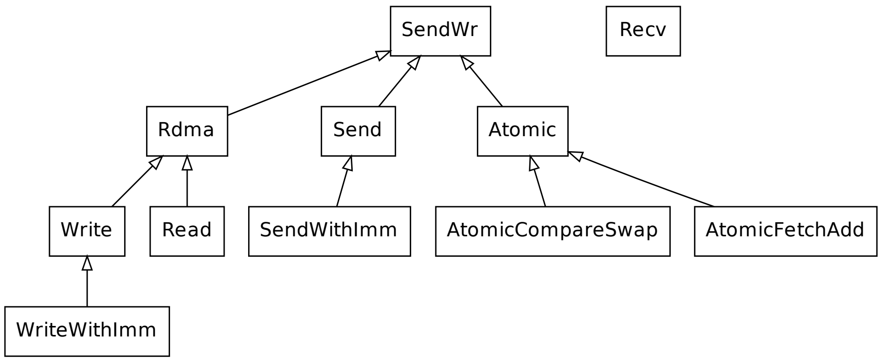

Figure 3.1: Libibverbscpp work request specialization hierarchy: While all descendants of SendWr are based on the same C struct, dedicated subclasses reduce complexity by limiting callable operations.

Table 3.1: Manually managed resources in libibverbs

Acquire	Release
`ibv_alloc_mw()`	`ibv_dealloc_mw()`
`ibv_alloc_pd()`	`ibv_dealloc_pd()`
`ibv_create_ah()`	`ibv_destroy_ah()`
`ibv_create_comp_channel()`	`ibv_destroy_comp_channel()`
`ibv_create_cq()`	`ibv_destroy_cq()`
`ibv_create_flow()`	`ibv_destroy_flow()`
`ibv_create_qp()`	`ibv_destroy_qp()`
`ibv_create_srq()`	`ibv_destroy_srq()`
`ibv_open_device()`	`ibv_close_device()`
`ibv_open_xrcd()`	`ibv_close_xrcd()`
`ibv_reg_mr()`	`ibv_dereg_mr()`

3.1.3 Resource Management

Additionally, in libibverbs all resources are managed completely manual. Many resources provided by libibverbs need to be cleaned up manually, as outlined in Table 3.1. Since multiple exit paths for a function are common, resource cleanup is usually handled via error prone `goto fail` constructs. In complex applications with long-living resources, this is not a robust approach to cleanup, especially when using C++, which prominently features the RAII idiom.

Libibverbs also requires, that resources are released in the inverse order in which they were acquired. Since this is the natural destruction order of C++ objects, this allows for natural resource management with C++ unique pointers. The standard approach would be to implement a wrapper with a custom destructor around the libibverbs resources, however those cannot be constructed in standard C++, because their memory is completely managed by libibverbs itself. A more appropriate way to implement this, is a subclass with a custom `operator delete()`, which delegates object cleanup back to libibverbs. This allows for all resources to be managed in idiomatic C++ RAII style, as shown in the listing below.

LISTING 3.1: Libibverbs compared to libibverbscpp

```
{ // Pure libibverbs
ibv_context * ctx = ibv_open_device(device);
ibv_pd * pd = ibv_alloc_pd(ctx);
byte buffer[64];
ibv_mr * mr = ibv_reg_mr(pd, &buffer, 64, 0);
// don't forget to explicitly call releasing methods!
}
{ // Using our libibverbscpp implementation
unique_ptr<Context> ctx = device.open();
unique_ptr<ProtectionDomain> pd = ctx->allocProtectionDomain();
byte buffer[64];
unique_ptr<MemoryRegion> mr = pd->registerMemoryRegion(buffer, 64, {});
} // cleanup happens automatically
```

3.1.4 C++ MANAGEMENT OF C CONSTRUCTS

Because of libibverbs explicit memory management with custom acquisition and releasing functions, only pointers to such objects can exist. Value classes, which call the appropriate functions in the destructor would be a possibility, but they would require special care to allow special member functions to work (cf. “Rule of Three” [16]) and would effectively just duplicate the functionality of a `unique_ptr`.

Instead, we created pointer only C++ subclasses of the libibverbs C structs. This way, those resources can seamlessly be used with standard C++ utilities, but also allow access to the underlying raw structs. I.e. new or unsupported features can still be used with our classes by casting the objects to their base classes when needed.

LISTING 3.2: Support for raw verbs

```
unique_ptr<QueuePair> qp;
Write workrequest;
SendWr *bad;
// [...]
// This might throw an exception
qp->postSend(workrequest, bad);
// Legacy calls still work
int status = ibv_post_send(qp.get(), &workrequest, (ibv_send_wr **)(&bad));
// status != 0 indicates an error
```

Resource acquisition in libibverbscpp also models the resource hierarchy (Figure 2.1) much closer, by putting acquisition in the responsibility of parent objects. E.g. a protection domain (the owner) has a member function to register a memory region associated with that domain. This also makes it easier to discover which methods can be called

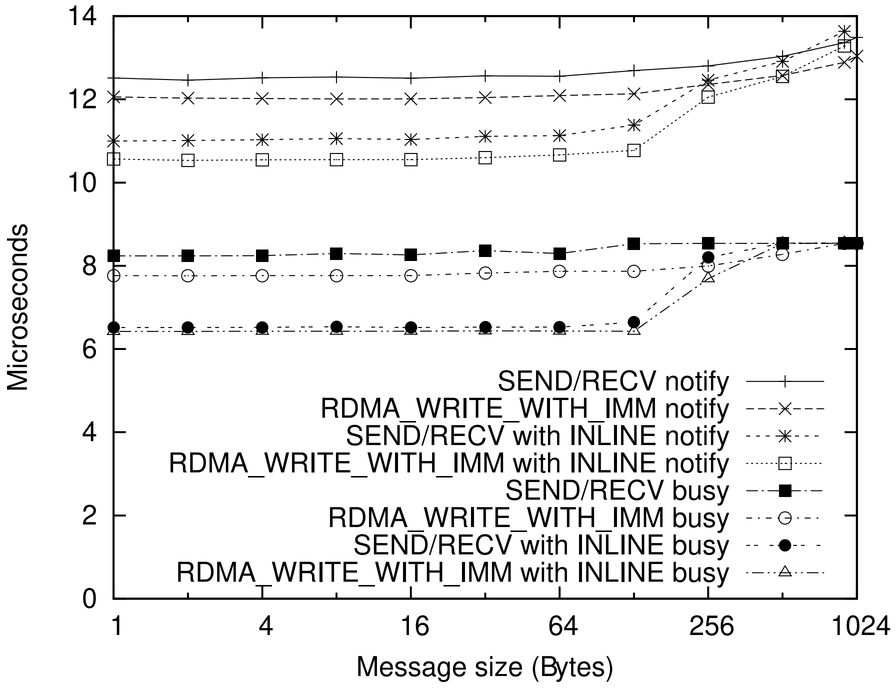

FIGURE 3.2: Latencies of RDMA verbs according to MacArthur and Russell [13]

with the object at hand, since each object only has a handful of possible methods to call on, opposed to searching through all verbs, which might take the object as a parameter. As a result, all classes have deleted constructors, since all resource allocation happens in the member methods of the object above in the resource hierarchy.

Most of the time, when RDMA verbs return an error, those are fatal, such as calling functions in the wrong order. As described in Listing 3.2, libibverbscpp throws exceptions in such cases, which reduces the complexity of calling functions by defaulting to propagate fatal errors to the topmost function. This eliminates the potential to swallow errors in case return values are not explicitly checked and require explicit handling for silent errors.

3.2 LATENCY

After creating a comfortable API to work with, we could analyze the latency characteristics of RDMA by measuring the performance of different Infiniband verbs. Since we have a fairly low level API at hand, there are many variables to be adjusted and different primitives to be used, which all might have implications for low latency operation. Traditional RDMA optimizations focus more on parallel throughput, than on low latency.

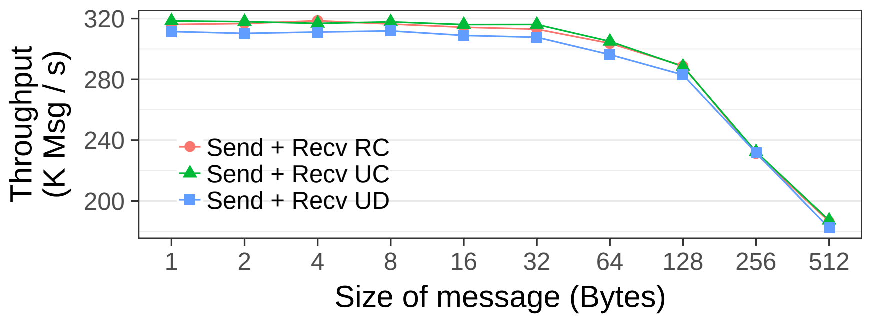

FIGURE 3.3: Serial throughput of identical messages over all three connection types

Those optimizations might also lead to low latency operation, however which are truly beneficial is unclear.

MacArthur and Russell [13] showed with Figure 3.2, that “For [. . .] small messages [. . .], much lower one-way time [. . .] [can be] achieved by using busy polling rather than event notification”. This implies, that busy polling can achieve almost twice the performance (half the latency) of event notification. Absolute latencies displayed in this graph are not applicable to our uses, since the hardware used is quite dated, but relative performance between event notification and polling should be comparable. Therefore, we discarded using event notification with completion event channels, since the goal of this thesis is to explore the lowest possible latencies using RDMA, even when trading CPU usage for it.

Furthermore, those results show, that the usage of inline reduces the latency even more. Additionally, “performance is much more sensitive to the choice of RDMA options when using small messages than when using large messages” [13]. For those, the actual transmission of the data takes up the majority of the time and independent of used operations. Inline messages also trade CPU resources for lower latency, since the CPU copies data to the NIC, instead of the NIC reading the message asynchronously.

3.2.1 CONNECTION TYPE

To compare different connection types, we considered an implementation using send and receive work requests for all three available connection types. E.g. Kalia, Kaminsky, and Andersen [10] used send work requests with 0 data size, using only the immediate value to transmit messages of 4 Byte, which is often enough information to call stored procedures. Since those messages are the only ones usable on all connection types (cf. Table 2.2), measuring them allows a fair, fundamental comparison of connection types.

The concrete system used for those experiments is described in detail in Chapter 5. However, the actual system is not important here, because absolute values have little meaning for the micro benchmarks presented in this chapter. The importance of measurements in this chapter only is to compare different approaches in a similar environment, relative to each other.

When comparing sequential throughput for different connection types in Figure 3.3, UD messages are slightly slower. Although the performance differences between all three connection types are only minor. The higher latency of the UD messages is probably caused by the 40 Byte global routing header (GRH), included in UD messages, to give the receiver the possibility to identify the origin of the message. Since the messages are then slightly larger than the other messages, performance is a notch lower.

3.2.2 WORK REQUEST TYPE

Figure 3.4 shows experimental measurements on RC connections of sequential messages per second, which simply echo the received data (similar to ping). The measurements plotted here are generally symmetrical, e.g. the same operations are used for both directions. The only exception is the experiment with read work requests, which first issue a write into remote memory and read that chunk of data back. The read workload consisted of sending the work request, then either poll memory for the data to appear or poll the work completion from the completion queue.

All measurements also include subsequent verification of the received data. In contrast to the other work requests, reads operate without involvement of the remote side. On the one hand, this is an advantage, since it can reduce resource usage on a server by efficiently retrieving data. On the other hand, this limits the general applicability, because transactional workloads might still need to be interactive on the remote.

As the next experiment, send work request with corresponding receives were tested. This experiment included issuing a send, then the remote side polled its completion queue, until it completed a receive work request. Afterwards the remote side issued a send in the opposite direction and in turn, the original sender completed a receive.

Lastly, write work requests were tested in both available flavours, with and without immediate data. The write with immediate data test case polls for completion of a receive request for the immediate data, while the work requests without immediate data simply poll the associated memory location, until the data appears.

From those measurements we can observe immediately, that all experiments depending on the completion of a receive request have significantly worse performance with higher

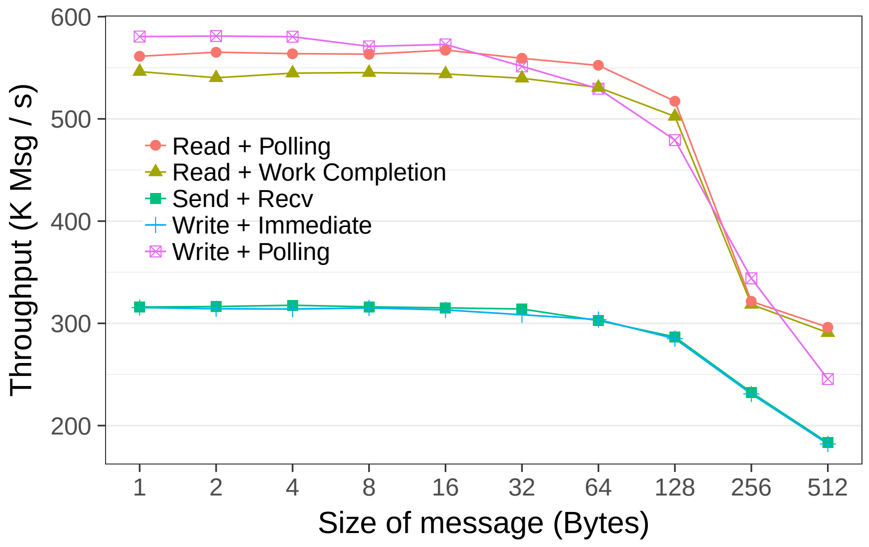

FIGURE 3.4: Serial throughput of messages via different raw work requests

latency. Both send and write with immediate work requests have significantly worse performance than the competing options.

Read work requests, which poll their work completion have slightly lower throughput than write work requests, which directly poll the memory location. Compared to polling work completions, polling the local memory, where the remote data is read into, has consistently higher throughput and is then very close to memory polling writes. For messages < 16 Byte, write work requests, combined with polling the incoming memory are around 3 % faster, while for bigger sizes read work requests win by 6 %. Overall, with respect to latency both reads and writes have no significant latency differences.

Round-trip times for of messages ≤ 16 Byte can be as low as 1.7 µs, which is roughly the performance the Infiniband products promise. I.e. Mellanox claims in the product brief of the used ConnectX-3 NIC: “1us MPI ping latency” [14]. Small latency differences might be explained by CPU or RAM differences, but this 70 % difference could be caused by a variety of reasons. Possible explanations might be, that the advertised latency might not include userspace processing but was only measured NIC to NIC, or that Mellanox used loopback connections with better latency characteristics.

3.3 EFFECTIVE INDIRECT WRITES

Direct, continuous writes can achieve excellent latencies, but are very limited. Often there is a need for an indirection to have more flexibility with write positions. This concept is similar to the concept of slotted pages [5], which allows flexible tuple storage in database systems.

The basic concept behind this mechanism is to write the position of the data to one location and then subsequently write the data to this “pointed to” position. This concept can also be applied to RDMA operation, so that the sender can choose an arbitrary address to write to and notify the receiver of the data written. Within the primitives of libibverbs, write work requests with and without immediate data seem applicable.

Write work requests with immediate appeal, since they can combine a write to memory with transmitting the position in the immediate data field. The immediate value is then noticed by the receiver via a receive work request. We initially hypothesized this would be the fastest way to implement indirect writes, because there is only one work request to be processed by the NIC. However Section 3.2 showed, that receive requests have a severe negative impact on the performance, so we compared them to other implementations.

The competing concept is to use simple **write work requests and write the position in a separate write**, effectively doubling the number of work requests to post. The core implementation can be seen in the following listing.

LISTING 3.3: Indirected write work request

```
auto dataWrite = createWrite(data);
auto positionWrite = createWrite(sendPos);

const auto destination = rand() % size;
*sendPos = destination;
dataWrite.setRemoteAddress(remoteMr.offset(destination));

qp.postWorkRequest(dataWrite);
qp.postWorkRequest(positionWrite);
```

Please note, that the order in which the work requests are posted to the queue pair is significant. Since write operations can only be used with connected queue pairs (cf. Table 2.2), the specification guarantees the ordering in which those writes become visible at the receiving side. Therefore, we can have two variations: **Writing the data first** or **writing the position first**. Data first implies, that all data is written before the position becomes visible to the CPU. This has the advantage of only needing to poll the position, without the need to wait until the data write has been completed. On the other

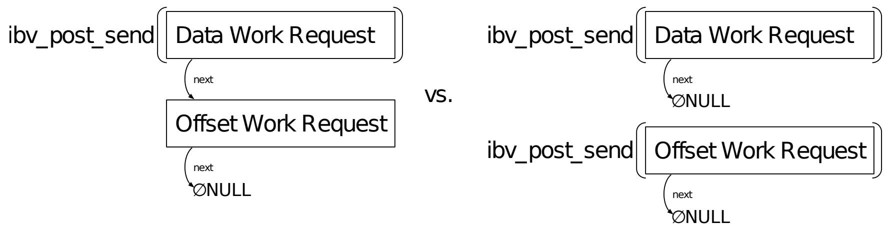

FIGURE 3.5: The chained work requests on the left can be posted with a single verb, while unchained work requests on the right require two method calls

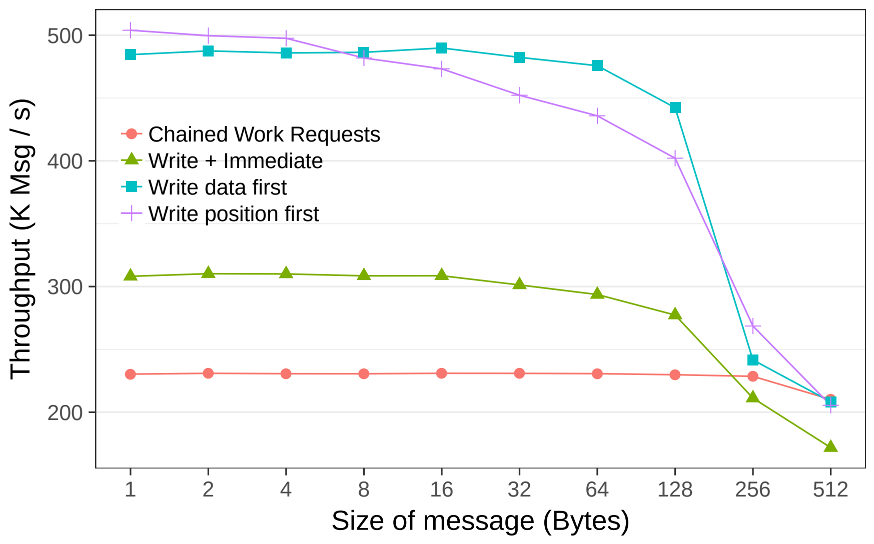

FIGURE 3.6: Latencies of different indirect write implementations

hand, this might introduce additional latency, since this effectively serializes the data access. Writing the position first allows the CPU to already anticipate the data written and access it as soon as the NIC finishes writing.

Additionally, a commonly recommended technique for multiple requests is to link them to a list via the `next` pointer [1, 22] to get **chained work requests**, that can be posted all at once. Since both work requests are always created and posted together, this can potentially decrease the work the sender needs to do. The alternative are two separate calls to `ibv_post_send()`, as displayed in Figure 3.5.

The results of an experiment, which measures round trips by echoing different sized messaged back to the sender, both times via the write indirection is shown in Figure 3.6.

In this benchmark, we can see, that using two work requests to indirectly write messages has a similar advantage, as already measured in Section 3.2.2. Using only a single continuous write would be around 10 % faster for comparable message sizes. Curiously, using chained work requests decreases the overall performance by a factor of 2. We assume chaining work requests actually totally serialize the transmission on the sending side, where posting two separate work requests enable parallel transmission with serial consistency at the receiving side.

Which work request is posted first, only has a minuscule impact on the performance. Since writing the position last allows for easier correctness reasoning and reduced complexity, this approach should probably be preferred.

3.4 GUIDELINES

From the measurements done in this chapter, we can now deduce some guidelines, which can be used to implement low latency RDMA messaging.

MacArthur and Russell [13]'s work showed, that `INLINE` with busy polling promises the lowest latencies. Additionally, receive operations should be entirely avoided, because they induce a latency overhead, which more than doubles the total latency. Instead, writing to a memory location and polling that memory for completion should be preferred. Unfortunately, this also means that the send and receive combination, which would be the natural fit for transactional workloads, is out of the question.

Furthermore, unreliable connections do not provide significant enough latency benefits to justify the risk of packet loss. Any mechanism to compensate that risk is likely to have more negative impact for latency.

Also from a latency perspective, the advice to chain work requests to reduce overhead is plain false. It seems the chained work requests are then issued in serial, where separate requests can be processed in parallel.

Just be aware, that all our measurements were conducted on an unsaturated link. Maybe in congestion situations, unreliable and send messages are at an advantage, but for the common and fast case this approach works best.

All in all, simple use cases should use a single continuous write work request. More complex tasks can be implemented using indirect writes with a slotted-pages-like approach.

CHAPTER 4

IMPLEMENTATION

With the knowledge from the observations in Chapter 3, we can now reason about, how low latency communication over an RDMA network should be implemented for transactions. The main conclusion we can draw, is that memory polling has significant latency advantages and should therefore be favoured over other notification mechanisms.

4.1 ONE-TO-ONE COMMUNICATION

First, we can take a look at the simplest form of network communication: Point to point. In this variant, we will only consider an implementation of a single sender and a single receiver. The use case for this system is a series of messages between two systems, where an arbitrary amount of messages can be outstanding and messages can theoretically have unlimited message size.

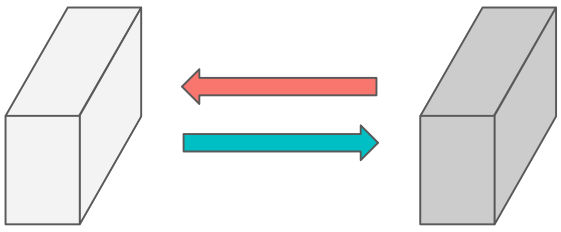

FIGURE 4.1: Simple setup for bidirectional communication

4.1.1 RING BUFFER IMPLEMENTATION

Communication between processes using shared memory is usually based around a ring buffer. When accessing the end of of the buffer, the buffer wraps around to the beginning, allowing to send an infinite amount of messages in a finite amount of memory, as long as messages get read fast enough. The ring buffer can contain up to the ring buffers size of unread messages, so the sender needs to keep track of the last message read by the receiver, otherwise unread data could be overwritten. In a shared memory system, this can simply be implemented by atomic counters of bytes read and written.

For an RDMA environment, we can borrow some ideas from this shared memory communication, with some adaptions. When optimizing RDMA for latency, the utmost goal is to reduce message round trips by reducing the total amount of individual reads and writes. This means, more information needs to be tracked implicitly.

Let r be the count of bytes last read by the receiver, s the count of bytes written by the sender and t the total size of the ring buffer. $s - r$ is then the number of bytes that should not be overwritten due to potential data loss. However, the sender can still write $t - (s - r)$ bytes unconditionally. Therefore, it only needs to update the read counter occasionally, which amortized over the size of the buffer. The overall cost per message can be estimated by: $\frac{\text{Messagesize}}{\text{Buffersize}} \xrightarrow[\text{Buffersize}\to\infty]{} 0$

4.1.2 DETECTING NEW MESSAGES

In Section 3.2.2, we saw, that directly polling memory for incoming messages promises significantly lower latencies than using receive requests. To use memory polling for general purpose messaging, we can use the fact, that the memory location of the next incoming write is known and utilize this knowledge by anticipating the write. Suppose the incoming message writes data $\neq 0$ to the first and last byte of the message. Furthermore, we assume the message is written front-to-back, i.e. write monotonicity, like RDMA guarantees with connected queue pairs. Then, we can detect incoming messages of constant size by polling the first and last memory location of the message, as shown in the listing below.

LISTING 4.1: Implementation of memory polling to detect new messages

```
byte buffer[MESSAGE_SIZE];
while(buffer[0] == 0 and buffer[MESSAGE_SIZE - 1] == 0) wait();
// message received
```

For this technique to work, the buffer needs to be cleared with zeros after each read of the message, so that subsequent polling for new messages can rely on a zeroed buffer. An alternative approach would be to write trailing zeros with each message, which are overwritten by the next message.

4.1.3 Message Format

To generalize this method for arbitrary message sizes, the messages we send need to have a well defined header and footer $\neq 0$. In our case, we define the header to contain an 8 Byte size, which defines the length of the following payload. After the payload, the footer can be arbitrary, but fixed $\neq 0$. This total message can then be written as a single continuous write request, which should be optimal for latency.

4.1.4 Bootstrapping

Setting up an RDMA connection requires connecting the reliable connected queue pairs and exchanging information about the address and length of the buffer. This information can be exchanged using a standard TCP connection, since this setup process is not latency sensitive. Over the TCP connection, both ends first exchange identifiers necessary to connect the queue pairs: port local identifier (LID) and RDMA queue pair number (QPN). The next message contains: The address of the buffer, the corresponding memory region's RDMA remote key (rKey), the address of the counter of bytes currently read and this counter's rKey. With this information, the queue pair states can be transitioned to RTS (as described in Section 2.2) and messages can be exchanged.

4.1.5 Results

In Figure 4.2 we compare direct memory access latencies with socket implementations. In the first graph, our implementations of direct memory access communication are compared, while the second graph shows both socket based variants. A shared memory ring buffer (cf. Section 4.1.1) unsurprisingly has the best overall performance. Our RDMA implementation has significantly worse performance, but considering that it interconnects two separate computers, is very competitive.

In contrast, Unix domain sockets for local and TCP sockets for remote communication are an order of magnitude slower than the shared memory approaches. In conclusion, RDMA can give similar improvements to the use of local shared memory: Using shared

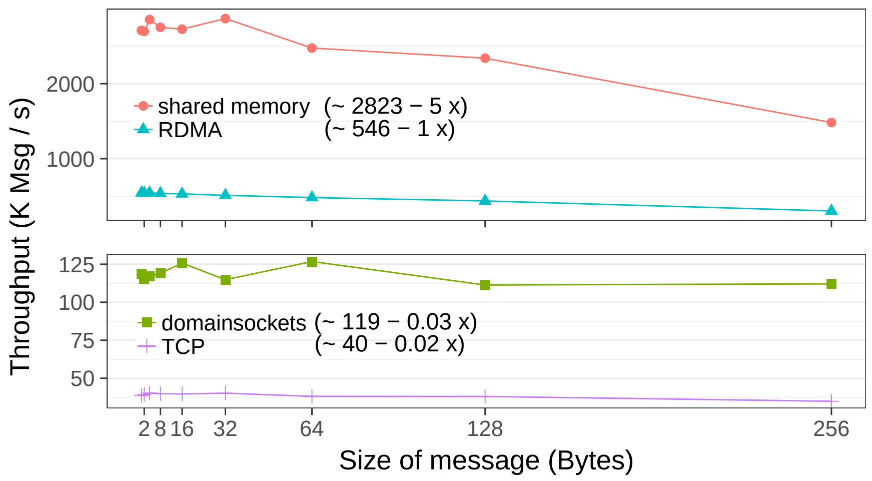

FIGURE 4.2: Latencies of client to client implementations: Both RDMA and shared memory scale linear with message size, while the socket based implementations have nearly constant throughput, because their performance is dominated by overhead.

memory instead of domain sockets gives a 30 × speedup, while using RDMA instead of TCP increases throughput by a factor of 13 ×.

To also test the scale out performance, Figure 4.3 shows the results of a benchmark with multiple parallel connections, where 64 Byte messages are echoed between client and server. In this benchmark, each thread on the server creates its own RDMA resources, only sharing the physical interface with other threads. The overall throughput increases up to 20 parallel connections. This correlates with the number of execution contexts available on the test machine. The throughput for each client is $\frac{\text{Throughput}}{\text{Connections}}$, with the average round trip time (RTT) the inverse of that value. The RTT gradually increases from 2 µs up to around 2.8 µs, with steep performance drops with more than 20 threads.

4.2 N-TO-ONE COMMUNICATION

Unfortunately, the previous system of one-to-one communication does not scale to many connected clients. Every connection needs a separate communication channel, where each channel is polled by a separate thread. This effectively limits the scaling to the number of execution contexts, in our case 20.

Therefore, we also implemented a system in which one thread can effectively poll multiple incoming connections, allowing a single thread to serve multiple clients. The

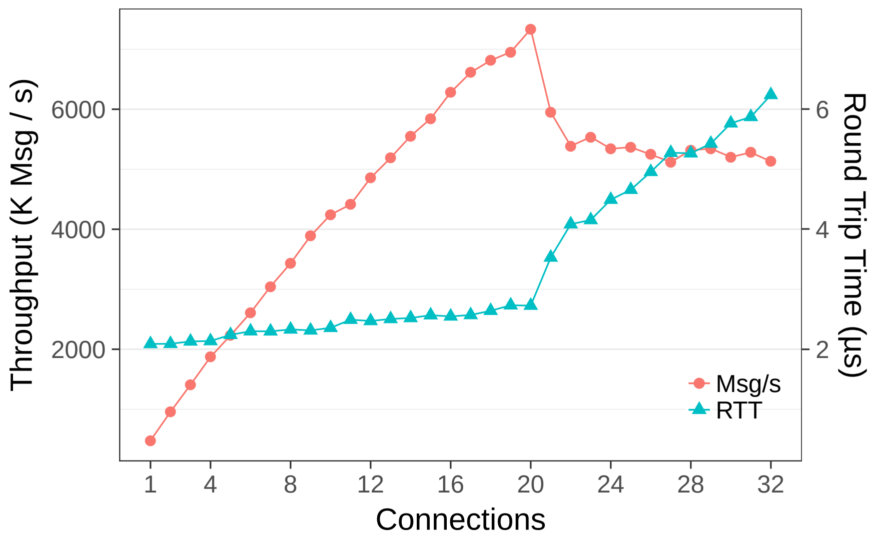

FIGURE 4.3: Multithread scale-out performance of one-to-one communication: N clients sending 64 Byte messages to equally many echoing servers

main goal is to enable performance, that is on par with the one-to-one communication in the fast case, while also effectively handling multiple clients.

For this use case, several assumptions used in the one-to-one case do not hold anymore. E.g. a unique position for the next incoming message, that can be effectively polled does not exist anymore. Incoming messages need to be handled via a write indirection (cf. Section 3.3), while outgoing messages can be handled via a dedicated one-to-one channel. This is useful, if many clients connect to a single server, but do not completely saturate their connection, since they only send messages infrequently.

4.2.1 RANDOMIZED WRITES

Our first approach was to issue a single write work request with an immediate value to indicate the position of the data. This requires a large enough buffer, where writes can happen at random positions. Since in this configuration, collisions between writes are unlikely, the common case should be fast.

To estimate how many collisions between messages will occur, we can look at the likelihood that this will happen. Let m be the message size and b the size of the buffer. Then, the probability of two messages colliding is $p = \frac{m(m-1)}{b} \approx \frac{2m}{b}$, because only a single byte in both messages needs to overlap, which effectively doubles the collision target. Now assume, there are c clients, each sending with a rate of r messages per second and with a message lingering time t. When each client sends messages

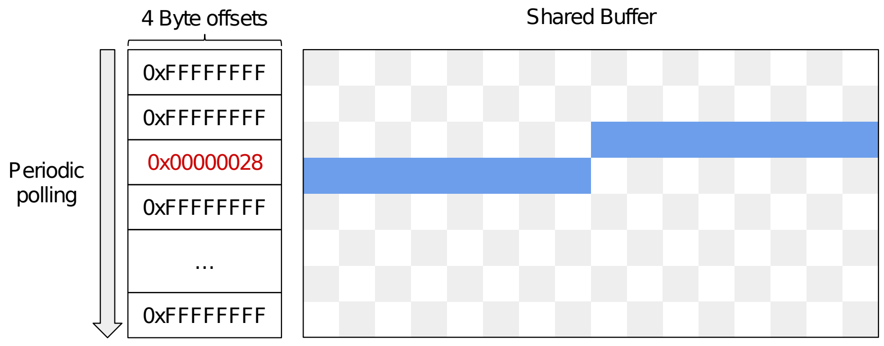

FIGURE 4.4: Visualization of a message sent with a shared buffer and offsets into the buffer

uniformly distributed every $\frac{1}{r}$, then we get a probability of $\frac{t}{\frac{1}{r}} = tr$ for simultaneous messages. We then expect to have $E = 1 - c(1 - tl)^{c-1}$ messages that can potentially collide with probability p in each unit of time.

For example, in a setting with 100 clients, each sending 1000, 64 Byte messages per second, i.e. one every 1 ms with lingering times equal to the latency $\leq 3\,\mu s$. With a server buffer of 128 MByte, we get:

$$p = \frac{2 \cdot 64}{128 \cdot 2^{20}} = \frac{1}{2^{20}}$$

$$\begin{aligned} E &= 1 - 100 \cdot (1 - (1 \cdot 3 \cdot 10^{-3}))^{99} \\ &= 1 - 100 \cdot 0.997^{99} \approx 1 - 74.3 \approx 26 \\ p_{\text{error}} &= \frac{26}{2^{20}} = 2.48 \cdot 10^{-5} \end{aligned}$$

Since those probabilities are for each millisecond, the mean time to happen for an error is approximately:

$$\frac{2^{20}}{26} \text{ms} = 40.33\,\text{s}$$

Unfortunately, performance numbers from Figure 3.6 immediately disqualify the usage of writes with immediate values. Figure 4.4 displays a refined approach using multiple write requests. In this implementation, there still exists a shared buffer, but instead of using immediate values, a continuous array of positions is used to indicate the indirect writes. When a client connects, it gets assigned a position in the buffer of offsets and the server remembers this association. For the offsets, we used `UINT_MAX` as the indicator for no message pending in this slot.

The server can now effectively poll for incoming messages by periodically scanning the offsets array for valid offsets. When an incoming message was detected, the position is reset to no message pending. At the detected offset, the first 8 Byte contain the message size, and the message from `buffer[offset + 8]` onwards can be processed.

A major flaw in this implementation still exists, since there can be collisions between parallel incoming messages. A proposed collision detection mechanism would be to calculate a checksum over the senders id (to detect identical messages from different senders) and all of the data. This checksum would be transmitted out of bounds, similarly to the message position. The receiver can then also calculate the checksum and detect corrupted messages. When the checksum does not match, a special message is sent to the initial sender, which triggers a retransmission. Since collisions are unlikely, the optimal case is still fast, but occasional retransmissions take a bit longer.

For the checksum, we can take a look at CRC32, which has built-in hardware support in modern processors. Assuming a perfect 32-bit CRC, which only fails to detect 1 in 2^{32} collisions, the mean time of a CRC collision to happen is:

$$2^{32} \cdot 40.33\text{s} = 1.73 \cdot 10^{11}\,\text{s} \approx 5489\,\text{years}$$

While this is an acceptable rate for 100 000 messages per second, the guarantees might still be too little when accounting higher scaling systems. A CRC64 checksum should have sufficient guarantees to detect collisions with a mean time to happen of 1700× the age of the universe.

4.2.2 Mailbox Flags

The randomized writes approach can transmit arbitrarily large messages, but many use-cases, especially transaction processing, only need small messages. To make use of this fact, we limit maximum the message size and give each connection an exclusive buffer of that size. An effective size for this limitation would be exactly the same as the limits of `IBV_SEND_INLINE`. Crossing this limit already has a rather large latency jump (see Figure 3.2), so we do not induce any additional limitations for latency sensitive tasks.

Each client has exclusive write access to a buffer, which eliminates the possibility of message collisions. Additionally, this gives a predictable estimate of memory reserved for each connection. Since the inline message size is typically limited to a maximum of 512 Byte, memory overhead per connection is rather small. E.g. with a buffer of 1 MByte, messages of up to 2048 clients could be handled.

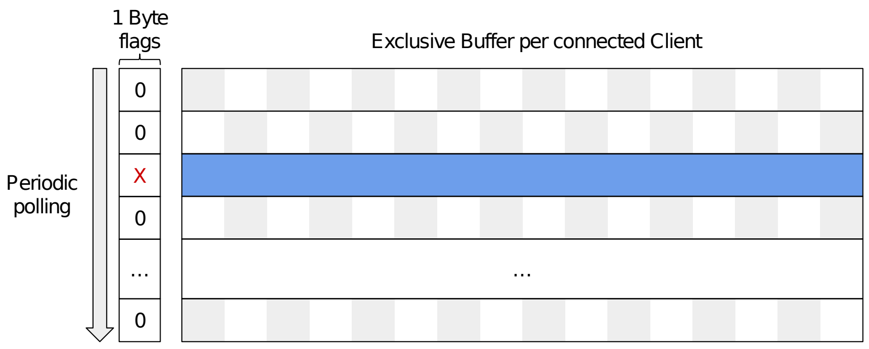

FIGURE 4.5: Visualization of a message sent via an exclusive buffer with mailbox flags

In this concept, messages do not need to be written at random offsets to avoid collisions. This means, each connected client only needs a simple flag to signal incoming messages, just like a mailbox flag. RDMA supports writing single bytes, so the flags can be 1 Byte. This also has the advantage, that polling the memory should be faster, since less memory needs to be scanned for incoming messages.

4.2.3 OPTIMIZATIONS

Additionally, for many connected clients, implementations featuring vectorized instructions should be at an advantage. In our case, where memory is changed by the NIC, we need to work with `volatile` memory, where auto vectorization applied by compilers is not available. In Figure 4.6 we compare latencies of sending messages over RDMA while polling a 1 Byte flag in comparison to polling offsets. We also benchmarked three variations utilizing vector instructions, where the 4 Byte offsets as well as the 1 Byte flags are polled with SSE `_mm_cmpeq_epi32()` instructions. The flags can also be polled using parallel string comparisons with `_mm_cmpestri()`. Unfortunately, the available hardware (see Chapter 5) did only support the AVX1 instruction set, which does not support 32 Byte integer operations. This means, only a maximum of 16 Byte could be processed per instruction.

From those results, we can see, that polling single bytes has a latency advantage of around 5 %. Vectorization of single byte flags yields an additional, small but consistent improvement. For 4 Byte offsets, we do not see latency improvements when using vectorized instructions. This is probably caused by the fact, that with 32 bit values, the number of memory loads is only reduced by a factor of 4. When loading flags with 16 Byte in parallel, the number of loads is a bottleneck and the use of vectorized instructions gives an advantage. Code fragments, which implement those polling experiments can be found in the Appendix, Listings A.2 to A.8.

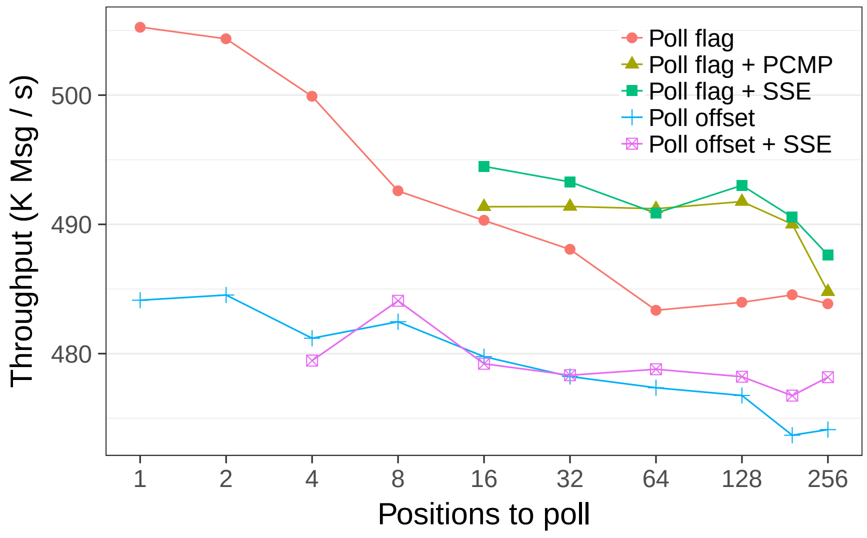

FIGURE 4.6: Comparison of multi client polling mechanisms to detect incoming RDMA messages, which transmit 64 Byte data

Our final N-to-one implementation uses single byte flags and limits the message size to 512 Byte. Figure 4.7 shows such a proposed setup, where a central instance can effectively receive messages, but is limited to receiving small messages. However, outgoing messages do not have such restrictions and can be larger.

This approach has several drawbacks, which need to be considered when using the implementation. As previously discussed, we limited the message size to a relatively small amount. We also do not support multiple outstanding messages, because when a previous message already occupies the buffer, subsequent messages would overwrite it. It is therefore necessary, that the sender coordinates and serializes its messages. Limited outstanding messages can be supported by opening multiple connections, which then are processed out of order.

Furthermore, the current implementation does not support dynamic client management, i.e. clients arbitrarily dis- and reconnecting. All session management could be implemented based on TCP socket keep alive mechanisms, which mark specific RDMA queue pairs as disconnected. These queue pairs then need to be reset, but could be reused for future clients.

Another potential problem is, that malicious clients could fake messages from other clients. Libibverbs does not support limiting the memory region to a specific client, but all registered memory regions are writable by all connected clients within a protection domain. It is possible to create a separate protection domain for each client in the

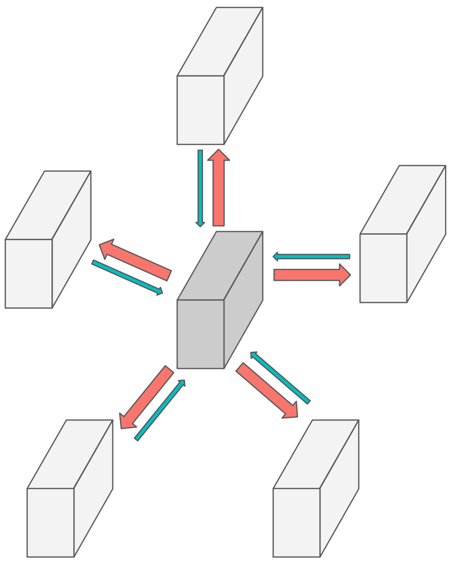

FIGURE 4.7: Use case scenario for N-to-one communication. The central server can only receive small messages, but send arbitrary sized responses.

one-to-one case, but the N-to-one implementation cannot work this way, since efficient polling cannot work across multiple memory regions and protection domains. Similar spoofing attacks exist in traditional networks, where clients can impersonate others by changing their MAC addresses. In wireless networks with many untrusted devices, this is problematic, but in small and wired networks with trusted devices, this is a non-issue.

A common performance pitfall when scaling for multiple clients is libibverbs' handling of work completion. Usually, for each work request on the server a flag `IBV_SEND_SIGNALED` can be set, when a completion is needed. Each work request with this flag set generates a work completion in the corresponding completion queue. For inline messages, the server side does not need the information when a work request has been completed. Therefore, unsignaled requests can be used, which saves time otherwise spent on processing unnecessary work completion. Due to limitations of libibverbs, this does not work all the time, since even unsignaled operations use resources in the completion queue. Thus, too many unsignaled writes might result in a `ENOMEM` error when posting work request. As a solution, Xue [22] describes a system of selective signaling, “where a signaled send is posted once in a while, and then [the] completion event [is polled] from the CQ”. This process clears the completion queue and allows more work requests to be posted without an error, while still having little overhead. The implementation of this optimization for our communication library is shown in the Appendix, Listing A.1.

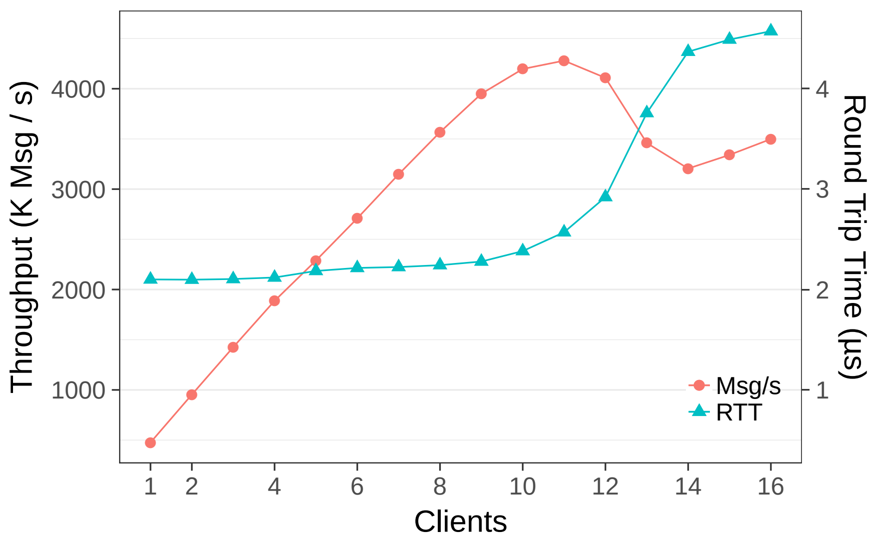

FIGURE 4.8: Performance of a single threaded N-to-one server echoing 64 Byte messages

4.2.4 EVALUATION

A benchmark, measuring throughput and mean latency is shown in Figure 4.8. This system has all the previously discussed optimizations, like selective signaling and polling incoming messages with SSE in place. In this benchmark, all clients send 64 Byte messages sequentially at the maximum rate. On the server side, a single thread continuously polls for incoming messages and echos the same 64 Byte message back to the sender. Each client then verifies the received data and continues to send the next message.

In result, we can see, that a single server thread can handle up to 4 million incoming messages per second. Each client can send approximately 500 000 messages per second with an average latency of around 2.1 µs. With 10 clients, the server starts to get saturated and latencies start to increase more and more. Already with 14 clients, the overall throughput goes down and latencies have more than doubled.

4.3 BIG MESSAGES

Since transmitting big messages is not supported by this N-to-one implementation, we need to consider which restrictions this design decision imposes. When looking at big messages in general, latency are significantly higher. Optimizing for latency should therefore first and foremost be a reduction of message sizes.

Link aggregation, advertised as 4 × or 8 × does not help for messages latencies, since each of the links still operates at the same speed. E.g. QDR links have a 10 Gbit/s signaling rate, which means, that messages of 50 KByte already have serialization times alone, which are as long as the latencies we get for small messages. Even for way smaller messages, we see a clear trend: when the data size exceed 128 Byte, we get diminishing returns for our improved message processing.

Fortunately, small messages are perfectly fine for most transactional workloads. E.g. in the TPC-C [21] schema, only a single field, (`C_DATA`) is big enough to maybe exceed our limitations with a maximum of 500 characters[1], while all other fields are an order of magnitude smaller. Even then, the field is only used in “The Payment Transaction”, while all other transactions easily fit within the limitations of our implementation.

Nevertheless, bigger messages can still use a fallback method. A good approach would be, to open a dedicated one-to-one channel, with special messages in the N-to-one channel to trigger a method switch. Another approach would be to just use regular send and receive primitives, which have higher latency than a dedicated channel, but do not need to be set-up before use. The latency differences are not as drastic as for small messages anyway, because most of the time is spent transmitting the data. Since our use case did not need such big messages, no fallback path was implemented in this work.

4.4 ZERO COPY OPERATION

Our C++ implementation enables zero copy operation with higher order functions. Instead of traditional `send(void*, size_t)` function prototypes, the send and receive functions can be passed a callback lambda, which can directly perform computation with the received data, without the need to copy data. E.g. when a key-value store receives a request, in the traditional API the requested key is first copied to a specified location and then used to retrieve the data. With a callback lambda, the copy can be avoided and the lookup can directly operate on the network buffers within the lambda.

The following code fragment shows an experiment, which sends data to a server and expects to receive the same data back. Because in this particular experiment, we already generated the test data outside of the benchmark loop, the zero copy call

[1] `C_DATA` is a `VARCHAR(500)` field, which only needs to be accessed for 10 % of all rows

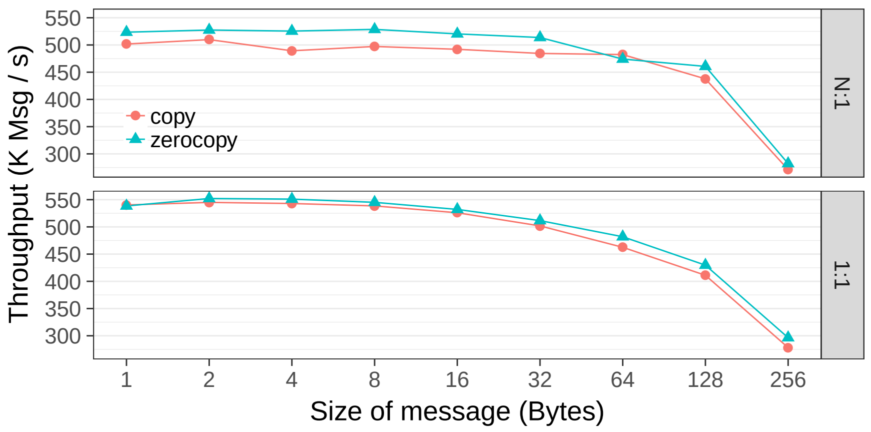

FIGURE 4.9: Impact of zero copy operation on an echoing workload

actually needs to copy the data. However, for the receive call no data needs to be copied, since we only verify the received data, which can be done in place.

LISTING 4.2: Zero copy enabled ping client

```
client.send([&](auto begin) {
    std::copy(testdata.data(), testdata.data() + size, begin);
    return size;
});
client.receive([&](auto begin, auto end) {
    if (not std::equal(begin, end, testdata.data())) throw runtime_error("NEQ");
});
```

This example shows, that while this technique might not be universally applicable, it gives more control to the caller, allowing more fine-grained control over which data is actually copied. Benchmarking this example versus the traditional, copying API shows, that zero-copy allows for around 5 % (0.1 µs) lower latency, as shown in Figure 4.9. The difference is relatively small, since the overall work of copying data between buffers only takes a fraction of the time. For this particular workload, the optimization is easy and worthwhile. For real applications, results might vary and should definitely be measured, since the overall impact is largely dependent on the workload.

CHAPTER 5

EVALUATION

In this chapter, we evaluate our networking library, as described in the previous chapter, on a database workload. All systems, on which our experiments ran, had identical hardware. Each machine featured dual Intel Xeon E5-2660 v2 processors at 2.2 GHz, with 10 cores and 20 hyper-threads, each. Each machine had 256 GByte DDR3 memory available, running at 1866 MHz. For network communication we used Mellanox ConnectX-3 VPI NICs, connected via an Infiniband network at $4 \times$ QDR with a nominal data rate of 40 Gbit/s.

The operating systems unfortunately were not identical, since one machine was still running Ubuntu 16.04.1 LTS, while the other machine was already updated to Ubuntu 17.10. To keep the relevant RDMA software comparable, the software stack was updated using the rdma-core repository[1].

On this platform, we ran experiments to evaluate the transactional performance with a standardized benchmark and noticed challenges with the dual processor platform.

5.1 YAHOO! CLOUD SERVING BENCHMARK (YCSB)

Cooper et al. [2] created a benchmark for cloud applications with a set of defined operations on a key-value store, called Yahoo! cloud serving benchmark (YCSB). In this benchmark, there is only a single table with a 32 bit primary key and records with ten

[1] Available online, `https://github.com/linux-rdma/rdma-core`

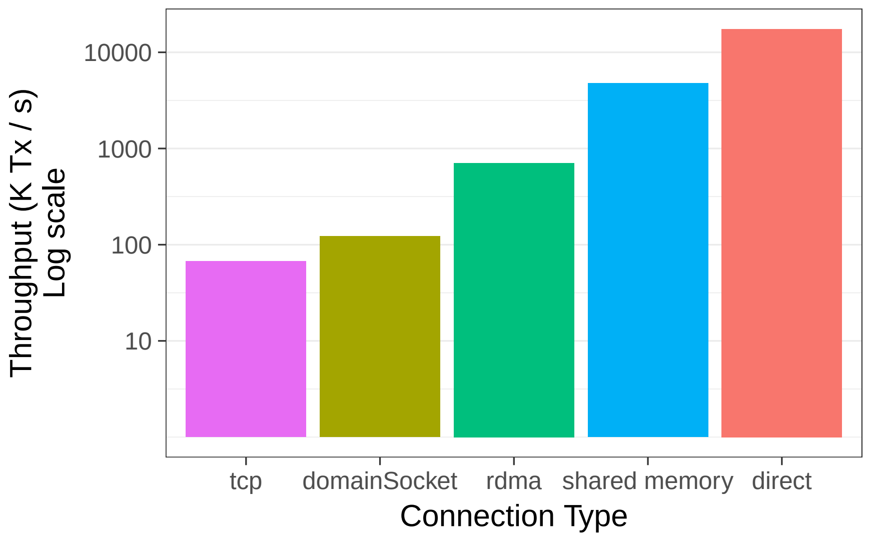

FIGURE 5.1: YCSB benchmark (8 Byte request, 100 Byte response) over different connections on localhost

100 Byte fields each. Typical workloads read tuples with Zipfian distributed probability to model real-world data skew [6] and a uniformly distributed random selection of the field.

YCSB is a widely used benchmark for transactional systems, which enables fair comparisons between them. To showcase our RDMA library, we implemented a proof of concept key-value store. This system features an in-memory hash table, based on libcpp's `unordered_map`, which can accept requests via different communication mechanisms.

Without loss of generality, we selected YCSB workload C, which is a read-only workload. However, this is not a limitation of the networking or RDMA component, but a measure to reduce noise within measurements. Mixed read and write workloads can be implemented similarly, with additional work, but would need to consider thread safety.

The results of a single threaded benchmark with different communication mechanisms are displayed in Figure 5.1. These experiments were run on localhost, i.e. via the loopback interface, to get an upper performance bound for each of the connections. Please note, that the throughput is plotted on a logarithmic scale, since the performance differences are rather dramatic. A loopback TCP transaction took on average 15 µs, the same transaction over domain sockets 8.3 µs, over RDMA 1.4 µs, and with shared memory communication 0.21 µs. In comparison to that, running the hash table in a loop, without communication with others only takes 0.057 µs on average.

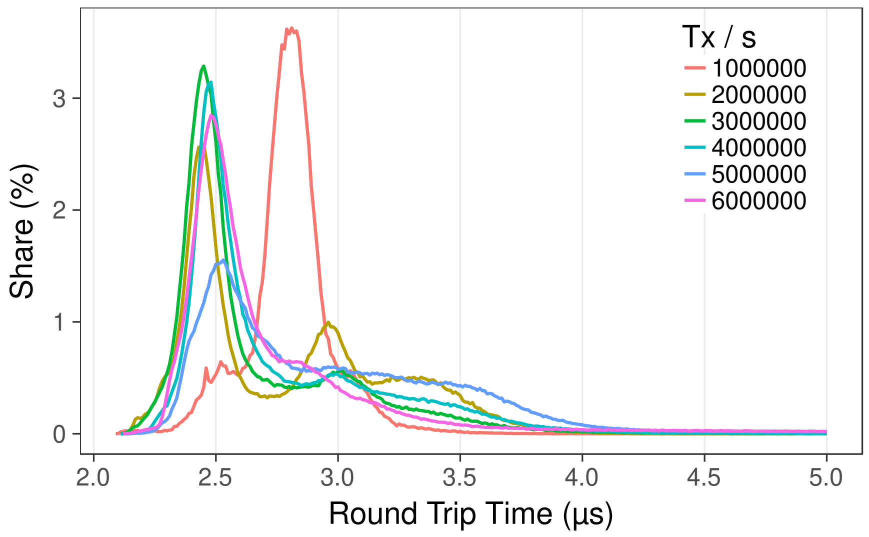

Figure 5.2: Latency distribution of YCSB requests from multiple senders to a single N:1 server

Table 5.1: Statistics of message response times with different loads (in µs)

Tx / s	mean	median	99 percentile	max
1000000	2.80	2.80	3.29	4.75
2000000	2.79	2.67	3.77	6.96
3000000	2.67	2.51	3.89	8.93
4000000	2.76	2.58	3.97	11.79
5000000	3.30	2.89	11.44	71.13
6000000	4.35	2.59	24.45	9109.43

When running the benchmark via an actual network, in optimal non-uniform memory access (NUMA) configuration, we can get a performance of around 440 000 sequential lookups per second, with an average latency of 2.3 µs.

Figure 5.2 displays the latency distribution of requests from multiple clients to a single threaded YCSB database server. For this benchmark we used different loads on the database, between 1 and 6 million requests per second. We then measured the response time for each transaction under that load conditions. To reduce measurement noise, we averaged the round trip times of 50 consecutive round trips. Afterwards, the RTTs were rounded to the nearest 1/100 µs with the overall share of each of those slices plotted on the vertical axis. This graph only shows samples up to 5 µs, which captures the 99 percentile of most loads. Overload situations have significantly longer tails, which would render the graph unreadable.

To increase the load on the database, we steadily increase the number of clients, all sending at constant rates. E.g. to execute 1 M transactions per second, we have 10

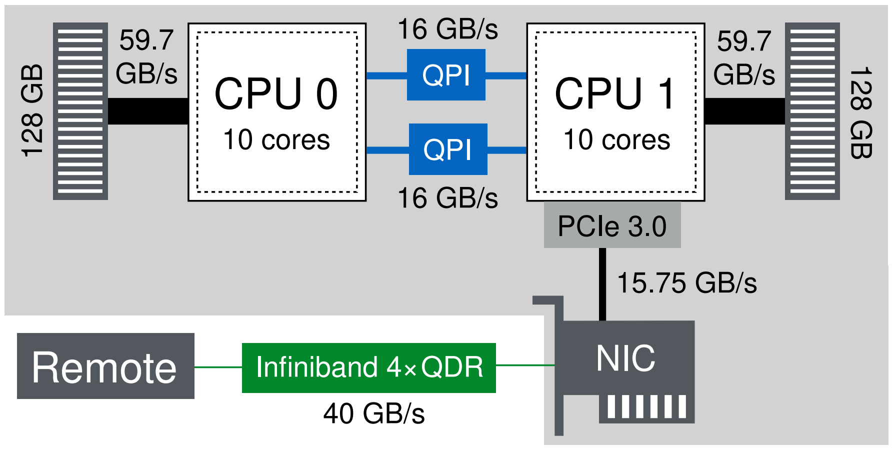

FIGURE 5.3: NUMA setup used in all measurements [19]

threads, exclusively running on execution contexts, running 100 000 transactions per second. 2 M were reached with 20 threads, 3 M with 30 and so forth.

Curiously, the average latencies of 1 M transactions per second were actually higher than with higher loads. This looks like an outlier, but measurements were consistent over multiple runs. A possible explanation for such behaviour would be power saving features, which are triggered by rate limiting. When comparing higher loads, we have much more consistent latencies with typical round trip time of around 2.5 µs. Median round trip times actually stay approximately the same, while mean, maximum and 99 percentile values increase. An overload of the server starts with around 5 M transactions per second with significantly higher maximum round trip times.

The overload situation with 6 M transactions per second still has acceptable median response times, but abysmal maximum response times of over 9 ms. This is probably caused by the polling mechanism, which always starts by polling the first client. If two clients simultaneously send a message, the first polled client always wins, which leads to starvation of later polled clients.

5.2 NUMA IMPACT

The dual CPU architecture of our systems is displayed in Figure 5.3. The two distinct CPUs are connected via Intel QuickPath interconnect (QPI) [3]. Each CPU comes with its own memory and I/O subsystem, resulting in a combined system with NUMA. Because the NIC used to communicate with the Infiniband network is only connected to the

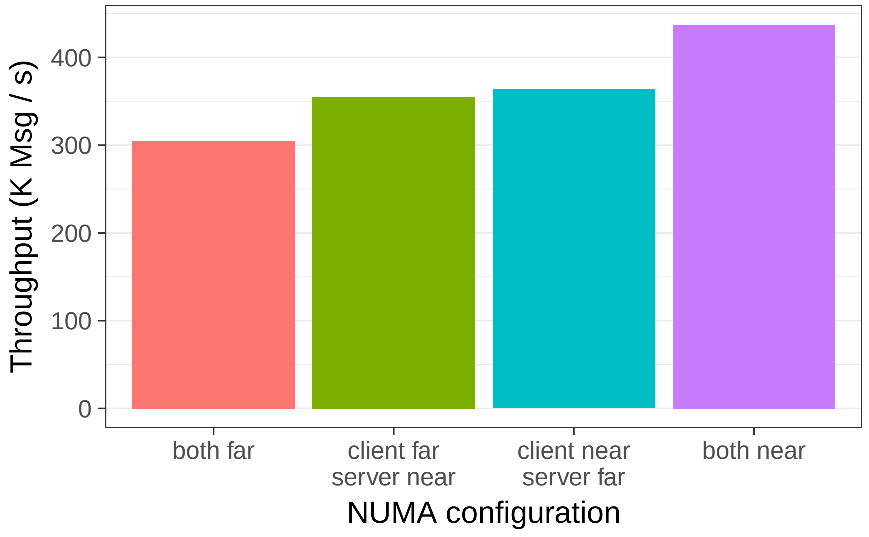

FIGURE 5.4: Impact of NUMA on RDMA latencies of a key-value database

PCI express bus of one of those CPUs, the network operations also have non-uniform latencies.

Figure 5.4 visualizes this effect on networking latencies by running a benchmark on different NUMA nodes (cf. Appendix, Listing A.1). The first column shows the extreme case, where each network message needs to be written from the CPU via QPI to the NIC on the sending side. On the receiving side, the NIC also needs to write the received data through QPI to the destination memory. In total this adds the NUMA latency twice. The middle columns show the effect of only running one side of the connection on the higher latency NUMA node. In those measurements, running the receiving side on the lower latency node has a slight advantage, but those differences are not necessarily significant.

When running both sides in optimal configuration, we get a performance improvement of > 40 % in comparison to the worst case. RDMA can be seen as an extreme case of NUMA, since the difference between local system latency and network latency is only a factor of 3. Alternatively, as Rödiger et al. [19] put it, NUMA and RDMA can both be considered networks.

However, accounting for NUMA impact is usually not a big deal. Many systems do not even have multiple CPU sockets, such that memory access is naturally uniform. A possible solution, enabling full performance for NUMA systems, would be to use a NIC for each NUMA node or to simply not use the far node for networking. Therefore, we performed all experiments on the near to near setup.

Chapter 6

Related Work

This work is not the first to use RDMA for fast networking. Rödiger et al. [19] already implemented high-speed join processing and query processing using RDMA. In their system, they built a network aware query processing engine. This query engine tries to minimize network contention by generating query execution plans with special exchange operators and coordinated network operation scheduling. By additionally factoring in NUMA effects, they achieve significant speedups compared to other distributed query processing engines.

In contrast to our use-case, their system focused on 512 KByte or larger messages. Big messages hide the latency overhead, so that high throughput with low CPU overhead becomes the main characteristic of RDMA. In our work, we focused on OLTP workloads, which has network payloads several orders of magnitude smaller.

In another approach Kalia, Kaminsky, and Andersen [10, 9] implemented a key-value store, that utilizes RDMA very efficiently for small messages. Their system scales up to 26 million parallelized operations per second, with average latencies of 5 µs as reported in their first paper and 2.8 µs as reported in their second paper.

They achieve this performance using the basic idea of remote procedure calls (RPC) using RDMA send and receive messages, instead of directly reading or writing the desired memory, because network latency is usually the dominating factor for most OLTP workloads. In combination with techniques of request and response batching, which reduce NIC to CPU communication, their system excellently scales for parallel workloads.

Compared to theoretical latencies of $\leq$ 1 µs, this system seems still not optimal when focusing on latency. Furthermore, they heavily use RDMA in unreliable datagram (UD) mode, which they reason to be reliable, since “packet loss is extremely rare” [9] in RDMA networks. However, when providing ACID guarantees, there is no slack for eventualities. Even when “observ[ing] zero packet loss in over 50 PB of data transferred” [9], that does not mean, catastrophic failure cannot happen any time. In contrast to their work, we showed, that OLTP systems with even lower latency are possible while still providing a reliable connection.

An earlier system by Dragojević et al. [4] called FaRM, used a message passing approach using RDMA writes, similar to ours. Their system also used a ring buffer and a message detection approach which polled the memory location of the next message. This message passing system was used for writing transactions, which are synchronized by the remote CPU with memory barriers. Read operations are entirely handled by speculative RDMA reads, which might fail if the object was locked by a writing transaction. In contrast to our system, they built an one-sided system without interaction of the remote CPU for reading operations. The main advantage is, that the remote system does not need to interact with requests, but this effectively wastes the compute power on one system.

While their approach works great to pass individual messages, each access to data requires multiple network operations. With around 2 µs for network round trips, but only around 0.1 µs for main memory lookups, the network latencies add up, so that overall latency is relatively poor. This is an inherent weakness of their multiple round trip approach, which traverses remote data structures. Any system, which uses an one-sided approach has the fundamental problem of chaining high latency network operations. Compared to 31 µs lookup latencies in their system, our single round trip approach is an order of magnitude faster.

Pilaf [17] implemented a cuckoo hash table with self-verifying entries, using a similar check summing approach as the proposed system discussed in Section 4.2. Pilaf clients delegate writes to the server by using RDMA send and receive messages. This allows for efficiently synchronized writes on the server, with only read-write conflicts from RDMA. Reads are implemented as RDMA read operations, which might collide occasionally, but concurrent writes are efficiently detected using CRC64 and the reads eventually retried. However, the remote traversal of the hash table still causes a huge increase of lookup latency.

Szepesi et al. [20] proposed Nessie, as a system which not only allows one-sided reads, but also one-sided writes using RDMA atomic compare-and-swap operations. As a result, the remote side has no involvement in managing and synchronizing the underlying hash table anymore. Unfortunately latencies still suffer from the multiple

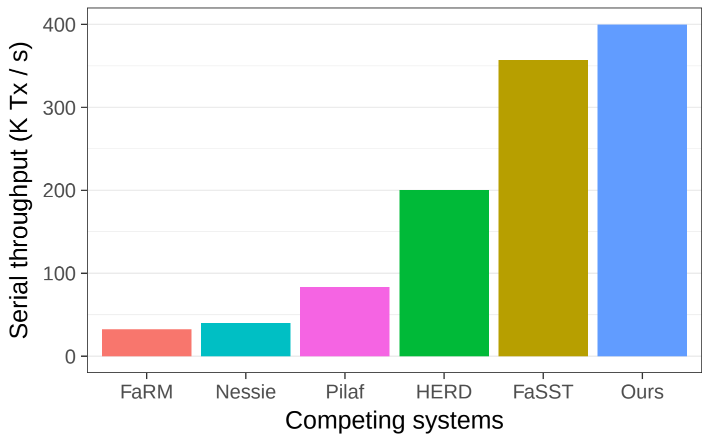

FIGURE 6.1: Latency comparison of different RDMA key-value stores, as reported in their publications

round trips approach and are still an order of magnitude worse than using single round trip approaches.

Figure 6.1 shows a non-representative comparison between the discussed systems. All values for this graph were taken directly from literature and do not implement the same benchmarks. Nevertheless, all systems benchmarked similar key value stores. FaRM used 16 Byte keys with 32 Byte values and achieved latencies of 31 µs. Nessie reports 25 µs latency for write requests of unspecified sizes. But their work also reports negligible latency differences for 10 Byte or 1 KByte requests, which is rather surprising, given that our measurements show a clear correlation between size and round trip time. Pilaf reports numbers as low as 12 µs for their benchmark of small (16 Byte) one-sided lookups in their hash table. Herd - as the first system to use both sided lookups - benchmarked 48 Byte items with a read intensive workload and reached average latencies of 5 µs. The only competing system reporting their latency of 2.8 µs on a standardized benchmark (TATP) was FaSST. This benchmark uses a key-value store with 8 Byte keys to look up 40 Byte values. Our YCSB benchmark had 8 Byte keys and comparatively large 100 Byte values as described in Section 5.1 with an average latency of 2.3 µs.

Chapter 7

Conclusions

Based on our work, we can conclude, that databases can get massive latency improvements by using RDMA networking. The major network bottlenecks are already eliminated when using any RDMA primitives alone. Our optimizations, as presented in this work, can additionally reduce latencies by almost 50 % for small workloads. With such dramatic latency improvements of over an order of magnitude in total, transactional workloads over networks should unconditionally use RDMA.

As Section 4.1 showed, latency bound applications do not saturate the NIC or the Infiniband network when only using a single thread. Our approach therefore uses a single threaded server handling multiple connections (Section 4.2), which allows efficient scaling to many more threads.

All in all, we built a system, which is not limited to transaction processing, but can also be used for general purpose message passing. While it is optimized for OLTP workloads, there are actually no conceptual limitations. This allows our system to be adopted by any application in need for low latency networking.

Future Work

In future work, it might be possible to adapt techniques that are useful on NVM systems to RDMA setups. Since RDMA also provides byte addressability, similar systems like the buffer manager Renen et al. [18] implemented for NVM, can be built using our library. Such a system would promise to make transactions over the network comparably fast to in-memory systems.

Another topic for future work might be a system for remote procedure calls (RPCs). In our current system, we only send messages to the remote side. The messages themselves need to contain information about initiated actions on the remote side. This process can be further optimized, e.g. by instead of transmitting message sizes, have the size implicitly declared as the size of the arguments and only transmit an identifier for the remote called procedure. In N-to-one communication, there is also the signal byte, which only has the purpose of a flag. This byte could also be repurposed to encode up to 255 different procedure calls.

Finally, the RDMA communication should be integrated into a fully featured database system. Ideally, this database would allow calling stored procedures as RDMA RPC using messages via our communication library.

CHAPTER A

APPENDIX

LISTING A.1: Benchmarking helper script, which uses numactl to pin threads to the near NUMA node, logs the benchmarking output to a CSV file and displays the data in the terminal.

```
#!/usr/bin/env bash
set -e
if [[ $# -lt 2 ]]; then
    echo "Please␣specify␣executable,␣client␣/␣server,␣optionally␣IP"
    exit
fi
set -x

NODE=1
numactl --membind="$NODE" --cpunodebind="$NODE" ./"$1" "$2" "$3"\
    | tee "$1"_"$2".csv\
    | column -s, -t
```

Polling code used to detect incoming messages:

LISTING A.2: Scalar polling of mailbox flags

```
static size_t poll(char *doorBells, size_t count) noexcept {
    for (;;) {
        for (size_t i = 0; i < count; ++i) {
            if (*(volatile char *)(&doorBells[i]) != '\0') {
                doorBells[i] = '\0';
                return i;
            }
        }
    }
}
```

LISTING A.3: SSE polling of mailbox flags

```
static size_t pollSSE(char *doorBells, size_t count) {
    assert(count % 16 == 0);
    const auto zero = _mm_set1_epi8('\0');
    for (;;) {
        for (size_t i = 0; i < count; i += 16) {
            auto data = *(volatile __m128i *)(&doorBells[i]);
            auto cmp = _mm_cmpeq_epi8(zero, data);
            uint16_t cmpMask = compl _mm_movemask_epi8(cmp);
            if (cmpMask != 0) {
                auto lzcnt = __builtin_clz(cmpMask);
                auto sender = 32 - (lzcnt + 1) + i;
                doorBells[sender] = '\0';
                return sender;
            }
        }
    }
}
```

LISTING A.4: PCMP polling of mailbox flags

```
static size_t pollPCMP(char *doorBells, size_t count) {
    assert (count % 16 == 0);
    constexpr auto flags = _SIDD_NEGATIVE_POLARITY;
    const auto needle = _mm_set1_epi8('\0');
    for (;;) {
        for (size_t i = 0; i < count; i += 16) {
            const auto haystack = *(volatile __m128i *)(&doorBells[i]);
            const auto match = _mm_cmpestri(needle, 1, haystack, 16, flags);
            if(match != 16) {
                const size_t sender = match + i;
                doorBells[sender] = '\0';
                return sender;
            }
        }
    }
}
```

LISTING A.5: AVX2 polling of mailbox flags (not benchmarked, due to the lack of hardware support)

```
static size_t pollAVX2(char *doorBells, size_t count) noexcept {
    assert(count % 32 == 0);
    const auto zero = _mm256_setzero_si256();
    for (;;) {
        for (size_t i = 0; i < count; i += 32) {
            auto data = *(volatile __m256i *)(&doorBells[i]);
            auto cmp = _mm256_cmpeq_epi8(zero, data);
            uint32_t cmpMask = compl _mm256_movemask_epi8(cmp);
            if (cmpMask != 0) {
                auto lzcnt = __builtin_clz(cmpMask);
                auto sender = 32 - (lzcnt + 1) + i;
                doorBells[sender] = '\0';
                return sender;
            }
        }
    }
}
```

LISTING A.6: Scalar polling of message offset

```cpp
static std::tuple<size_t, int32_t> poll(int32_t *offsets, size_t count) noexcept {
    for (;;) {
        for (size_t i = 0; i < count; ++i) {
            int32_t writePos = *(volatile int32_t *)(&offsets[i]);
            if (writePos != -1) {
                offsets[i] = -1;
                return {i, writePos};
            }
        }
    }
}
```

LISTING A.7: SSE polling of message offset

```cpp
static std::tuple<size_t, int32_t> pollSSE(int32_t *offsets, size_t count) noexcept {
    assert(count % 4 == 0);
    const auto invalid = _mm_set1_epi32(-1);
    for (;;) {
        for (size_t i = 0; i < count; i += 4) {
            auto data = *(volatile __m128i *)(&offsets[i]);
            auto cmp = _mm_cmpeq_epi32(invalid, data); // sadly no neq
            uint16_t cmpMask = compl _mm_movemask_epi8(cmp);
            if (cmpMask != 0) {
                auto lzcnt = __builtin_clz(cmpMask);
                // 4 bits per value, since cmpeq32, but movemask8
                auto sender = i + ((32 - lzcnt) / 4 - 1);
                auto writePos = offsets[sender];
                offsets[sender] = -1;
                return {sender, writePos};
            }
        }
    }
}
```

LISTING A.8: AVX2 polling of message offset (not benchmarked, due to the lack of hardware support)

```cpp
static std::tuple<size_t, int32_t> pollAVX2(int32_t *offsets, size_t count) noexcept {
    assert(count % 8 == 0);
    const auto invalid = _mm256_set1_epi32(-1);

    for (;;) {
        for (size_t i = 0; i < count; i += 8) {
            auto data = *(volatile __m256i *)(&offsets[i]);
            auto cmp = _mm256_cmpeq_epi32(invalid, data);
            uint32_t cmpMask = compl _mm256_movemask_epi8(cmp);
            if (cmpMask != 0) {
                auto lzcnt = __builtin_clz(cmpMask);
                auto sender = i + ((32 - lzcnt) / 4 - 1);
                auto writePos = offsets[sender];
                offsets[sender] = -1;
                return {sender, writePos};
            }
        }
    }
}
```

LISTING A.9: Selective signaling for faster server side response times

```
auto &con = connections[receiverId];

[...] // message preparation omitted for brevity

if (++sendCounter % 1024 == 0) { // selectively signaled
    con.answerWr.setFlags({Flags::INLINE, Flags::SIGNALED});
    con.qp.postWorkRequest(con.answerWr);
    sharedCq.pollSendCompletionQueueBlocking(Opcode::RDMA_WRITE);
} else {
    con.answerWr.setFlags({Flags::INLINE});
    con.qp.postWorkRequest(con.answerWr);
}
```

CHAPTER B

GLOSSARY

DMA	Direct Memory Access. Independent and direct access to main memory by hardware components other than the CPU, usually via PCIe.
IPC	Inter-Process Communication. Mechanisms, that an operating system provides to allow communication between otherwise isolated processes.
LID	RDMA Port Local IDentifier. A 16 bit identifier which identifies an RDMA device within its subnet [15].
NIC	Network Interface Card. A hardware component, that connects a computer with a Network.
NUMA	Non-Uniform Memory Access. A processor interconnect, used to combine processors to a shared memory architecture.
NVM	Non-Volatile Memory. A storage technology, which combines the byte addressability and low latency of DRAM with persistent storage on power loss.
OLTP	OnLine Transaction Processing. A database workload, that consists of small, mixed reading and writing transactions with high throughput.
QPI	Intel QuickPath Interconnect. A processor interconnect, used to combine processors to a shared memory architecture.
QPN	RDMA Queue Pair Number. An identifier for the queue pair registered with an RDMA device.
RDMA	Remote Direct Memory Access. A networking model, that allows direct access to the memory of another computer.

rKey	RDMA remote key. The remote key of a libibverbs memory region. It is used to identify the memory region for incoming work requests.
RTT	Round Trip Time. The total time elapsed between sending a message and receiving an answer for that message.
TCP	Transmission Control Protocol. Stream-oriented, reliable, transport layer protocol.
WC	Work Completion. The counterpart of a work request. When the work completion for a given work request is emitted to a completion queue, the work request has been processed by the NIC. The work completion is not necessarily positive, but might indicate which error occurred when processing the work request.
WR	Work Request. One of the key components to interact with an RDMA NIC. It is used to instruct the NIC to asynchronously execute operations.
YCSB	Yahoo! Cloud Serving Benchmark. A popular key-value store benchmark framework with different workloads.

Bibliography

[1] Dotan Barak. *Tips and tricks to optimize your RDMA code*. 2013. URL: http://www.rdmamojo.com/2013/06/08/tips-and-tricks-to-optimize-your-rdma-code/.

[2] Brian F. Cooper et al. “Benchmarking Cloud Serving Systems with YCSB”. In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 143–154. URL: http://www.brianfrankcooper.net/home/publications/ycsb.pdf.

[3] Intel Corporation. *An Introduction to the Intel QuickPath Interconnect*. 2009. URL: https://www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf.

[4] Aleksandar Dragojević et al. “FaRM: Fast Remote Memory”. In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. 2014, pp. 401–414. URL: https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-dragojevic.pdf.

[5] Hector Garcia-Molina, Jeff Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice-Hall, 2008.

[6] Jim Gray et al. “Quickly Generating Billion-Record Synthetic Databases”. In: *Acm Sigmod Record*. Vol. 23. 2. ACM. 1994, pp. 243–252. URL: https://jimgray.azurewebsites.net/papers/SyntheticDataGen.pdf.

[7] Jeff Hilland et al. *RDMA Protocol Verbs Specification*. Internet-Draft. IETF Secretariat, 2003. URL: https://tools.ietf.org/html/draft-hilland-rddp-verbs-00.

[8] Anuj Kalia, Michael Kaminsky, and David G. Andersen. “Design Guidelines for High Performance RDMA Systems”. In: *2016 USENIX Annual Technical Conference*. 2016, p. 437. URL: http://www.cs.cmu.edu/~akalia/doc/atc16/rdma_bench_atc.pdf.

[9] Anuj Kalia, Michael Kaminsky, and David G. Andersen. “FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs.” In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design*

and Implementation (OSDI'16). Vol. 16. 2016, pp. 185–201. URL: https://www.cs.cmu.edu/~akalia/doc/osdi16/fasst_osdi.pdf.

[10] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Using RDMA Efficiently for Key-Value Services". In: *ACM SIGCOMM Computer Communication Review*. Vol. 44. 4. ACM. 2014, pp. 295–306. URL: https://www.cs.cmu.edu/~akalia/doc/sigcomm14/herd_readable.pdf.

[11] Alfons Kemper and Thomas Neumann. "HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots". In: *2011 IEEE 27th International Conference on Data Engineering (ICDE)*. IEEE. 2011, pp. 195–206. URL: https://cs.brown.edu/courses/cs227/archives/2012/papers/olap/hyper.pdf.

[12] Viktor Leis, Alfons Kemper, and Thomas Neumann. "The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases". In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE. 2013, pp. 38–49. URL: http://db.in.tum.de/~leis/papers/ART.pdf.

[13] Patrick MacArthur and Robert D. Russell. "A Performance Study to Guide RDMA Programming Decisions". In: *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*. IEEE, 2012. DOI: 10.1109/hpcc.2012.110. URL: http://www.cs.unh.edu/~rdr/rdr-hpcc12.pdf.

[14] Mellanox Technologies Ltd. *ConnectX®-3 VPI Product Brief*. 2013. URL: http://www.mellanox.com/related-docs/prod_adapter_cards/ConnectX3_VPI_Card.pdf.

[15] Mellanox Technologies Ltd. *RDMA Aware Networks Programming User Manual*. 2015. URL: http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[16] Scott Meyers. *Effective modern C++: 42 specific ways to improve your use of C++ 11 and C++ 14*. O'Reilly Media, Inc., 2014.

[17] Christopher Mitchell, Yifeng Geng, and Jinyang Li. "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store." In: *USENIX Annual Technical Conference*. 2013, pp. 103–114. URL: https://www.usenix.org/system/files/conference/atc13/atc13-mitchell.pdf.

[18] Alexander van Renen et al. "Managing Non-Volatile Memory in Database Systems". In: *SIGMOD'18, International Conference on Management of Data*. preprint. 2018. URL: https://db.in.tum.de/~leis/papers/nvm.pdf.

[19] Wolf Rödiger et al. "High-Speed Query Processing over High-Speed Networks". In: *Proceedings of the VLDB Endowment* 9.4 (2015), pp. 228–239. URL: http://www.vldb.org/pvldb/vol9/p228-roediger.pdf.

[20] Tyler Szepesi et al. “Designing a Low-Latency Cuckoo Hash Table for Write-Intensive Workloads Using RDMA”. In: *First International Workshop on Rack-scale Computing*. 2014. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/02/szepsei14designing.pdf.

[21] Transaction Processing Performance Council. *TPC Benchmark C, Standard Specification Revision 5.11*. 2010. URL: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[22] Jiachen Xue. *RDMA Tutorial*. 2017. URL: https://github.com/jcxue/RDMA-Tutorial/wiki.