Implementation and Analysis of an InfiniBand based Communication in a Real-Time Co-Simulation Framework

Dennis Potter
Matriculation Number: 315248

Master Thesis

The present work was submitted to
RWTH Aachen University
Faculty of Electrical Engineering and Information Technology
Institute for Automation of Complex Power Systems
Univ.-Prof. Antonello Monti, Ph. D.

Supervisors: Lukas Razik
Steffen Vogel

The present's work LaTeX source code, its images, all used datasets, and all scripts that were used to create graphs are publicly available on https://git.dennispotter.eu/Dennis/masters-thesis. The aforementioned Git repository also provides a compiled version of the present work.

Author Dennis Potter <dennis@dennispotter.eu>

Supervisors Lukas Razik <lrazik@eonerc.rwth-aachen.de>
Steffen Vogel <svogel2@eonerc.rwth-aachen.de>

Submitted to RWTH Aachen University on November 9, 2018.

Copyright

Abstract

The present work evaluates the feasibility and added value of an InfiniBand based communication in the co-simulation framework VILLASframework and its simulation data gateway VILLASnode. InfiniBand is characterized by its high throughput and low latencies, which makes it particularly suitable for the hard real-time requirements of VILLASnode. It allows applications on different host systems to communicate with each other, without many of the latency bottlenecks that are present in other technologies such as Ethernet.

The present work shows that—with some optimizations—sub-microsecond latencies were achievable in a benchmark that mimics the characteristics of the co-simulation framework. After it presents how InfiniBand was integrated in the framework, thereby only making minor adjustments to the existing communication API, it shows how the newly implemented interface performs compared to the existing ones.

The results showed that, regarding latency, the InfiniBand interface performed more than one order of magnitude better than VILLASnode's other interfaces that enable server-server communication. Furthermore, much higher transmission rates could be achieved and the latency's predictability substantially improved. Its latencies, which lie between 1.7 µs and 4.9 µs, were only 1.5–2.5 µs worse than the zero-latency reference, in which VILLASnode uses the *POSIX shared memory* API to communicate. However, since the shared memory interface is only supported when the different VILLASnode instances are located on the same computer, the InfiniBand interface turned out to have the lowest latency of the currently implemented server-server interfaces.

Keywords: InfiniBand, co-simulation, RDMA, real-time, VILLASframework, VILLASnode, OFED, OpenFabrics, HPC

Contents

Acronyms

ABR Adjusted Block Received. 41, 42

AETH ACK Extended Transport Header. 34

AH Address Handle. 53, 88, 91, 127, 129, 142–144

API Application Programming Interface. v, 49, 96, 97, 101, 129, 141

ARP Address Resolution Protocol. 56

ASPM Active State Power Management. 111, 131

AtomicAckETH Atomic ACK Extended Transport Header. 34

AtomicETH Atomic Extended Transport Header. 33

BECN Backward Explicit Congestion Notification. 42, 43

BTH Base Transport Header. 33, 42

CA Channel Adapter. 22, 30, 34, 38, 106

CC Completion Channel. 54, 142, 143, 146, 147

CCA Congestion Control Architecture. 42, 43, 169

CCM Congestion Control Manager. 42, 43, 169

CCT Congestion Control Table. 43, 169

CI/CD continuous integration and continuous delivery. 101

CM Communication Manager. 46, 49, 56, 81, 100, 120–122, 125, 129, 147, 148, 150, 174

CPU central processing unit. 11, 15, 52, 54, 57, 60–65, 91, 97, 103–105, 108, 109, 111, 124, 175

CQ Completion Queue. 17, 46, 54, 55, 84, 88, 93, 106, 108, 109, 112, 142, 143, 145, 146, 175

CQE Completion Queue Entry. 25, 26, 51–55, 75, 81, 84, 90, 91, 93, 106, 116, 144–146, 161, 171–175

CRC Cyclic Redundancy Check. 32, 34

CSV comma-separated values. 94

DETH Datagram Extended Transport Header. 33

DMA Direct Memory Access. 11, 26, 38, 52, 111, 112

DPWRR dual priority weighted round robin. 39

DREP response to DREQ. 47, 49

DREQ request for communication release. 47, 48

DRTS digital real-time simulation. 7

ETH Extended Transport Header. 33

EUI-48 48-bit Extended Unique Identifier. 56

EUI-64 64-bit Extended Unique Identifier. 35, 37

FC packet Flow Control Packet. 40–42, 169

FCCL Flow Control Credit Limit. 42

FCTBS Flow Control Total Blocks Sent. 41

FECN Forward Explicit Congestion Notification. 42, 43

FIFO first-in, first-out. 9, 25, 67

GID Global Identifier. 30, 35–37, 47, 56, 141, 143, 145, 169

GMP General Management Packet. 31, 32, 46

GPL GNU General Public License. 8, 49

GRH Global Routing Header. 32, 33, 35–38, 125, 127, 129, 169

GUID Global Unique Identifier. 31, 35, 48, 144, 145

HCA Host Channel Adapter. 22, 24–29, 43–46, 49–52, 56, 60, 61, 74, 75, 77, 81–86, 91, 93, 103, 104, 106, 111–113, 117, 118, 124–126, 131, 137, 144, 151, 170

I/O input/output. 7, 64, 131

IB InfiniBand. 19, 21, 141, 174

IBA InfiniBand Architecture. 20–24, 27, 32, 33, 35, 37, 38, 41, 45, 46, 48, 49, 52, 56, 73, 77, 81, 86, 135, 169, 173

IBTA InfiniBand℠ Trade Association. 20, 49

ICRC Invariant CRC. 34

IETH Invalidate Extended Transport Header. 34

ImmDt Immediate Data. 34, 52, 86

IPoIB Internet Protocol over InfiniBand. 56, 131

IRQ interrupt request. 64, 65, 175

iWARP Internet Wide-area RDMA Protocol. 49

JSON JavaScript Object Notation. 101, 153

LFENCE Load Fence. 98

LID Local Identifier. 29–31, 34, 47, 141, 143

lkey local key. 44

LLFC Link-Level Flow Control. 40, 41, 169

LMC LID Mask Control. 34

LRH Local Routing Header. 33–35, 38, 169

LSB least significant bit. 35, 37, 56, 58, 98

MAD Management Datagram. 30–32, 46, 169

MMU memory management unit. 57, 58

MR Memory Region. 44, 142, 145, 146, 169

MRA message receipt acknowledgment. 47, 48

MSB most significant bit. 35, 37, 56, 98

MTU Maximum Transmission Unit. 21, 30, 31, 34, 118, 119, 121, 122, 145, 174

MW Memory Window. 44, 45, 142, 169

NIC network interface controller. 8, 15–17, 144

NTP Network Time Protocol. 82

NUMA non-uniform memory access. 12, 60–63, 103, 104, 170

OFED™ OpenFabrics Enterprise Distribution. 23, 46, 49, 50, 52, 54, 56, 57, 82, 103, 115, 120, 137, 169, 173

OS operating system. 15, 16, 46, 57, 59, 87, 103, 108, 111, 131, 137

PCI-e Peripheral Component Interconnect Express. 9, 60, 61, 103, 111, 131, 170

PD Protection Domain. 44, 45, 141, 142, 146, 147

PHIL (power) hardware-in-the-loop. 7

PID process identifier. 62

PM QoS Power Management Quality of Service. 111

POSIX Portable Operating System Interface. v, 9, 129

QoS Quality of Service. 8, 10, 12, 38, 138

QP Queue Pair. 22, 23, 26–29, 32, 43–46, 51–53, 56, 57, 73, 74, 76, 77, 86, 88, 91, 100, 117, 119–122, 126, 141–149, 169, 170, 174

QPN Queue Pair Number. 29, 48, 56

RAM random-access memory. 103

RC Reliable Connection. 22, 23, 28, 29, 46, 48, 85, 88, 105, 106, 108, 109, 114, 117, 119–123, 126–128, 162, 163, 171, 174

RD Reliable Datagram. 22, 23, 46, 48, 52, 173

RDETH Reliable Datagram Extended Transport Header. 33

RDMA Remote Direct Memory Access. 11, 12, 18, 24, 29, 33, 34, 44–46, 49, 51–53, 56, 81, 85, 87, 100, 113–116, 120–122, 125, 129, 138, 141, 144, 147, 148, 150, 161, 171, 172, 174

RDTSC Read Time-Stamp Counter. 97, 98, 175

RDTSCP Read Time-Stamp Counter and Processor ID. 97, 98, 175

REJ reject. 47, 48

REP reply to REQ. 47, 48

REQ request for communication. 47, 48, 56

RETH RDMA Extended Transport Header. 33

rkey remote key. 44, 45

RoCE RDMA over Converged Ethernet. 12, 13, 138

RQ Receive Queue. 22, 25, 73–76, 78, 90, 91, 142, 145, 169, 170

RT real-time. 7, 137

RTU ready to use. 47, 48

SA Subnet Administration. 31, 32

sge scatter/gather element. 50

SIDR_REP Service ID Resolution Response. 47, 49

SIDR_REQ Service ID Resolution Request. 47, 49

SL Service Level. 38, 40

SM Subnet Manager. 29–31, 38, 169

SMA Subnet Management Agent. 31

SMP Subnet Management Packet. 31, 32, 42

SQ Send Queue. 22, 25, 28, 51, 74–76, 84, 85, 91, 92, 111, 115, 145, 169, 170, 173, 175

SRQ Shared Receive Queue. 141–143, 145, 146

TCA Target Channel Adapter. 22

TCP Transmission Control Protocol. 8, 22

TCP/IP Internet protocol suite. 15, 56

TLB translation lookaside buffer. 58, 59

TMR timer. 43, 169

TSC Time-Stamp Counter. 97–99, 123, 128, 132, 138, 164, 172, 176

UC Unreliable Connection. 22, 23, 46, 48, 81, 85, 88, 100, 105, 106, 108, 109, 111, 113, 114, 116, 117, 119, 121, 128, 161–163, 165, 172

UD Unreliable Datagram. 22, 23, 46–48, 53, 85, 88, 105, 106, 108, 109, 113, 117–119, 121, 122, 126–128, 141, 143, 162, 163, 165, 171–173

UDP User Datagram Protocol. 8, 9, 22

VCRC Variant CRC. 34

VI Virtual Interface. 16–19

VIA Virtual Interface Architecture. 15–22, 26, 72, 73, 77, 78, 86, 135, 169

VL Virtual Lane. 26, 31, 32, 37–42

WQE Work Queue Element. 22, 25, 26, 28, 38, 52, 73, 85, 90

WR Work Request. 22, 24, 25, 27–29, 45, 46, 51–53, 75, 77, 81, 83, 85, 86, 90–92, 111–113, 115, 117, 124, 126, 129, 145, 173

WRR weighted round robin. 39

XRC eXtended Reliable Connection. 141, 142, 144, 146

1 Introduction

1.1 Motivation

At present, there is an increasing shift in electric energy generation from centralized—often environmentally harmful—power plants to distributed renewable energy sources. In their paper on intelligence in future electric energy systems, Strasser et al. [Str+15] describe the new challenges which arise together with this shift to sustainable electric energy systems.

1.1.1 New challenges in power system simulations

Nowadays, *digital real-time simulations* (DRTS) are most frequently used to get accurate models of the output waveforms of electric energy systems. In *real-time* (RT) simulations, the equations of one time step in the simulation have to be solved within the corresponding time span in the actual physical world. As Faruque et al. [Far+15] describe, DRTS can be divided into two classes: *full digital* and *(power) hardware-in-the-loop* (PHIL) real-time simulations. While the former are completely modeled inside the simulator, the latter provide *input/output* (I/O) interfaces which allow the user to replace digital models with actual physical components.

Since power grids should be reflected into power models as accurate as possible, more complex grids will naturally result in more complex simulations. Hence, the shift towards distributed electric energy generation poses new challenges regarding DRTS complexity. One possible solution to counteract the arising computational bottlenecks is the distribution of simulation systems into smaller subsystems [Far+15].

As a solution to this problem, Stevic et al. [Ste+17] propose a framework which enables geographically distributed laboratories to integrate their off-the-shelf real-time digital simulators virtually, thereby also enabling RT co-simulations. Later, Mirz et al. [Mir+18] summarized other important benefits of such a system: hardware and software of various laboratories can be shared; easy knowledge exchange among research groups is facilitated and encouraged; there is no need to share confidential data since every laboratory can decide to run its own simulations and only share interface variables; laboratories without certain hardware can now, nonetheless, test algorithms on this hardware.

The following subsection presents the implementation of such a system, as presented by Vogel et al. [Vog+17]: *VILLASframework*.

1.1.2 VILLASframework: distributed real-time co-simulations

VILLASframework[1] is an open-source set of tools to enable distributed real-time simulations, published under the *GNU General Public License* (GPL) v3.0. Within VILLASframework, *VILLASnode* instances form gateways for simulation data. Table 1.1 shows the interfaces—which are called *node-types* in VILLASnode—which are currently supported. Node-types can roughly be divided into three categories: node-types that can solely communicate with node-types on the same server (*internal communication*), node-types that can communicate with node-types on different servers (*server-server communication*), and node-types that form an interface between a simulator and a server (*simulator-server communication*). An instance of a node-type is called a *node*.

Figure 1.1 shows VILLASframework with its main components: VILLASnode and *VILLASweb*. The figure shows nodes in laboratories that form gateways between software (e.g., a file on a host system) or hardware (e.g., a simulator). A node can also be connected to other nodes; these can be located on the same host system, on a different host system in the same laboratory, or on a host system in a remote laboratory. Within VILLASframework, a distinction must be made between the *soft real-time integration layer* and the *hard real-time integration layer*.

Although node-types that realize internal communication are able to achieve hard real-time, none of the node-types that connect different hosts with each other are able to do so. So far, all node-types rely on the *Transmission Control Protocol* (TCP)—e.g., *amqp* and *mqtt*—or on the *User Datagram Protocol* (UDP)—e.g., *socket*. Both protocols are part of the Internet protocol suite's transport layer and these nodes thus rely on Ethernet as networking technology.

Within Ethernet, a large portion of the latency between submitting a request to send data and actually receiving the data is caused by software overhead, switches between user and kernel space, and interrupts. For example, Larsen and Huggahalli [Lar+09] report that, on average, it takes 3 µs on their Linux system before control is actually handed to the *network interface controller* (NIC) when a host tries to send a simple ping message. For the Intel® 82571 1 GbE controller they used, these 3 µs are 72 % of the time the message spends in the sending node. Similar proportions of software and hardware latency can be seen at the receiving host. After optimizations, Larsen and Huggahalli reduced the latency of an Intel® 82598 10 GbE controller to just over 10 µs, in which software latency was still predominant.

Another issue of Ethernet is its variability [Lar+09]: real-time applications require a high predictability and thus low variability of the latency of samples. Furthermore, *Quality of Service* (QoS) support is limited in Ethernet [Rei+06]. Techniques to avoid and control congestion can become essential for networks with a high load, which can be caused, for example, by a high number of small samples due to real-time communication.

[1] https://www.fein-aachen.org/projects/villas-framework/

Table 1.1: Interfaces supported by VILLASnode as of June 2018.[2]

Section	Node Name	Description
internal communication	*file*	support for file log/replay
	shmem	POSIX shared memory interface with external processes
	loopback	internal loopback using a queued FIFO buffer
	signal	configurable signal generator for testing purposes
	stats	send communication statistics to other nodes
	test_rtt	measurement of round-trip time, packet loss, and sending rates
server-server communication	*socket*	BSD network sockets for Packet, IP, or UDP layer
	zeromq	ZeroMQ publish/subscribe messaging
	influxdb	InfluxDB time-series database
	nanomsg	nanomsg publish/subscribe messaging
	amqp	Advanced Message Queuing Protocol
	mqtt	Message Queuing Telemetry Transport
	ngsi	OMA Next Generation Services Interface 10
	websocket	send and receive samples of a WebSocket connection
simulator-server communication	*opal*	OPAL-RT asynchronous processes
	fpga	VILLASfpga PCI-e card
	comedi	interface to Comedia devices
	gtwif	RTDS GTWIF workstation interface
	iec61850-9-2	IEC 61850-9-2 Samples Values
	iec61850-8-1	IEC 61850-8-1 GOOSE Telegrams

1.1.3 Hard real-time communication between different hosts

Thus, in order to achieve hard real-time between different hosts, a different technology than Ethernet must be used. An alternative technology that is particularly suitable for this purpose is *InfiniBand*. This technology is specifically designed as a low-latency, high-throughput inter-server communication standard. Due to its design, every process assumes that it owns the network interface controller and the

[2] https://villas.fein-aachen.org/doc/node-types.html

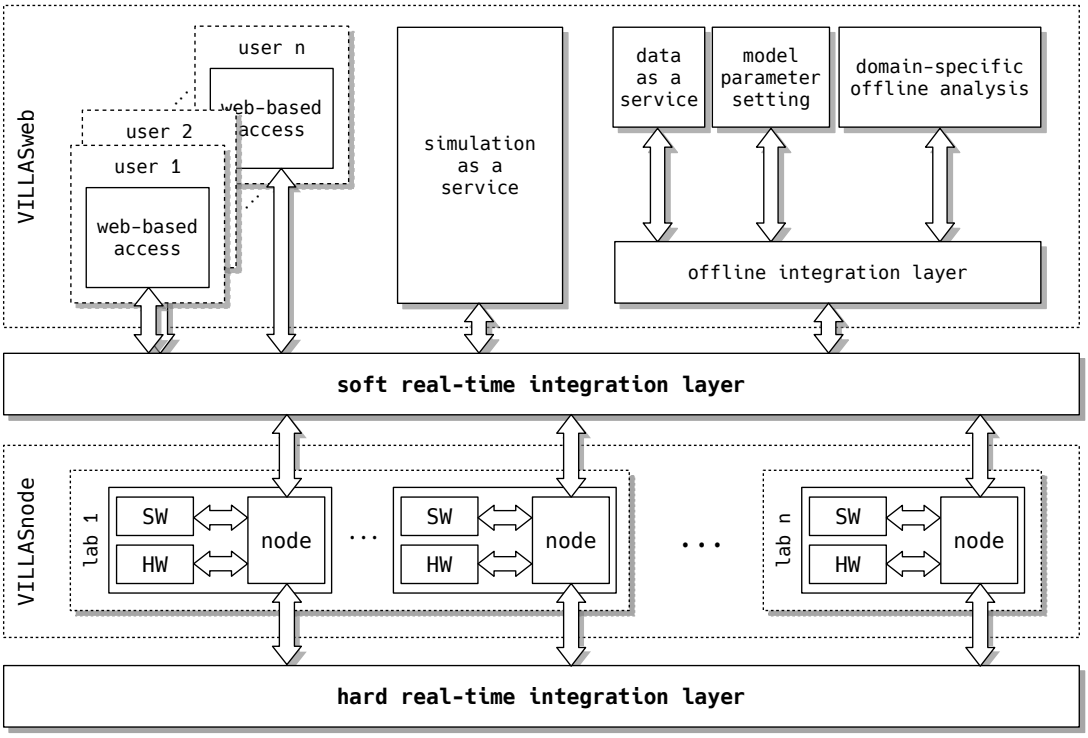


Figure 1.1: VILLASweb and VILLASnode, the main components of VILLASframework.

operating system does not need to multiplex it to processes. Consequently, processes do not need to invoke system calls—and thus trigger switches between user and kernel space—while transferring data. It is even possible to send data to a remote host without its software noticing that data is written into its memory. Furthermore, InfiniBand has extensive support for QoS and is a lossless architecture, which means that it—other than Ethernet—does not rely on dropping packets to handle congestion of the network. Finally, the InfiniBand Architecture handles many, more complex, tasks, such as reliability, directly in the hardware.

Because this technology seems so well suited for this purpose, the present work investigates the possibilities of implementing a VILLASnode node-type that relies upon InfiniBand as its communication technology.

1.2 Related work

The goal of the present work was to develop a communication channel among different host systems that is optimized regarding latency. Therefore, this section will examine previous performance studies on InfiniBand that present optimizations regarding latency.

In their work, MacArthur and Russel evaluate how certain programming decisions affect the performance of messages that are sent over an InfiniBand network [MR12]. They examine several features that potentially affect the performance:

1. The ***operation code***, which determines if a message will be sent with either channel or memory semantics.
2. The ***message size***.
3. The ***completion detection***, which determines whether the completion queue gets actively polled or provides notifications to the waiting application. This setting also heavily affects CPU utilization.
4. ***Sending data inline***, with which the CPU directly copies data to the network adapter instead of relying on the adapter's DMA.
5. ***Processing data simultaneously***, by sending data from multiple buffers instead of one.
6. Using a ***work request submission list***, with which instructions are submitted to the network adapter as a list instead of one at a time.
7. Turning ***completion signaling*** periodically on and off for certain operations.
8. The ***wire transmission speed***.

They conclude, that an application should use the operation code that best suits its needs. A limiting factor here is often the need to notify the receiver about new data. When comparing the operation codes that support notifying the receive side, i.e., *send* and *RDMA write with immediate*, the performance difference is negligible.

For "small" messages ($\leq 1024\,\text{KiB}$), the message size did not influence the latency too much under normal circumstances. For "large" messages ($\geq 1024\,\text{KiB}$), however, they observed that the latency increased with the message size.

When letting the completion queue provide notifications when new data arrived, they measured a CPU utilization of 20 % for messages smaller than 512 B and 0 % for messages larger than 4 MiB. When the queue was actively polled, the CPU utilization turned out to always be 100 %. Although completion detection with notifications was more resource friendly, they found that it, in case of small messages, resulted in latencies that were almost $4\times$ higher than when actively polling. For large messages this difference diminished. The latencies of messages larger than 16 KiB showed no difference at all anymore.

They advice to send data inline whenever this feature is supported by the network adapter that is used and when the message size is smaller than the cache line size of the adapter. They discovered that sending data inline required a few additional CPU cycles, but resulted in a latency decrease of up to 25 %. They also called attention to the fact that sending messages larger than the cache line size of the network adapter inline had a detrimental effect on latency.

With regards to the number of buffers, they found the ideal number of buffers to be around 8 for small messages and 3 for large messages. Using more buffers did not

increase the performance any more, and even resulted in slightly worse performance in some cases. By using 8 buffers and sending data inline, they detected one-way latencies as low as 300 ns. This is considerably less than the latencies for Ethernet, as reported by Larsen and Huggahalli [Lar+09].

Their recommendation regarding the submission of lists of instructions is to only use it when appropriate: whenever it is possible to submit an instruction individually, this should be the preferred method. In that way, the adapter can queue the instructions and is thus kept busy.

Last but not least, they examined the influence of completion signaling. Usually, after a message has been (successfully) sent, the sender gets notified, for example, to release the buffer. MacArthur and Russel first inspected periodic signaling, where only every $\left(\frac{n_{buffers}}{2}\right)^{th}$ message triggered a notification. They found that this usually had little effect on latency. It only had a larger effect when a list with multiple instructions was submitted to the adapter. However, when messages were sent inline, they found that it could be beneficial to disable signaling.

MacArthur and Russel also compared their InfiniBand setup with a contemporary *RDMA over Converged Ethernet* (RoCE) setup. Although they concluded that InfiniBand outperformed RoCE for large messages, they also concluded that the difference for small messages was negligible. However, as Reinemo et al. state in their publication [Rei+06], support for QoS is limited in Ethernet and abundantly available in InfiniBand.

In a later work [LR14], Liu and Russel solely focused on throughput. Although they exclusively focused on messages larger than 32 KiB, which are uncommon in VILLASnode, they drew a few conclusions that can generally be applied to communication over InfiniBand. They observed that:

- in most cases, NUMA affinity effects the performance of the network adapter;
- the performance (with regards to throughput) is sensitive to message alignment;
- the maximum number of unsignaled instructions before a signaled instruction should be sent is:

$$S = \begin{cases} \min(\frac{B}{s}, 1), & \text{if } 16\,\text{KiB} < \text{message size} < 128\,\text{KiB} \\ \min(\frac{D_{SQ}}{2}, D_{SQ} - B), & \text{otherwise} \end{cases}, \tag{1.1}$$

  with B the number of outstanding messages and D_{SQ} the depth of the send queue.

Furthermore, they preferred the *RDMA write with immediate* over the *send* operation.

1.3 Structure of the present work

2 Basics aims to give the reader an understanding of the communication architecture that lies at the heart of the VILLASnode node-type that was implemented as part of the present work. The chapter starts with an introduction on the Virtual Interface Architecture and proceeds with a section that is dedicated to InfiniBand. Before finishing with a section on real-time optimizations, chapter 2 elaborates upon the software libraries that are used to access InfiniBand hardware.

3 Architecture expands on the internals of VILLASnode. After having explained the concept of VILLASnode, this chapter discusses the adaptions that had to be made to its architecture to (efficiently) support an InfiniBand node-type. These include changes to function parameters of the interface between the global VILLASnode instance and an instance of a node-type, to the memory management of VILLASnode, and to the finite-state machine of instances of node-types.

4 Implementation first discusses the non-trivial parts of the implementation of the benchmark that was used to profile the InfiniBand hardware, the *InfiniBand* node-type, and the benchmark that was used to analyze VILLASnode node-types. Then, it discusses how an additional service type was enabled in the communication manager that was used and how the acquired data from the benchmarks was processed.

5 Evaluation evaluates the results that were found with the help of the benchmarks that were presented in the previous chapter.

6 Conclusion considers whether the assumptions from section 1.1 (Motivation) are legitimate and thus whether the *InfiniBand* node-type is a valuable addition to the VILLASframework.

7 Future Work presents possible optimizations that were not examined in the present work. It begins with a brief examination of the possibilities the `PREEMPT_RT` patch could bring, continues with a section on optimizations & profiling of the VILLASnode source code, and ends with a section on RoCE.

In addition to this brief introduction on the structure of the present work, every chapter begins with a paragraph that presents the structure of the sections within that chapter.

2 Basics

This first section of this chapter (2.1) introduces the Virtual Interface Architecture, of which the InfiniBand Architecture is a descendant. After this brief introduction on InfiniBand's origins, section 2.2 is completely devoted to the InfiniBand Architecture itself. Subsequently, section 2.3 introduces the software libraries that are used to operate the InfiniBand hardware in the present work's benchmarks and in the implementation of the VILLASnode *InfiniBand* node-type. Finally, section 2.4 goes on to discuss real-time optimizations in Linux, which is the operating system VILLASnode is most frequently operated on.

2.1 The Virtual Interface Architecture

InfiniBand is rooted in the *Virtual Interface Architecture* (VIA) [Pfi01], which was originally introduced by Compaq, Intel, and Microsoft [97]. Although InfiniBand does not completely adhere to the original VIA specifications, it is important to understand its basics. In that way, some design decisions in the InfiniBand Architecture will be more comprehensible. This section will therefore elaborate on the characteristics of the VIA.

The lion's share of the Internet protocol suite, also known as TCP/IP, is implemented by the *operating system* (OS) [Koz05]. Even though the concept of the TCP/IP stack allows the interface between a NIC and an OS to be relatively simple, a drawback is that the NIC is not directly accessible for consumer processes, but only over this stack. Since the TCP/IP stack resides in the operating system's kernel, communication operations result in *trap* machine instructions (or on more recent x86 architecture's: *sysenter* instructions), which cause the *central processing unit* (CPU) to switch from user to kernel mode [Ker10]. This back-and-forth between both modes is relatively expensive and thus adds a certain amount of latency to the communication operation that caused the switch. Furthermore, since the TCP/IP stack also includes reliability protocols and the (de)multiplexing of the NIC to processes, the operating system has to take care of these rather expensive tasks as well [Koz05]. Section 1.1 already described Larsen and Huggahalli's [Lar+09] research on the proportions of the latency in the Internet protocol suite. This overhead resulted in the need—and thus the development—of a new architecture which would provide each process with a directly accessible interface to the NIC: the Virtual Interface Architecture was born.

In their publication, Dunning et al. [Dun+98] describe that the most important characteristics of the VIA are:

- data transfers are realized through zero-copy;
- system calls are avoided whenever possible;
- the NIC is not multiplexed between processes by a driver;
- the number of instructions needed to initiate data transport is minimized;
- no interrupts are required when initiating or completing data transport;
- there is a simple set of instructions for sending and receiving data;
- it can both be mimicked in software and synthesized to hardware.

Accordingly, several tasks which are handled in software in the Internet protocol suite—e.g., multiplexing the NIC to processes, data transfer scheduling, and preferably reliability of communication—must be handled by the NIC in the VIA.

2.1.1 Basic components

A model of the VIA is depicted in Figure 2.1. At the top of the stack are the processes and applications that want to communicate over the network controller. Together with OS communication protocols and a special set of instructions which are called the *VI User Agent*, they form the *VI Consumer*. The VI consumer is colored light gray in Figure 2.1 and resides completely in the operating system's user space. The user agent provides the upper layer applications and communication protocols with an interface to the *VI Provider* and a direct interface to the *Virtual Interfaces* (VIs).

The VI provider, colored dark gray in Figure 2.1, is responsible for the instantiation of the virtual interfaces and completion queues, and consists of the *kernel agent* and the NIC. In the VIA, the NIC implements and manages the virtual interfaces and completion queues—which will both be further elaborated upon in subsection 2.1.2—and is responsible for performing data transfers. The kernel agent is part of the operating system and is responsible for resource management, e.g., creation and destruction of VIs, management of memory used by the NIC, and interrupt management. Although communication between consumer and kernel agent requires switches between user and kernel mode, this does not influence the latency of data transfers because no data is actually transferred via this interface.

2.1.2 Data transfer

One of the most distinctive elements of the VIA, compared to the Internet protocol suite, is the Virtual Interface (VI). Because of this direct interface to the NIC, each process assumes that it owns the interface and there is no need for system calls when performing data transfers. Each virtual interface consists of a send and a receive work queue which can hold *descriptors*. These contain all information necessary to transfer data, for example, destination addresses, transfer mode to be used, and the location of data to be transferred in the main memory. Hence, both send and receive data transfers are initiated by writing a descriptor memory structure to a VI, and subsequently notifying the VI provider about the submitted structure. This

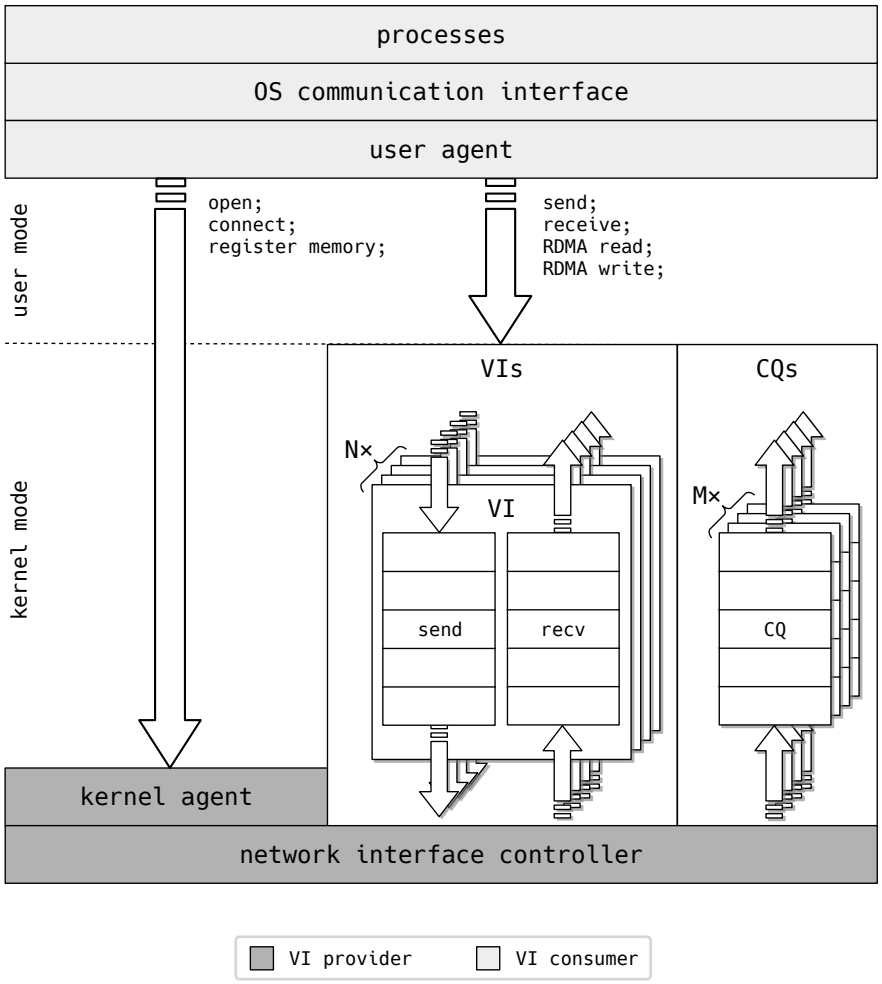


Figure 2.1: The Virtual Interface Architecture (VIA) model.

notification happens with the help of a *doorbell*, which is directly implemented in the NIC. As soon as the NIC's doorbell has been rung, it starts to asynchronously process the descriptors.

When a transfer has been completed—successfully or with an error—the descriptors are marked by the NIC. Usually, it is the consumer's responsibility to remove completed descriptors from the work queues. Alternatively, on creation, a VI can be bound to a *Completion Queue* (CQ). Then, notifications on completed transfers are directed to this queue. A CQ has to be bound to at least one work queue. This means that, on the other hand, completion notifications of several work queues can be redirected to one single completion queue. Hence, if there is an environment with N virtual interfaces with each two work queues, there can be

$$0 \leq M \leq 2 \cdot N \tag{2.1}$$

completion queues.

The Virtual Interface Architecture supports two asynchronously operating data transfer models: the *send and receive messaging* model and the *Remote Direct Memory Access* (RDMA) model. The characteristics of both models are described below.

Send and receive messaging model (channel semantics) This model is the concept behind various other popular data transfer architectures. First, a receiving node explicitly specifies where data which will be received shall be saved in its local memory. In the VIA, this is done by submitting a descriptor to the receive work queue. Subsequently, a sending node specifies the address of the data to be sent to that receiving node in its own memory. This location is then submitted to its send work queue, analogous to the procedure for the receive work queue.

Remote Direct Memory Access model (memory semantics) This approach is lesser-known. When using the RDMA model, one node, the active node, specifies both the local and the remote memory region. There are two possible operations in this model: *RDMA write* and *RDMA read*. In the former, the active node specifies a local memory region which contains data to be sent and a remote memory region to which the data shall be written. In the latter, the active node specifies a remote memory region which contains data it wants to acquire and a local memory region to which the data shall be written. To initiate an RDMA transfer, the active node has to specify the local and remote memory addresses and the operation mode in a descriptor and submit it to the send work queue. The operating system and software on the passive node are not aware of both RDMA operations. Hence, there is no need to submit descriptors to the receive work queue at the passive side.

2.1.3 The virtual interface finite-state machine

The original VIA proposal defines four states in which a virtual interface can reside: *idle*, *pending connect*, *connected*, and *error*. Transitions between states are handled by the VI provider and are invoked by the VI consumer or events on the network. The four states and all possible state transitions are depicted in the finite-state machine in Figure 2.2. A short clarification on every state is given in the list below:

- ***Idle***: A VI resides in this state after its creation and before it gets destroyed. Receive descriptors may be submitted but will not be processed. Send descriptors will immediately complete with an error.
- ***Pending connect***: An active VI can move to this state by invoking a connection request to a passive VI. A passive VI will transition to this state when it attempts to accept a connection. In both cases, it stays in this state until the connection is completely established. If the connection request times out, the connection is rejected, or if one of the VIs disconnects, the VI will return to the *idle* state. If a hardware or transport error occurs, a transition to the *error* state will be made. Descriptors which are submitted to either work queue in this state are treated in the same fashion as they are in the *idle* state.

- ***Connected***: A VI resides in this state if a connection request it has submitted has been accepted or after it has successfully accepted a connection request. The VI will transition to the *idle* state if it itself or the remote VI disconnects. It will transition to the *error* state on hardware, transport, or, dependent on the reliability level of the connection, on other connection related errors. All descriptors which have been submitted in previous states and did not result in an immediate error and all descriptors which are submitted in this state are processed.
- ***Error***: If the VI transitions to this state, all descriptors present in both work queues are marked as erroneous. The VI consumer must handle the error, transition the VI to the *idle* state, and restart the connection if desired.

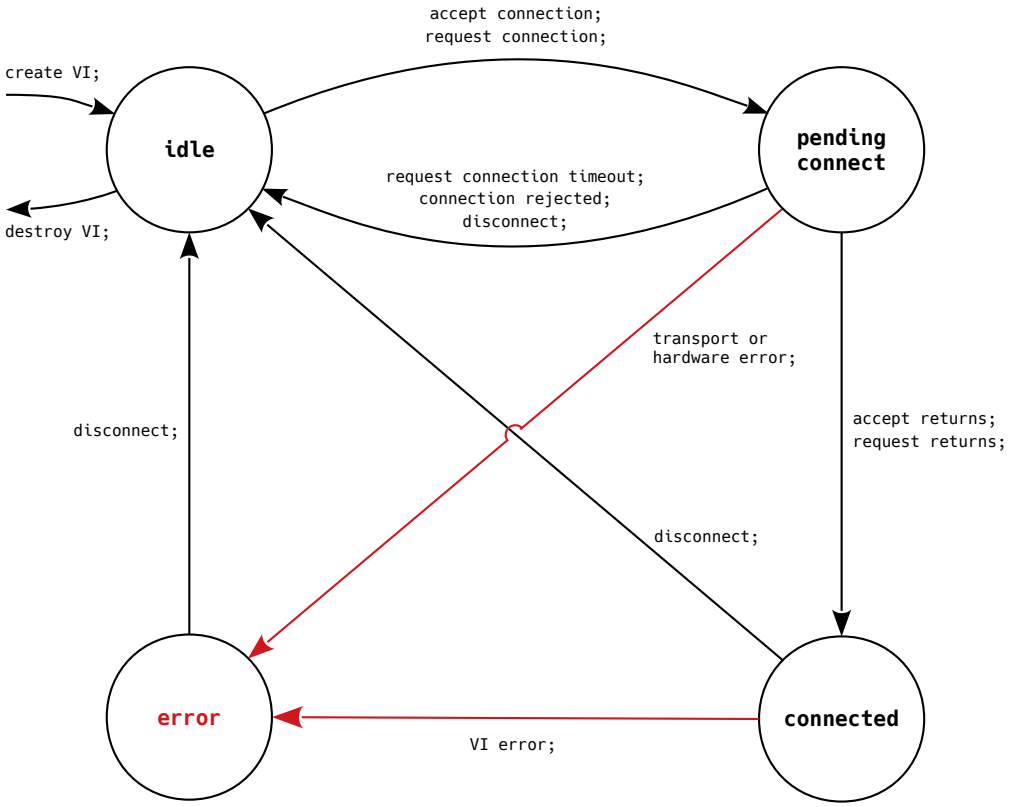


Figure 2.2: The Virtual Interface Architecture (VIA) state diagram.

2.2 The InfiniBand Architecture

After a brief introduction on the Virtual Interface Architecture in section 2.1, this section will further elaborate upon *InfiniBand* (IB). Because the VIA is an abstract

model, the purpose of the previous section was not to provide the reader with its exact specification, but rather to give him/her a general idea of the VIA design decisions. Since the exact implementation of various parts of the Virtual Interface Architecture is left open, the *InfiniBand Architecture* (IBA) does not completely correspond to the VIA. Therefore, a more comprehensive breakdown of the IBA will be given in this section.

The *InfiniBand℠ Trade Association* (IBTA) was founded by more than 180 companies in August 1999 to create a new industry standard for inter-server communication. After 14 months of work, this resulted in a collection of manuals of which the first volume describes the architecture [07] and the second the physical implementation of InfiniBand [16]. In addition, Pfister [Pfi01] wrote an excellent summary of the IBA.

2.2.1 Basics of the InfiniBand Architecture

Network stack Like most modern network technologies, the IBA can be described as a network stack, which is depicted in Figure 2.3. The stack consists of a physical, link, network, and transport layer. The IBA implementations of the different layers

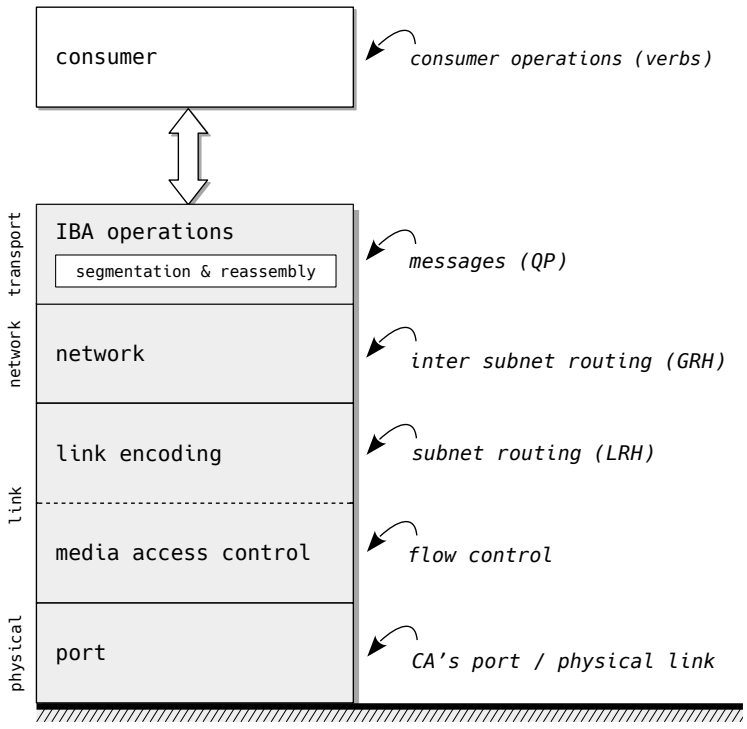


Figure 2.3: The network stack of the InfiniBand Architecture (IBA).

are displayed in the right column of Figure 2.3. Although the present work attempts to separate the different layers into different subsections, some features cannot be

explained without referring to features in other layers. Hence, the subsections do not directly correspond with the different layers.

First, this subsection gives some basic definitions for InfiniBand. It also includes some information about segmentation & reassembly of messages (although that is part of the transport layer). The main component of the transport layer, the queue pair, is presented in subsection 2.2.2. That subsection also points out some similarities and differences between the VIA and the IBA. Then, after the basics of the IBA subnet, the subnet manager, and managers in general are described in subsection 2.2.3, inner subnet routing and subnet routing will be elaborated upon in subsection 2.2.4. Subsequently, subsection 2.2.5 clarifies InfiniBand's virtual lanes and service levels. Subsection 2.2.6 and 2.2.7 go further into flow control and memory management in the IBA, respectively. Finally, subsection 2.2.8 explains how communication is established, managed, and destroyed.

An overview of the implementation of the physical link will not be given in the present work. The technical details on this can be found in the second volume of the InfiniBand™ Architecture Specification [16]. The implementation of consumer operations will be elaborated upon later, in section 2.3.

Message segmentation Communication on InfiniBand networks is divided into messages between 0 B and 2^{32} B (2 GiB) for all service types, except for unreliable datagram. The latter supports—depending on the *Maximum Transmission Unit* (MTU)—messages between 0 B and 4096 B.

Messages that are bigger than the MTU, which describes the maximum size of a packet, are segmented into smaller packets by IB hardware. The MTU can be—depending on the hardware that is used—256, 512, 1024, 2048, or 4096 B. Since segmentation and reassembly of packets is handled by hardware, the MTU should not affect performance [CDZ05]. Figure 2.4 depicts the principle of breaking a message down into packets. An exact breakdown of the composition of packets will be described in subsection 2.2.4.

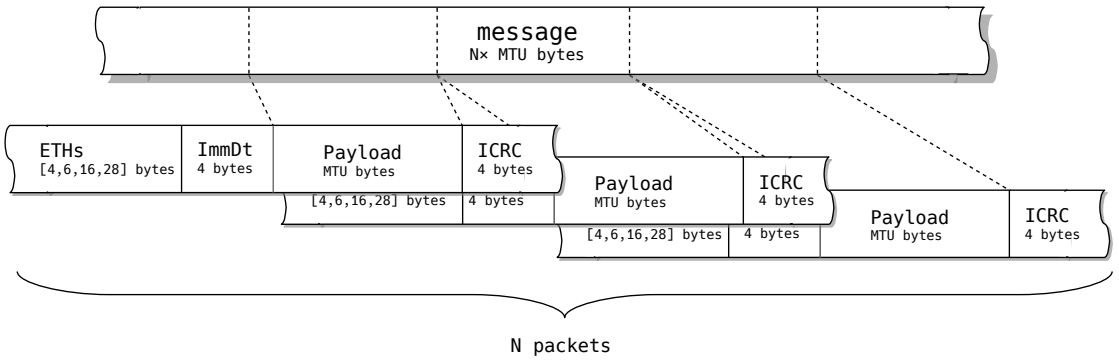


Figure 2.4: The segmentation of a message into packets.

Endnodes and channel adapters Ultimately, all communication on an InfiniBand network happens between *endnodes* (also referred to as nodes in the present work). Such an endnode could be a host computer, but also, for example, a storage system. A *Channel Adapter* (CA) forms the interface between the soft- and hardware of an endnode and the physical link which connects the endnode to a network. A channel adapter can either be a *Host Channel Adapter* (HCA) or a *Target Channel Adapter* (TCA). The former is most commonly used, and distinguishes itself from the latter by implementing so-called *verbs*. Verbs form the interface between processes on a host computer and the InfiniBand fabric; they are the implementation of the user agent from Figure 2.1.

Service types InfiniBand supports several types of communication services which are introduced in Table 2.1. Every channel adapter must implement *Unreliable Datagram* (UD), which is conceptually comparable to UDP. HCAs must implement *Reliable Connections* (RCs); this is optional for TCAs. The reliable connection is similar to TCP. Neither of the channel adapter types is required to implement *Unreliable Connections* (UCs) and *Reliable Datagram* (RD).

Table 2.1 describes the service levels on a very abstract level. More information on the implementation, for example, on the different headers which are used in IBA data packets, will be given later on. Furthermore, Table 2.1 already contains references to the abbreviation QP, which stands for queue pair and is InfiniBand's equivalent to a virtual interface (section 2.1). This will be elaborated upon in the next subsection.

2.2.2 Queue pairs & completion queues

As mentioned before, the InfiniBand Architecture is inspired by the Virtual Interface Architecture. Figure 2.5, which is derived from Figure 2.1, depicts an abstract model of the InfiniBand Architecture. In order to simplify this picture, the consumer and kernel agent are omitted. In the following, the functioning principle of this model will be explained.

Virtual interfaces are called *Queue Pairs* (QPs) in the IBA and also consists of *Send Queues* (SQs) and *Receive Queues* (RQs). They are the highest level of abstraction and enable processes to directly communicate with the HCA. After everything has been initialized, a process will perform most operations on queue pairs while communicating over an InfiniBand network.

Similarly to a descriptor in the VIA, a *Work Request* (WR) has to be submitted to the send or receive queue in order to send or receive messages. Submitting a WR results in a *Work Queue Element* (WQE) in the respective queue. Among others, a WQE holds the address to a location in the host's main memory. In case of a send WQE, this memory location contains the data to be sent to a remote host. In case of a receive WQE, the containing memory address points to the location in the main memory to which received data shall be written. Not every QP can access all memory locations; this protection is handled by specific memory management mechanisms.

Table 2.1: InfiniBand Architecture's service types.

Service Type	**Description**
Reliable Connection (RC)	In this mode, one QP on a local node is connected to one QP on a remote node. This service type ensures message delivery to—thus not consumption by—the remote node. Messages are sent in order and a combination of hardware and channel adapter software resends at communication failure.
Unreliable Connection (UC)	Like RC, this service type connects one local QP with one remote QP. It is unreliable and thus does not support acknowledgment of delivery and simply drops undelivered messages.
Unreliable Datagram (UD)	This service type allows a local QP to communicate with any other unreliable datagram QP without connecting to it. Like UC, this mode is unreliable and thus simply drops packets if they get lost. The ability to send data to another QP without connecting to it is beneficial for scalability.
Reliable Datagram (RD)	Reliable datagram enables a local QP to communicate with any other RD QP without connecting to it. Contrary to UD, this service type is reliable and thus tries to resend messages when they get lost. Since reliable datagram is not implemented in the OFED™ stack (section 2.3), hence not practically usable, it will not be further discussed in the present work.
Raw Datagram	This allows a QP to send raw datagram messages, which means that IBA specific headers are stripped from the packets. This service type can be divided into IPv6 raw datagram and EtherType raw datagram. This service type will not be further discussed in the present work.

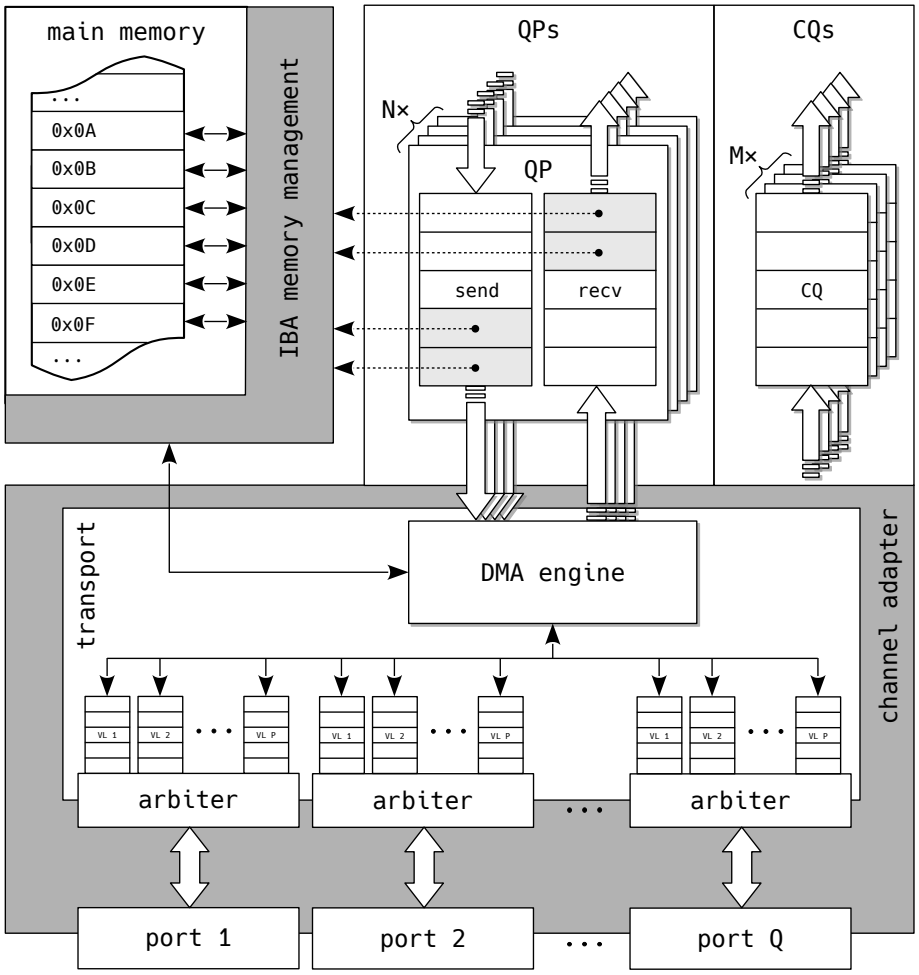


Figure 2.5: The InfiniBand Architecture (IBA) model.

These also handle which locations may be accessed by the remote hosts and by the HCA. More information on memory management can be found in subsection 2.2.7.

A work queue element in the send queue also contains the network address of the remote endnode and the transfer model, e.g., the send messaging model or an RDMA model. Except for the initialization of data transmissions, a work request can be used to bind a memory window to a memory region. This is further enlarged upon in subsection 2.2.7. A more comprehensive overview of the composition of WRs in general will be provided in section 2.3.

Example Figure 2.6 shows an example with three queue pairs in one node—in this example called *sending node*—that communicate with three queue pairs of another node—here, *receiving node*. Note that a queue pair is always initialized with a send and a receive queue; for the sake of clarity, the unused queues have been omitted in

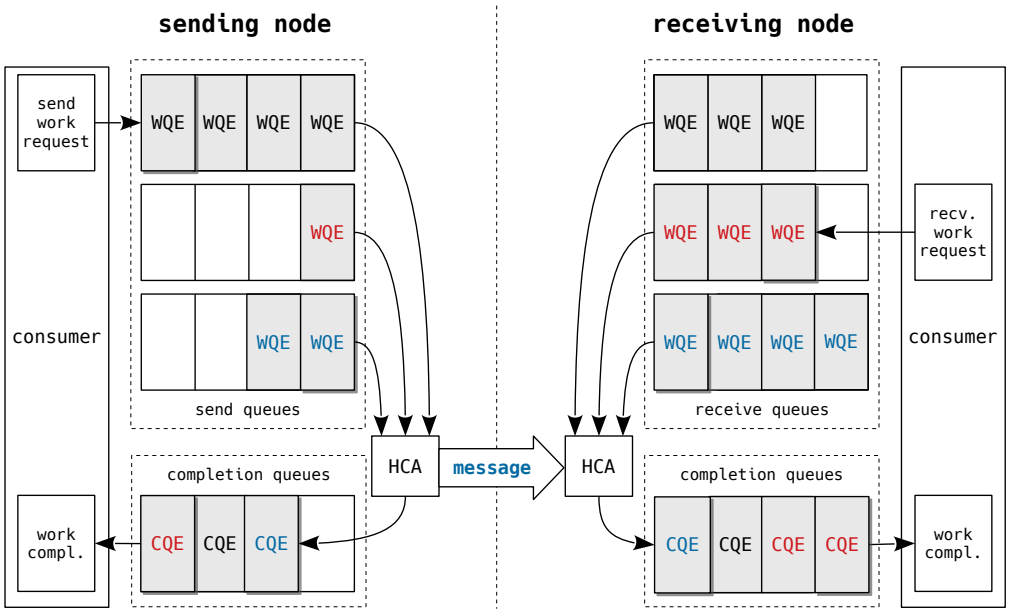


Figure 2.6: Three Send Queues (SQs) on a sending node communicate with three Receive Queues (RQs) on a receiving node. Both nodes have both a send and a receive queue, but the unused queues have been omitted for the sake of clarity.

this depiction. Hence, the image shows no receive queues for the sending node and no send queues for the receiving node.

First, before any message can be transmitted between the two nodes, the receiving node has to prepare receive WQEs by submitting receive work requests to the receive queues. Every receive WR includes a pointer to a local memory region, which provides the HCA with a memory location to save received messages to. In the picture, the consumer is submitting a WR to the red receive queue.

Secondly, send work requests may be submitted, which will then be processed by the channel adapter. Although the processing order of the queues depends on the priority of the services (subsection 2.2.5), on congestion control (subsection 2.2.6), and on the manufacturer's implementation of the HCA, WQEs in a single queue will alway obey the *first-in, first-out* (FIFO) principle. In this image, the consumer is submitting a send work request to the red send queue, and the HCA is processing a WQE from the blue send queue.

After the HCA processed a WQE, it places a *Completion Queue Entry* (CQE) in the completion queue. This entry contains, among others, information about the WQE which was processed, but also about the status of the operation. The status could indicate a successful transmission, but also an error, e.g., if not sufficient receive work queue elements were available in the receive queue. A CQE is posted

when a WQE is completely processed, so the exact moment that it is posted depends on the service type that is used. E.g., if the service type is unreliable, the WQE will be completed as soon as the channel adapter processed it and sent the data. However, if a reliable service type is used, the WQE will not complete until the message is successfully received by the remote host.

Obviously, after the message has been sent over the physical link, the receiving node's HCA will receive that same message. Then, it will acquire the destination QP from the packets' base transport headers—more on that in subsection 2.2.4—and grab the first available element from that QP's receive queue. In the case of this example, the channel adapter is consuming a WQE from the blue receive queue. After retrieving a work queue element, the HCA will read the memory address from the WQE and write the message to that memory location. When it is done doing so, it will post a completion queue entry to the completion queue. If the consumer of the sending node included immediate data in the message, that will be available in the CQE at the receive side.

Processing WQEs After a process has submitted a work request to one of the queues, the channel adapter starts processing the resulting WQE. As can be seen in Figure 2.5, an internal *Direct Memory Access* (DMA) engine will access the memory location which is included in the work queue element, and will copy the data from the host's main memory to a local buffer of the HCA. Every port of an HCA has several of these buffers which are called *Virtual Lanes* (VLs). Subsequently, separately for every port, an arbiter decides from which virtual lane packets will be sent onto the physical link. How packets are distributed among the virtual lanes and how the arbiter decides from which virtual lane to send is explained in subsection 2.2.5.

Queue pair state machine Like the virtual interfaces in section 2.1, queue pairs can reside in several states as depicted in Figure 2.7. All black lines are normal transitions and have to be explicitly initialized by a consumer with a *modify queue pair verb*. Red lines are transitions to error states, which usually happen automatically. Because this diagram is more extensive than the state machine of the VIA (Figure 2.2), the descriptions of the state transitions are omitted in this figure. All states, their characteristics, and the way to enter the state are summarized in the list below. Every list item has a sublist which provides information on how work requests, received messages, and messages to be sent are handled.

- ***Reset***: When a QP is created, it enters this state. Although this is not depicted, a transition from all other states to this state is possible.
  - Submitting **work requests** will return an immediate error.
  - **Messages that are received** by the HCA and targeted to this QP will be silently dropped.
  - No **messages are sent** from this QP.

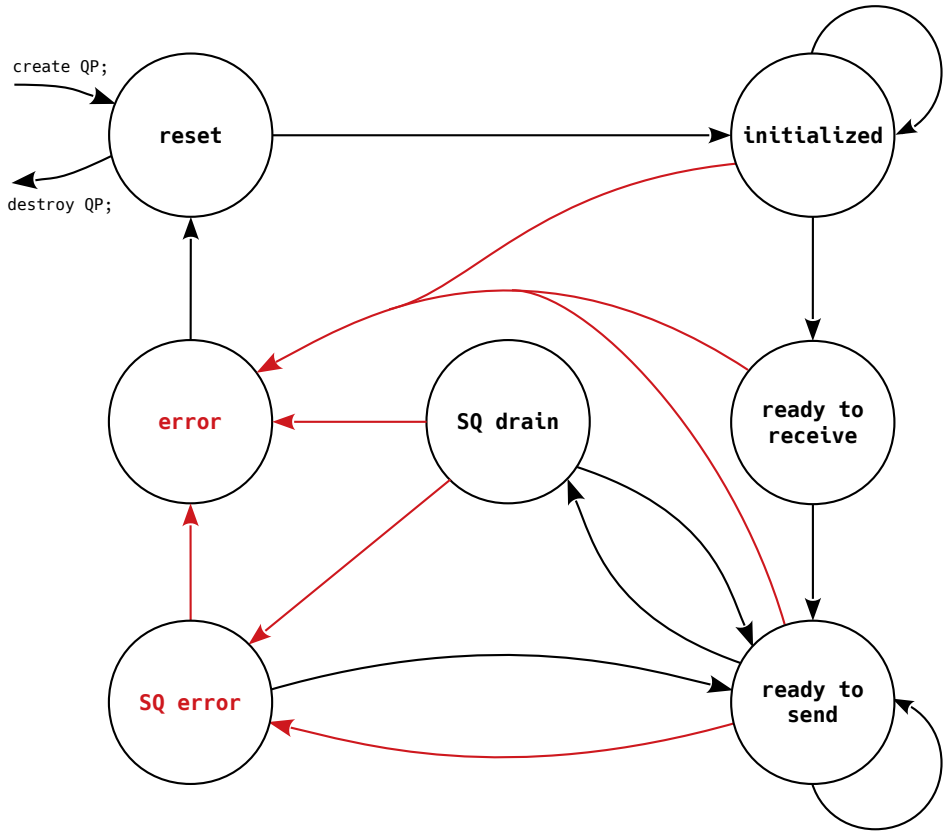


Figure 2.7: The state diagram of a Queue Pair (QP) in the InfiniBand Architecture (IBA).

- ***Initialized***: This state can be entered if the modify queue pair verb is called from the *reset* state.
  - **Work requests** may be submitted to the receive queue but they will not be processed in this state. Submitting a WR to the send queue will return an immediate error.
  - **Messages that are received** by the HCA and targeted to this QP will be silently dropped.
  - No **messages are sent** from this QP.
- ***Ready to receive***: This state can be entered if the modify queue pair verb is called from the *initialized* state. The QP can reside in this state if it only needs to receive, and thus not to send, messages.
  - **Work requests** may be submitted to the receive queue and they will be processed. Submitting a WR to the send queue will return an immediate error.

  - **Messages that are received** by the HCA and targeted to this QP will be processed as defined in the receive WQEs.
  - No **messages are sent** from this QP. The queue will respond to received packets, e.g., acknowledgments.

- ***Ready to send***: This state can be entered if the modify queue pair verb is called from the *ready to receive* or *SQ drain* state. Mostly, QPs reside in this state because the queue pair is able to receive and send messages and is thus fully operational.
  - **Work requests** may be submitted to both queues; WQEs in both queues will be processed.
  - **Messages that are received** by the HCA and targeted to this QP will be processed as defined in the receive WQEs.
  - **Messages are sent** for every WR that is submitted to the send queue.
- ***SQ drain***: This state can be entered if the modify queue pair verb is called from the *ready to send* state. This state drains the send queue, which means that all send WQEs that are present in the queue when entering the state will be processed, but all WQEs that are submitted after it entered this state will not be processed. The state has two internal states: *draining* and *drained*. While residing in the former, there are still work queue elements that are being processed. While residing in the latter, there are no more work queue elements that will be processed. When the *SQ drain* state transitions from the draining to the drained state, it generates an affiliated asynchronous event.
  - **Work requests** may be submitted to both queues. WQEs in the receive queue will be processed. WQEs in the send queue will only be processed if they were present when entering the *SQ drain* state.
  - **Messages that are received** by the HCA and targeted to this QP will be processed as defined in the receive WQEs.
  - **Messages are sent** only for WRs that were submitted before the QP entered this state.
- ***SQ error***: When a completion error occurs while the QP resides in the *ready to send* state, a transition to this state happens automatically for all QP types except the RC QP. Since an error in a WQE can cause the local or remote buffers to become undefined, all WQEs subsequent to the erroneous WQE will be flushed from the queue. The consumer can put the QP back to the *ready to send* state by calling the modify queue pair verb.
  - **Work requests** may be submitted to the receive queue and will be processed in this state. WRs that are submitted to the send queue will be flushed with an error.
  - **Messages that are received** by the HCA and targeted to this QP will be processed as defined in the receive WQEs.

  - No **messages are sent** from this QP. The queue will respond to received packets, e.g., acknowledgments.

- ***Error***: Every state may transition to the *error* state. This can happen automatically—when a send WR in an RC QP completes with an error or when a receive WR in any QP completes with an error—or explicitly—when the consumer calls the modify queue pair verb. All outstanding and newly submitted WRs will be flushed with an error.
  - **Work requests** to both queues will be flushed immediately with an error.
  - **Packets that are received** by the HCA and targeted to this QP will be silently dropped.
  - No **packets are sent**.

State transitions that are marked with black lines, which must be explicitly invoked by the consumer, will not succeed if the wrong arguments are passed to the modify queue pair verb. The first volume of the InfiniBand™ Architecture Specification [07] provides a list of all state transitions and the required and optional attributes that can be passed on to the verb. The present work will not provide the complete list of all transitions with their attributes, and will in the following only provide some examples of important states.

Queue pairs are not immediately ready to establish a connection after they have been initialized to the *reset* state. To perform the transition *reset* → *initialized*, the partition key index and, in case of unconnected service types, the queue key has to be provided. Furthermore, RDMA and atomic operations have to be enabled or disabled in this transition. A second important transition is *initialized* → *ready to receive* because here, in case of a connected service, the QP will connect to another QP. The consumer has to provide the modify QP verb with, among others, the remote node address vector and the destination *Queue Pair Number* (QPN) before it can perform the transition. If the QP must operate in loopback mode, this has to be defined here as well.

2.2.3 The InfiniBand Architecture subnet

The smallest entity in the InfiniBand Architecture is a *subnet*. It is defined as a network of at least two endnodes, connected by physical links and optionally connected by one or more switches. Every subnet is managed by a *Subnet Manager* (SM).

One task of switches is to route packets from their source to their destination, based on the packet's *Local Identifier* (LID) (subsection 2.2.4). The local identifier is a 16-bit wide address of which 48K values can be used to address endnodes in the subnet and 12K addresses are reserved for multicast. Switches support multiple service levels on several virtual lanes, which will be elaborated upon in subsection 2.2.5.

It is possible to route between different subnets with a 128-bit long *Global Identifier* (GID) (subsection 2.2.4).

Subnet manager In order for endnodes on a subnet to communicate properly with each other and for the operation of the subnet to be guaranteed, at least one managing entity has to be present to coordinate the network. Such an entity is called Subnet Manager (SM) and can either be located on an endnode, a switch, or a router. Tasks of the SM are:

- discovering the topology of the subnet (e.g., information about switches and nodes, including, for example, the MTU);
- assigning LIDs to CAs;
- establishing possible paths and loading switches' routing tables;
- regularly scanning the network for (topology) changes.

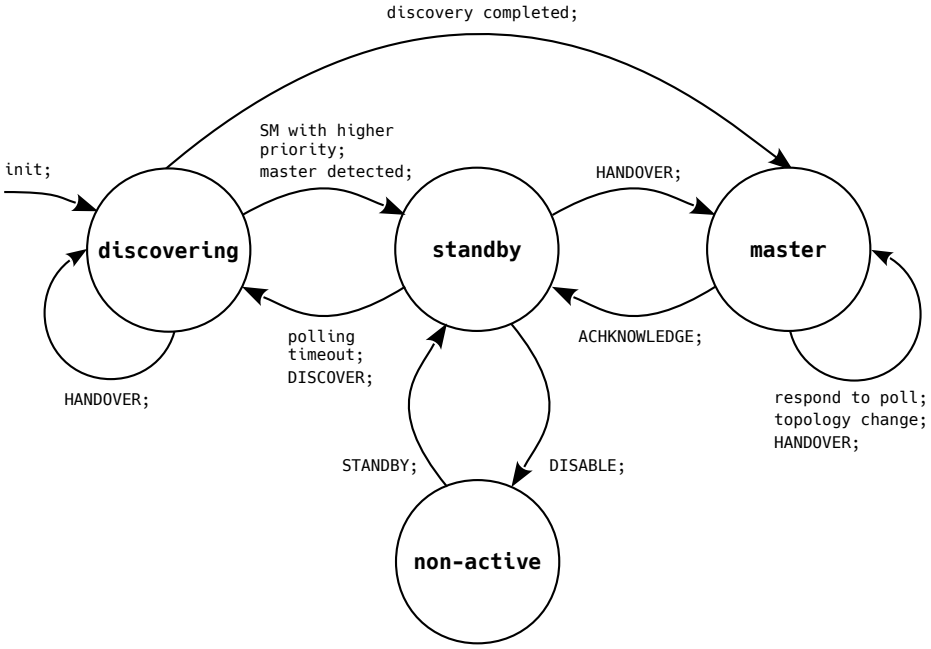


Figure 2.8: The state machine for the initialization of a Subnet Manager (SM). *AttributeModifiers* from the Management Datagram (MAD) header (Figure 2.9) are completely written in capital letters.

A subnet can contain more than one manager but only one of them may be the *master* SM. All others must be in standby mode. Figure 2.8 depicts the state machine a subnet manager goes through to identify whether it should be master or not. An SM starts in the *discovering* state in which it scans the network. As soon as it discovers another SM with a higher priority, it transitions into *standby*

mode in which it keeps polling the newly found manager. If the polled manager fails to respond (*polling time-out*), the SM goes back to the *discovering* state. If the node completes the discovery without finding a master or a manager with a higher priority, it transitions into the *master* state and starts to initialize the subnet. A master can put other SMs which are currently in standby mode and have a lower priority in the *non-active* mode by sending a *DISABLE* datagram. If it detects an SM in standby mode with a higher priority, it will exchange the mastership. To do so, it will send a *HANDOVER* datagram, which will transition the newly found SM into the *master* state. If that SM responds with an *ACKNOWLEDGE* datagram, the old master will move to the *standby* state.

Subnet management agents Every endnode has to contain a passive acting *Subnet Management Agent* (SMA). Although agents can send a trap to the SM—for example if the *Global Unique Identifier* (GUID) changes at runtime—they usually only respond to messages from the manager. Messages from the SM to an SMA can, for example, include the endnode's LID or the location to send traps to.

Subnet administration Besides SMs and SMAs, the subnet also contains a *Subnet Administration* (SA). The SA is closely connected to the SM and often even a part of it. Through *subnet administration class management datagrams*, endnodes can request information to operate on the network from the administrator. This information can, for example, contain data on paths, but also non-algorithmic data such as *service level to virtual lane mappings*.

Management datagrams *Management Datagrams* (MADs) are used to communicate management instructions. They are always 256 B—the exact size of the minimal MTU—and are divided into several subclasses. There are two types of MADs: one for general services and subnet administration, and one for subnet management. The subnet management MAD is used for communication between managers and agents, and is also referred to as *Subnet Management Packet* (SMP). The subnet administration MAD is used to receive from and send to the subnet administration, and falls under the category of *General Management Packets* (GMPs). Other than the SA, general services like performance management, baseboard management, device management, SNMP tunneling, communication management (subsection 2.2.8), and some vendor and application specific protocols make use of GMPs.

Figure 2.9 shows the management datagram base format. It is made up of a common header (between byte 0 and 23) which is used by all management packets; both SMPs and GMPs use this header. The header is followed by a 232 B data field which is different for every management datagram class.

SMPs have some particular characteristics. To ensure their transmission, VL_{15} is exclusively reserved for SMPs. This lane is not subjected to flow control restriction (subsection 2.2.6) and it is passed through the subnet ahead of all other virtual lanes. Furthermore, SMPs can make use of directed routing, which means that the

bytes \ bits	31 - 24	23 - 16	15 - 8	7 - 0
0 - 3	BaseVersion	MgmtClass	ClassVersion	R | Method
4 - 7	Status		ClassSpecific	
8 - 11	TransactionID [63,32]			
12 - 15	TransactionID [31, 0]			
16 - 19	AttributeID			
20 - 23	AttributeModifier			
24 - 27	Data [255,244]			
...	...			
252 - 255	Data [31, 0]			

Figure 2.9: The composition of a Management Datagram (MAD). The first 24 B are reserved for the common MAD header. The header is followed by up to 232 B of MAD class specific data.

ports of the switch it should exit can be defined instead of a local identifier. SMPs are always received on QP_0.

Usually, GMPs may use any virtual lane but VL_{15} and any queue pair, but this is different for SA MADs. Although they can use any virtual lane but VL_{15}, they have to be sent to QP_1.

2.2.4 Data packet format & addressing

Figure 2.10 shows the composition of a complete InfiniBand data packet. Blocks with a dashed border are optional—e.g., the *Global Routing Header* (GRH) is not necessary if the packet does not leave the subnet from which it originated—and blocks with continuous borders are mandatory—e.g., the *Cyclic Redundancy Checks* (CRCs) have to be computed for every packet.

In order to send data to non-IBA subnets, the architecture supports raw packets in which the InfiniBand specific transport headers and the invariant CRC are omitted. The present work will not go into detail on raw packets; more information on these packets can be found in the IBA specification [07].

Important information about the different kinds of transport headers, immediate data, the payload, and the two kinds of CRCs can be found in Table 2.2. Because of their importance, information on the local and global routing header will be given in a separate section below.

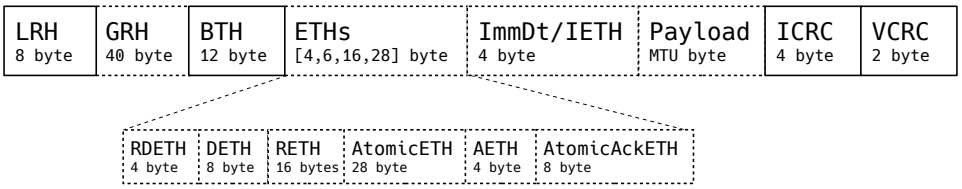


Figure 2.10: The composition of a complete packet in the InfiniBand Architecture (IBA).

Table 2.2: Explanation of abbreviations from Figure 2.10. More details on the content of the different packets can be found in the IBA specification [07].

Abbreviation	**Description**
LRH	*Local Routing Header*: detailed information on this header is provided in a separate paragraph below.
GRH	*Global Routing Header*: detailed information on this header is provided in a separate paragraph below.
BTH	*Base Transport Header*: Every packet in the IBA contains this header. It contains fields for the IBA transport and holds, i.a., the packet type, the destination queue pair number, and the packet sequence number.
ETHs	*Extended Transport Headers*: These headers are optional and are used if applicable, based on the packet type in the BTH. All following headers which end with *ETH* are extended transport headers.
RDETH	*Reliable Datagram Extended Transport Header*: This header contains the end-to-end context, used with the reliable datagram service type.
DETH	*Datagram Extended Transport Header*: This header contains the queue key and the source queue pair number for datagram transfers.
RETH	*RDMA Extended Transport Header*: This header contains the virtual address, remote key, and DMA length for an RDMA operation.
AtomicETH	*Atomic Extended Transport Header*: This header is used for atomic operations and is similar to the RETH. Instead of a length field, it contains a swap (or add) field and a compare data field.

AETH	*ACK Extended Transport Header*: This header serves as acknowledge field in RDMA read response first, RDMA read response last, RDMA read response only, and acknowledge packets.
AtomicAckETH	*Atomic ACK Extended Transport Header*: This header is similar to the AETH, but for atomic acknowledgments. It only contains the original remote data.
ImmDt	*Immediate Data*: This optional block can be used to add 32 bit of custom data to *send* or *RDMA write* packets. The containing 32 bit of data will be visible in the receive completion queue element.
IETH	*Invalidate Extended Transport Header*: This header contains a remote key which will be used to invalidate a remote memory region.
Payload	*Payload*: This is the actual payload to be sent. This field will be as big as the MTU of the network.
ICRC	*Invariant CRC*: This is the redundancy check for blocks that do not change during their transmission from source to destination. The CRC which is used is the same as in the Ethernet standard: the CRC-32 with the polynomial `0x04C11DB7`.
VCRC	*Variant CRC*: This is the redundancy check for blocks that do change during their transmissing from source to destination.

Local routing header The *Local Routing Header* (LRH) contains all necessary information for a packet to be correctly passed on within a subnet. Figure 2.11 depicts the composition of the LRH.

The most crucial fields of this header are the 16-bit source and destination *local identifier* fields. A channel adapter's port can be uniquely identified within a subnet by its LID, which the subnet manager assigns to every port in the subnet. Besides with an identifier, the subnet manager also provides CAs with an *LID Mask Control* (LMC). This value, which can range from 0 to 7, indicates how many low order bits of the LID can be ignored by the CA in order to determine if a received packet is targeted to that CA. These bits are also called *don't care bits*, and switches do not ignore them; this results in up to 128 different paths to a port in a subnet, which is a large benefit. Consequently, with this mask, it is possible to reach one single port with up to 128 different unicast LIDs. As mentioned earlier, the 16-bit LID can hold approximately 48K unicast entries and 16K multicast entries.

The 11-bit *packet length* field indicates the length of the complete packet in 4-byte words. This not only includes the length of the payload, but also of all headers. The *VL* and *SL* fields indicate which virtual lane and service level are used, respectively. Later, in subsection 2.2.5, virtual lanes, service levels, and their connection will be explained in more detail.

The 4-bit *LVer* field indicates which link level protocol is used. *LNH* stands for *Link Next Header* and this 2-bit field indicates the header that follows the mandatory local routing header. The LNH's *most significant bit* (MSB) indicates if the packet uses IBA transport or raw transport. The second bit indicates if an optional GRH is present.

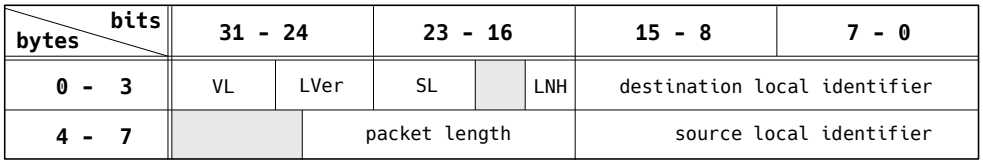


Figure 2.11: The composition of the Local Routing Header (LRH).

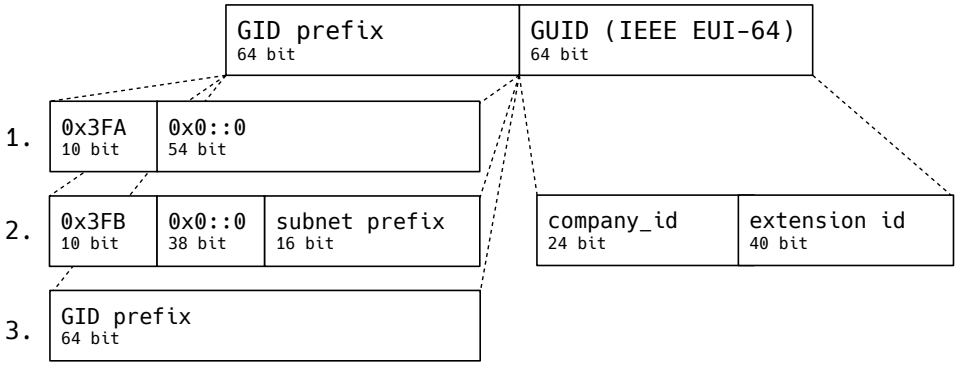
Global routing header The Global Routing Header (GRH) contains all necessary information for a packet to be correctly passed on by a router between subnets. Figure 2.12 depicts the composition of the GRH.

The most crucial fields of this header are the 128-bit source and destination *global identifier* fields. Figure 2.13 shows the possible compositions of a GID. Figure 2.13a shows the composition of the unicast GID; it consists of a GID prefix—more on that later—and a GUID. The GUID is an IEEE *64-bit Extended Unique Identifier* (EUI-64) and uniquely identifies each element in a subnet [17]. The GUID is always present in a unicast global identifier. The 24 MSBs of the GUID are reserved for the company identifier, which is assigned by the IEEE Registration Authority. The 40 *least significant bits* (LSBs) are assigned to a device by the said company, to uniquely identify it. The subnet manager may change the GUID if the scope is set to local, more on that below.

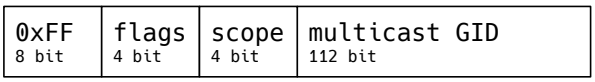
The composition of the 64-bit prefix depends on the scope in which packets will be sent. It all comes down to three cases, which are listed below. The enumeration of the list below corresponds to the enumeration in Figure 2.13a. Each port will have at least one unicast GID, which is referred to as *GID index 0*. This GID can be created using the first or the second option from the list below. Both options are based on the default GID prefix `0xFE80::0`. Packets that are constructed using the default GID prefix and a valid GUID must always be accepted by an endnode, but must never be forwarded by a router. That means that packets with only a GID index 0 are always restricted to the local subnet.

bytes \ bits	31 - 24	23 - 16	15 - 8	7 - 0
0 - 3	IPVer	TClass	flow label	
4 - 7	PayLen		NxtHdr	HopLmt
8 - 11	source global identifier [127,96]			
12 - 15	source global identifier [95,64]			
16 - 19	source global identifier [63,32]			
20 - 23	source global identifier [31, 0]			
24 - 27	destination global identifier [127,96]			
28 - 31	destination global identifier [95,64]			
32 - 35	destination global identifier [63,32]			
36 - 39	destination global identifier [31, 0]			

Figure 2.12: The composition of the Global Routing Header (GRH).



(a) The three possible compositions of a unicast GID.



(b) The composition of a multicast GID.

Figure 2.13: The possible structures of Global Identifiers (GIDs).

1. **Link-local**: The global identifier only consists of the default GID prefix `0XFE80::0` and the device's EUI-64 and is only unique within the local subnet. Routers will not forward packets with this global identifier. `0x3FA` in Figure 2.13a is another representation of the default GID prefix:

$$\texttt{0x3FA} = (\texttt{0xFE8} \gg 2). \tag{2.2}$$

   It is used to clarify the extra bit which has to be set in the second option of this list. The two LSBs of `0xFE8` which are eliminated by the right shift are zero and are absorbed by the 54-bit `0x0::0` block.

2. **Site-local**: The global identifier consists of the default GID prefix with the 54th bit of the GID prefix set to `'1'`. In the representation of Figure 2.13a, this corresponds to:

$$\texttt{0x3FB} = (\texttt{0xFE8} \gg 2) + 1 = \texttt{0x3FA} + 1. \tag{2.3}$$

   The 16-bit *subnet prefix* is set to a value chosen by the subnet manager. This GID is unique in a collection of connected subnets, but not necessarily globally.

3. **Global**: This is the only GID type which is forwarded by routers, since it is guaranteed to be globally unique.

Multicast GIDs, as depicted in Figure 2.13b, are fundamentally different from unicast GIDs. To indicate that it is a multicast packet, the 8 MSBs are all set to `'1'`. The LSB of the *flags* field indicates whether it is a permanently assigned multicast GID (`'0'`) or not (`'1'`). The remaining three bits of the flags block are always `'0'`. The 4-bit *scope* field indicates the scope of the packet. E.g., if scope equals `0x2`, a packet will be link-local and if scope equals `0xE`, a packet will be global. The complete multicast address scope is described in the IBA specification [07]. The 122 LSBs are reserved for the actual multicast GID.

Although the source and destination identifiers account for 80 % of the global routing header (Figure 2.12), there are some other fields. The 4-bit *IPVer* field indicates the version of the header and the 8-bit *TClass* field indicates the global service level, which will be elaborated upon in subsection 2.2.5. The 20-bit *flow label* field helps to identify groups of packets that must be delivered in order. The 8-bit *NxtHdr* field identifies the header which follows the GRH in the IBA packet. This is, in case of a normal IBA packet, the IBA transport header. The only remaining block is the 8-bit *HopLmt*, which limits the number of hops a packet can make between subnets, before being dropped.

2.2.5 Virtual lanes & service levels

Virtual Lanes (VLs) are independent sets of receive and transmit packet buffers. A channel adapter can be seen as a collection of multiple logical fabrics—lanes—which share a port and physical link.

As introduced in subsection 2.2.1 and in particular in Figure 2.4, after a WQE appears in the send queue, the channel adapter segments the message (i.e., the data the WQE points to) into smaller chunks of data and forms IBA packets, based on the information present in the WQE. Subsequently, a DMA engine copies them to a virtual lane.

Every switch and channel adapter must implement VL_{15} because it is used for subnet management packets (subsection 2.2.3). Furthermore, between 1 and 15 additional virtual lanes $VL_{0...14}$ must be implemented for data transmission. The actual number of VLs that is used by a port (1, 2, 4, 8, or 15) is determined by the subnet manager. Until the SM has determined how many VLs are supported on both ends of a connection and until it has programmed the port's SL to VL mapping table, the mandatory data lane VL_0 is used.

To understand QoS in InfiniBand, which signifies the ability of a network technology to prioritize selected traffic, it is essential to understand how packets are scheduled onto the VLs. Crupnicoff, Das, and Zahvai [CDZ05] did a great deal of describing the functioning of QoS in the IBA. This section will first explain how packets are scheduled onto the VLs. Then, it will describe how the virtual lanes are arbitrated between the actual physical link that is connected to the channel adapter's port.

Scheduling packets onto virtual lanes The IBA defines 16 *Service Levels* (SLs). The 4-bit field that represents the SL is present in the local routing header (Figure 2.11) and stays constant during the packet's path through the subnet. The SL depends on the service type which is used (Table 2.1). The first volume of the IBA specification [07] describes how the level is acquired for the different types. Besides the SL field, there is also the VL field in the LRH. This is set to the virtual lane the packet is sent from, and, as will be discussed below, may change during its path through the subnet.

Although the architecture does not specify a relationship between certain SLs and forwarding behavior—this is left open as a fabric administration policy—there is a specification for SL to VL mapping in switches. If a packet arrives in a switch, the switch may, based on a programmable *SLtoVLMappingTable*, change the lane the packet is on. This also changes the corresponding field in the LRH. It may happen that a packet on a certain VL passes a packet on another VL while transitioning through a switch. Service level to virtual lane mapping in switches allows, among others, interoperability between CAs with different numbers of lanes.

There is a similar mechanism to service levels for global routing: the *traffic class* (TClass) field in the GRH (Figure 2.12). The present work will not further elaborate upon traffic classes.

Arbitrating the virtual lanes The arbitration of virtual lanes to an output port has yet to be discussed. Figure 2.14 depicts the logic in the arbiters which were previously depicted in Figure 2.5 as a black box. The arbitration is implemented

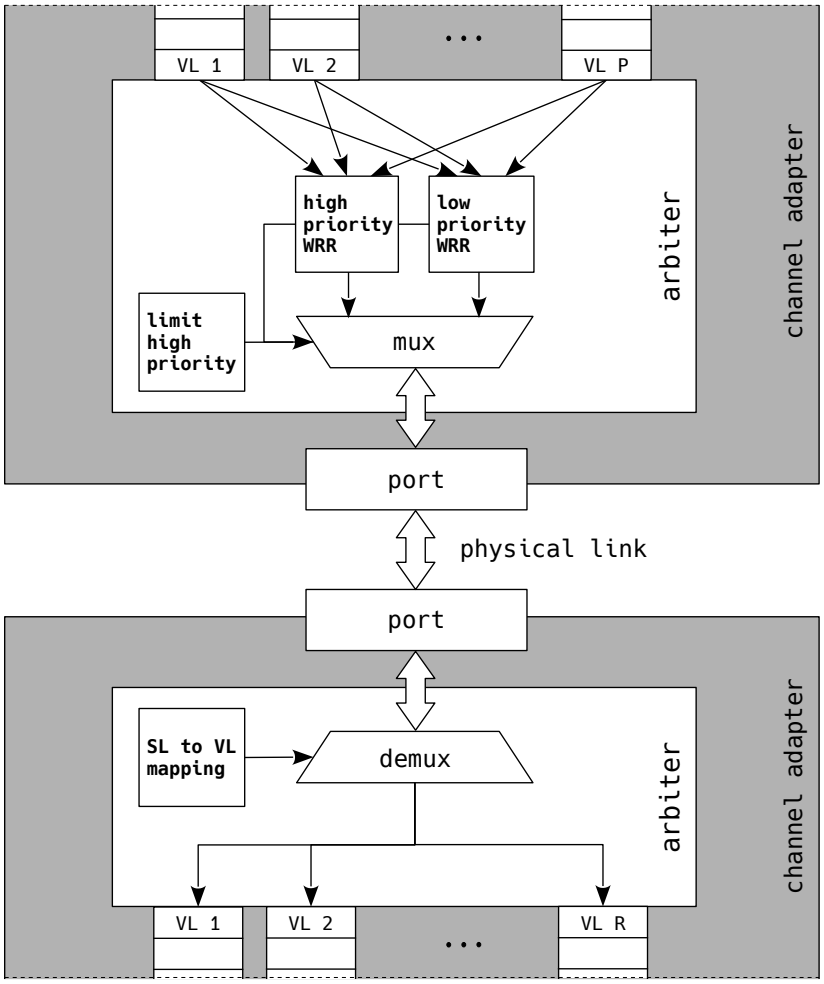


Figure 2.14: Functional principle of the arbiter.

as a *dual priority weighted round robin* (DPWRR) scheme. It consists of a *high priority-WRR* table, a *low priority-WRR* table, and a *limit high priority* counter. Both tables are lists with a field to indicate the index of a virtual lane and a weight with a value between 0 and 255. The counter keeps track of the number of high priority packets that were sent and whether that number exceeds a certain threshold.

If at least one entry is available in the high priority table and the counter is not exceeded, this table is active and a packet from this table will be sent. Which packet depends on the weighted round robin scheme. E.g., two lanes, VL_0 and VL_1, are listed in a table and they have a weight of 2 and 3, respectively. When the table is active, in $\frac{2}{2+3} \cdot 100\,\% = 40\,\%$ of the cases a packet from VL_0 and in $\frac{3}{2+3} \cdot 100\,\% = 60\,\%$ of the cases a packet from VL_1 will be sent.

If the counter reaches its threshold, a packet from a low priority lane will be sent and the counter is reset to 0. If the high priority table is empty, the low priority table will be checked immediately.

VL_{15} is not subjected to these rules and always has the highest priority. A VL may be listed in either one or in both tables at the same time. There may be more than one entry of the same VL in one table.

The bottom of Figure 2.14 shows how packets are distributed among virtual lanes based on their SL. This is similar to the mapping in switches, as described above. Figure 2.14 does not depict a switch and assumes a direct connection between two channel adapters.

2.2.6 Congestion control

InfiniBand is a lossless fabric, which means that congestion control does not rely on dropping packets. Packets will only be dropped during severe errors, e.g., during hardware failures. InfiniBand supports several mechanisms to deal with congestion without dropping packets. In the following, two control mechanisms will be described.

Link-level flow control The first mechanism, *Link-Level Flow Control* (LLFC), prevents the loss of packets caused by a receive buffer overflow. This is done by synchronizing the state of the receive buffer between source and target node with *Flow Control Packets* (FC packets), of which the composition is depicted in Figure 2.15. Flow control packets coexists with data packets, which were presented in subsection 2.2.4.

Flow control packets for a certain virtual lane shall be sent during the initialization of the physical link and prior to the passing of 65,536 *symbol times* since the last time such a packet was sent for that VL. A symbol time is defined as the time it takes to transmit an 8 bit data quantity onto a physical lane. If the physical link is in initialization state (referred to as *LinkInitialize* in the IBA specification [07]), *Op* shall be `'1'`in the flow control packet. If the packet is sent when the link is up and not in failure (*LinkArm* or *LinkActive*), *Op* shall be `'0'`.

bytes \ bits	31 - 24	23 - 16	15 - 8	7 - 0
0 - 3	Op	FCTBS	VL	FCCL
4 - 7	LPCRC			

Figure 2.15: The structure of a Flow Control Packet (FC packet).

The flow for a complete synchronization—from a source node with a sending queue, to a target node with a receiving queue, back to the sending queue—is described in the list below and depicted in Figure 2.16. Flow control packets are sent on a per virtual lane base; the 4-bit *VL* field is used to indicate the index of VL_i. VL_{15} is excluded from flow control.

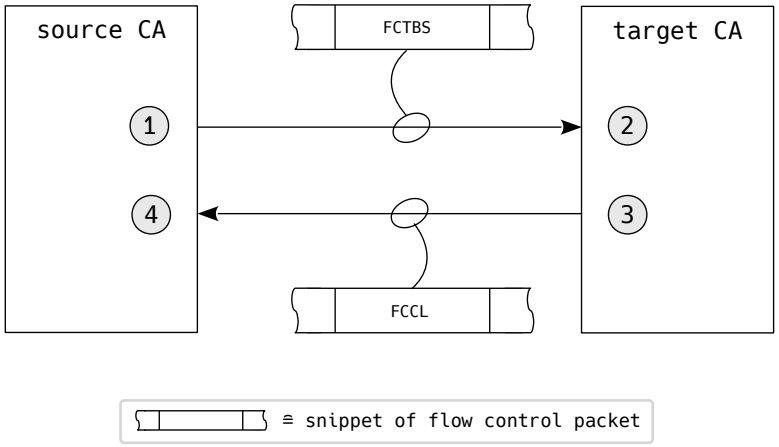


Figure 2.16: Working principle of Link-Level Flow Control (LLFC) in the InfiniBand Architecture (IBA).

1. **Set FCTBS & send FC packet**: Upon transmission of an FC packet, the 12-bit *Flow Control Total Blocks Sent* (FCTBS) field of the FC packet is set to the total number of blocks transmitted since the VL was initialized. The *block size* of a packet i is defined as

$$B_{packet,i} = \left\lceil S_i/64 \right\rceil, \tag{2.4}$$

with S_i the size of a packet, including all headers, in bytes. Hence, the total number of blocks transmitted at a certain time is defined as:

$$\mathrm{FCTBS} = B_{total} = \sum_i B_{packet,i}. \tag{2.5}$$

2. **Set and update ABR**: Upon receipt of an FC packet, a 12-bit *Adjusted Block Received* (ABR) field is set to:

$$\mathrm{ABR} = \mathrm{FCTBS}. \tag{2.6}$$

Every time a data packet is received and not discarded due to lack of receive capacity, the value is updated according to:

$$\mathrm{ABR} = \mathrm{ABR} + (B_{packet} \bmod 4096), \tag{2.7}$$

with B_{packet} the block size of the received data packet.

3. **Set FCCL & send FC packet**: Upon transmission of an FC packet, the 12-bit *Flow Control Credit Limit* (FCCL) has to be generated. If the receive buffer could permit the receipt of 2048 or more blocks of every possible combination of data packets in the current state, the credit limit is set to:

$$\text{FCCL} = \text{ABR} + 2048 \bmod 4096. \tag{2.8}$$

   Otherwise, it is set to:

$$\text{FCCL} = \text{ABR} + N_B \bmod 4096, \tag{2.9}$$

   with N_B the number of blocks the buffer could receive in the current state.

4. **Use FCCL for data packet transmission**: After a valid FCCL is received, it can be used to decide whether a data packet can be received by a remote node and thus whether it should be sent. To make this decision, a variable C is defined:

$$C = (B_{total} + B_{packet}) \bmod 4096, \tag{2.10}$$

   with B_{total} the total blocks sent since initialization and B_{packet} the block size of the packet which will potentially be transmitted. If the condition

$$(\text{FCCL} - C) \bmod 4096 \leq 2048 \tag{2.11}$$

   holds, the data packet may be sent.

Feedback based control architecture Figure 2.17 illustrates how the *Congestion Control Architecture* (CCA) works. Similar to link-level flow control, the CCA only controls data VLs; VL_{15} is excluded and thus SMPs will never be restricted.

The control consists of five steps that are listed below. The enumeration of the list below corresponds to the numbers in Figure 2.17.

1. **Detection**: The first step is the actual detection of congestion. This is done by monitoring a virtual lane of a given port and reviewing whether its throughput exceeds a certain threshold. This threshold is set by the *Congestion Control Manager* (CCM) and must always be between 0 and 15, where a value of 0 will turn off congestion control completely and a value of 15 corresponds to a very low threshold and thus aggressive congestion control on that virtual lane.

   If the threshold is reached, the *Forward Explicit Congestion Notification* (FECN) flag in the base transport header is set before the packet is forwarded to its destination.

2. **Response**: When an endnode receives a packet where the FECN flag in the BTH is set, it sends a *Backward Explicit Congestion Notification* (BECN) back to node the packet came from. In the case of connected communication (e.g., reliable connection, unreliable connection), the response might be carried in an ACK packet. If communication is unconnected (e.g., unreliable datagram) an additional *congestion notification packet* has to be sent.

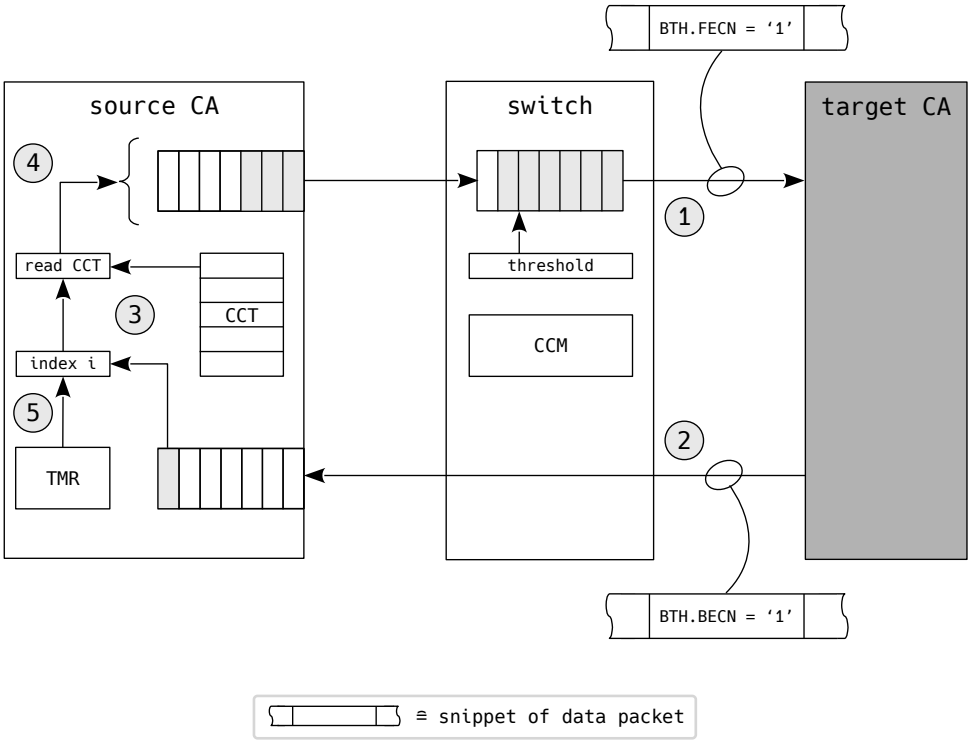


Figure 2.17: Working principle of the Congestion Control Architecture (CCA). The Congestion Control Table (CCT), timer (TMR), and threshold value are initialized by the Congestion Control Manager (CCM).

3. **Determine injection rate reduction**: When a node receives a packet with the BECN flag set, an index (illustrated as i in Figure 2.17) will be increased by a preset value. This index is used to read from the *Congestion Control Table* (CCT). This table is set by the CCM during initialization and contains inter-packet delay values. The higher the index i, the higher the delay value it points to.

4. **Set injection rate reduction**: The value from the CCT will be used to reduce the injection rate of packets onto the physical link. The reduction can either be applied to the QP that caused the packet which got an FECN flag, or to all QPs that use a particular service level (and thus virtual lane).

5. **Injection rate recovery**: After a certain time, which is set by the CCM as well, the index i, and thus also the inter-packet delay, is reduced again. If no more BECN flags are received, i and the delay will go to zero. If they do not go to zero, the card adapter probably go into an equilibrium at a certain point. In this equilibrium, the HCA will send packets with an inter-packet delay which is just above or just under the threshold that causes new FECN flags to be generated.

2.2.7 Memory management

An HCA's access to a host's main memory is managed and protected with three primary objects: *Memory Regions* (MRs), *Memory Windows* (MWs), and *Protection Domains* (PDs). The relationship between queue pairs and these objects is depicted in Figure 2.18.

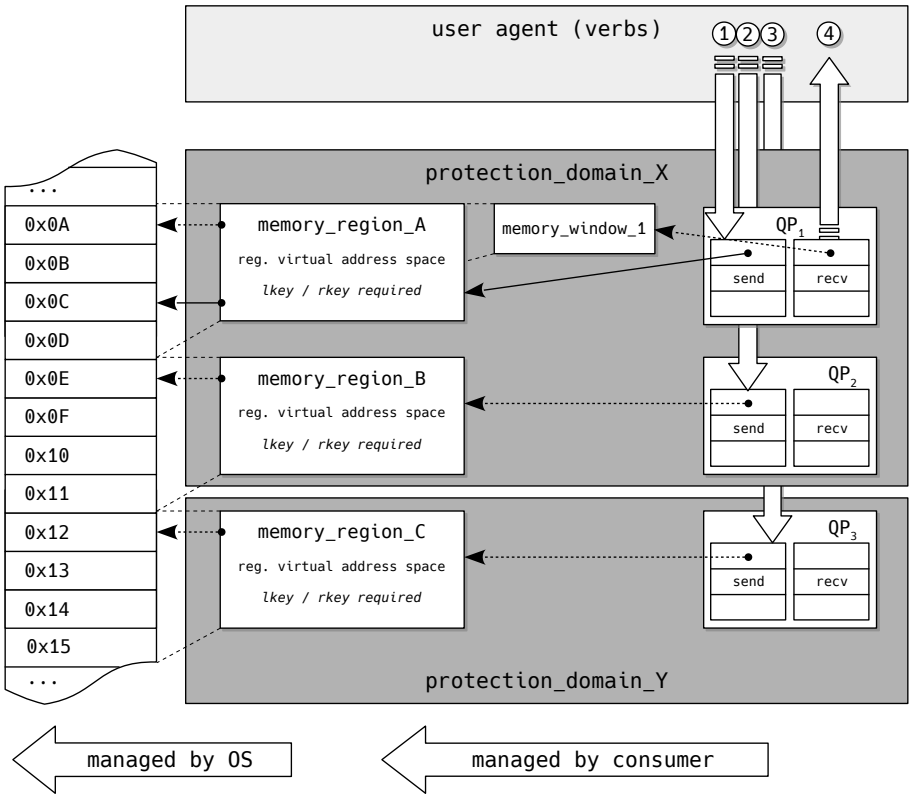


Figure 2.18: The relationship between Queue Pairs (QPs), Memory Windows (MWs), Memory Regions (MRs), and the host's main memory.

Memory regions A memory region is a registered set of memory locations. A process can register a memory region with a verb, which provides the HCA with the virtual-to-physical mapping of that region. Furthermore, it returns a *local key* (lkey) and *remote key* (rkey) to the calling process. Every time a work request which has to access a virtual address within a local memory region is submitted to a queue, the local key has to be provided within the work request. The region in the main memory is pinned on registration, which means that the operating system is prohibited from swapping that region out (subsection 2.4.1).

When a work requests tries to access a remote memory region on a target node, e.g., with an *RDMA read* or *write* operation, the remote key of the memory region

on the target host has to be provided. Hence, before an RDMA operation can be performed, the source node has to acquire the rkey of the remote memory region it wants to access. This can, for example, be done with a regular *send* operation which only requires local keys.

Protection domains Protection domains associate memory regions and queue pairs and are specific to each HCA. During creation of memory regions and queue pairs, both have to be associated with exactly one PD. Multiple memory regions and queue pairs may be part of one protection domain.

A QP, which is associated with a certain PD, cannot access a memory region in another PD. E.g., a QP in protection_domain_X in Figure 2.18 can access memory_region_A and memory_region_B, but not memory_region_C.

Memory windows If a reliable connection, unreliable connection, or a reliable datagram is used, memory windows can be used for memory management. First, memory windows are allocated, and then they are bound to a memory region. Although allocation and deallocation of a memory window requires a system call—and is thus time-consuming and not suitable for use in a datapath—binding a memory window to (a subset of) a memory region is done through a work request submitted to a send queue. A memory window can be bound to a memory region if both are situated in the same protection domain, if local write access for the memory region is enabled, and if the region was enabled for windowing at initialization.

The rkey that the MW returns on allocation is just a dummy key. Every time the window is (re)bound to (a subset of) a memory region, the rkey is regenerated. Memory windows can be thoroughly handy for dynamic management of remote access of memory. A memory window with remote rights can be bound to a memory region without remote rights, and enable remote access this way. Furthermore, remote access can be granted and revoked dynamically without using system calls.

There are two types of memory windows: Type 1 and Type 2. Whereas the former are addressed only through virtual addresses, the latter can be addressed through either virtual addresses or zero based virtual addresses. More information on the types is given in the first volume of the IBA specification [07].

Examples The list below provides some examples regarding memory regions, protection domains, and memory windows. The enumerations in the list correspond with the numbers in Figure 2.18.

1. A send work request with a pointer to `0x0C` was submitted. Since memory_region_A is bound to the address range this address lies in, the WR has to include memory_region_A's local key. This is necessary so that the HCA will be able to access the data when it starts processing the WR. A WR submitted to QP_1 can only access memory_region_A and memory_region_B—and thus only memory with addresses between `0x0A` and `0x11` in the current configuration—since these regions share the protection domain with QP_1.

Note that, although a memory window is bound to memory_region_A, QP_1 can access the region directly by providing the local key.

2. This case is similar to case 1, but for QP_2. Like QP_1, QP_2 can access all memory regions in the same protection domain as long as the work request that tries to access the memory region contains the right local key.

3. This case is similar to case 1 and 2, but for memory_region_C since QP_3 resides in protection_domain_Y. It is thus only possible to access memory locations in the main memory in the address range from `0x12` to `0x15` with the current configuration. To access other addresses, memory_region_C would have to be rebound.

4. This case illustrates the reception of an *RDMA write*. **Important note**: If a remote host writes into the local memory with an *RDMA write*, this will not really consume a receive WR. It is completely processed by the HCA without the QPs and CQs, and thus the OS and processes, even noticing this. Displaying (4) like this was done for the sake of simplicity and clarity.

   If a remote host wants to access `0x0A` or `0x0B` it can use the remote key of memory_window_1 to access it. Note that remote access does not necessarily have to be turned on for memory_region_A; only local write access is necessary.

2.2.8 Communication management

The *Communication Manager* (CM) provides protocols to establish, maintain, and release channels. It is used for all service types which were introduced in subsection 2.2.1. In the following, a brief introduction on the establishment and termination of communication will be given. As aforementioned, the present work will ignore special cases for the reliable datagram service type, since it is not supported by the OFED™ stack.

Since the communication manager is a general service, it makes use of GMPs for communication (see "Management datagrams" in subsection 2.2.3 and the composition of MADs in Figure 2.9). The CM has a set of messages which is set in the *AttributeID* of the common MAD header. A short summary of communication management related messages which are mandatory for IBA hosts that support RC, UC, and RD can be found in Table 2.3. Conditionally required messages for IBA hosts that support UD can be found in Table 2.4. Every message type needs different additional information which is set in the MAD data field. The exact content of this data for all message types can be found in the IBA specification [07].

As mentioned in subsection 2.2.2, the queue pair gets all necessary information in order to reach a remote node as arguments while transitioning *initialized* → *ready to receive*.

Communication establishment sequences There are various sequences of messages to establish or terminate a connection. Figure 2.19 introduces three commonly

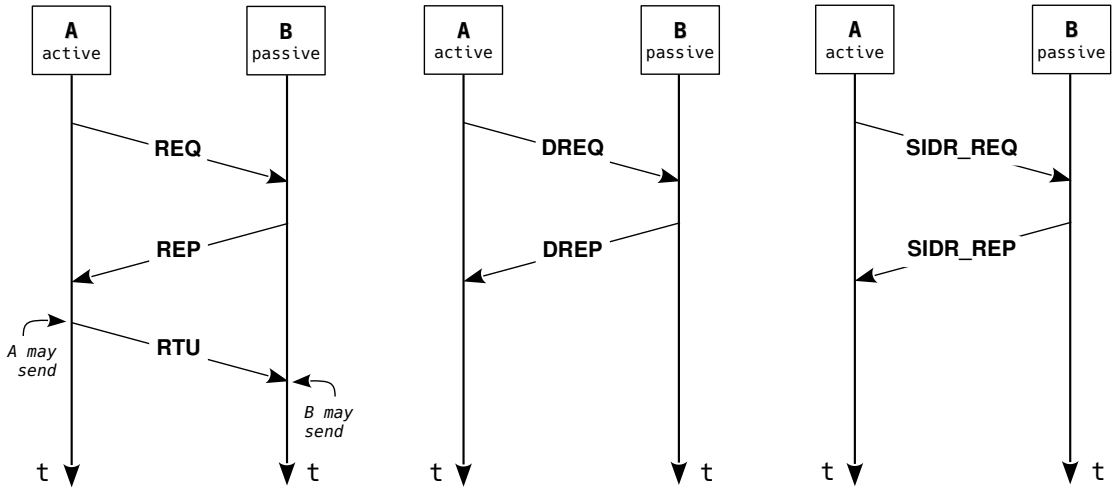
Table 2.3: Required Communication Management messages, used for all service types except Unreliable Datagram (UD).

CM message	**Description**
REQ	A *request for communication* is used to initiate the communication establishment sequence. The node that sends this message provides the remote host with its queue pair number and its GID and/or LID.
MRA	A *message receipt acknowledgment* is used as a response to a REQ, a LAP (Load Alternate Path, an optional communication message), or a REP. It is used if the node which receives one of the formerly mentioned messages does not expect to be able to respond within the specified time-out. With this mechanism, unnecessary time-outs are prevented.
REJ	By replying with *reject* to a REQ, a node indicates that it will not continue with the communication establishment sequence. The reason can be found, i.a., in the data field of the REJ.
REP	A node sends a *reply to REQ* if it wants to accept a previously received REP and all its parameters.
RTU	A node replies with *ready to use* after it received a REP to indicate that transmission can be started.
DREQ	A *request for communication release* is sent if a node wants to disconnect a queue pair.
DREP	A *response to DREQ* is used to acknowledge that a DREQ is received.

Table 2.4: Conditionally required Communication Management messages, used to acquire Unreliable Datagram (UD) addressing information.

CM message	**Description**
SIDR_REQ	The *Service ID Resolution Request* is used to request UD addressing information from a remote node for a certain service ID.
SIDR_REP	The *Service ID Resolution Response* is a response to the SIDR_REQ and contains all information to communicate with the entity that was specified as service ID in the request message.

used sequences. In all cases, the communication is established between an active client (A) and a passive server (B). It is also possible to establish communication between two active clients. If two active clients send a REQ, they will compare their GUID (or, if both clients share a GUID, their QPN), and the client with the smaller GUID (or QPN) will get assigned the passive role. A client can make its reply to a communication request conditional, e.g., rejecting the connection if it gets assigned the passive role.



(a) Communication establishment sequence for RC, UC, and RD.

(b) Communication release sequence for RC, UC, and RD.

(c) Service ID Request for UD.

Figure 2.19: Several Communication Management sequences. All depicted sequences take place between an active and a passive IBA host.

Communication establishment Figure 2.19a depicts the communication establishment sequence for connected service types and for reliable datagram. First, the active host A sends a *request for communication* (REQ). If B wants to accept the communication it replies with *reply to REQ* (REP). If it does not want to accept the communication request, it replies with *reject* (REJ). If it is not able to reply within the time-out that is specified in the received REQ, it answers with *message receipt acknowledgment* (MRA).

As soon as A has received the REP, it sends a *ready to use* (RTU) to indicate that transmission can start.

Communication release Figure 2.19b depicts the communication release sequence for RC, UC, and RD. The active host takes the initiative and sends a *request for communication release* (DREQ). The passive node acknowledges this with a

response to DREQ (DREP). These messages travel out of band, so if there are still operations in progress, it cannot be predicted how they will be completed.

Service ID request Figure 2.19c illustrates how *A* sends a *Service ID Resolution Request* (SIDR_REQ) in order to receive all necessary information from *B* to communicate over unreliable datagram. This information is sent from *B* to *A* over a *Service ID Resolution Response* (SIDR_REP).

2.3 OpenFabrics software libraries

Although the IBA specification [07] defines the InfiniBand Architecture and abstract characteristics of functions which should be included, it does not define a complete *Application Programming Interface* (API). Initially, the IBTA planned to leave the exact API implementation open to the several vendors. However, in 2004, the non-profit OpenIB Alliance (since 2005: OpenFabrics Alliance) was founded and released the *OpenFabrics Enterprise Distribution* (OFED™) under the GPL v2.0 or BSD license [18g]. The OFED™ stack includes, i.a., software drivers, core kernel-code, and user-level interfaces (verbs) and is publicly available online.[1,2] Most InfiniBand vendors fetch this code, sometimes make small enhancements and modifications, and ship it with their hardware.

Figure 2.20 shows a simplified sketch of the OFED™ stack. This illustration is based on a depiction of Mellanox' OFED™ stack [18f]. In this picture, the SCSI RDMA Protocol (SRP), all example applications, and all iWARP related stack components are omitted. The present work will mainly concentrate on the interface for the user space: the OpenFabrics user verbs (in the remainder of the present work, simply referred to as *verbs*) and the RDMA CM.

When having read section 2.2, the names of most verbs are self-explanatory (e.g., `ibv_create_qp()`, `ibv_alloc_pd()`, `ibv_modify_qp()`, and `ibv_poll_cq()`). This section will highlight some functions which often reoccur in the implementations in chapter 4—i.e., the structure of work requests and how to submit them in subsection 2.3.1—or functions which are not or hardly defined in the IBA—i.e., event channels in subsection 2.3.2 and the RDMA communication manager in subsection 2.3.3. A complete, alphabetically ordered list of all verbs with a brief description on them can be found in appendix A.

2.3.1 Submitting work requests to queues

Scatter/gather elements Submitting work requests is a crucial part of the datapath and enables processes to commission data transfers to the host channel adapter without kernel intervention. As presented in subsection 2.2.2, both send and receive work queue elements contain one or several memory location(s), which the HCA will

[1] https://github.com/linux-rdma/rdma-core
[2] https://git.kernel.org/pub/scm/linux/kernel/git/rdma/rdma.git

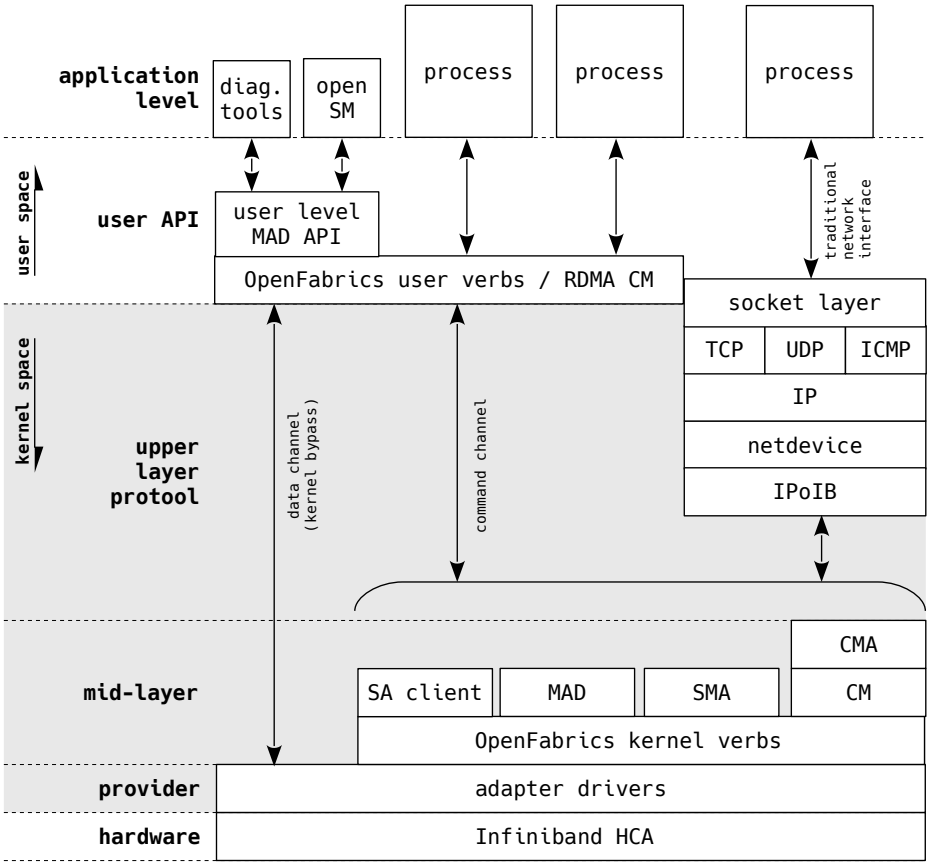


Figure 2.20: A simplified overview of the OFED™ stack.

use to read data from, or write data to. Work requests include a pointer to a list of at least one *scatter/gather element* (sge). This is a simple structure that includes the memory address, the length, and, in order for the HCA to be able to actually access the memory location, the local key. The structure of a scatter/gather element is displayed in Listing 2.1.

```
1 struct ibv_sge {
2         uint64_t                addr;
3         uint32_t                length;
4         uint32_t                lkey;
5 };
```

Listing 2.1: The composition of `struct ibv_sge`.

Receive work requests A receive work request, which is used to inform the HCA about the main memory location where received data should be written to, is a rather

simple structure as well. The structure, which is shown in Listing 2.2, includes a pointer to the first element of a scatter/gather list (`*sg_list`) and an integer to define the number of elements in the list (`num_sge`). Passing a list with several memory locations can be handy if data should be written to different locations, rather than to one big coherent memory block. The `*next` pointer can be used to link a list of receive work requests together. This is helpful if a process first prepares all work requests, and subsequently wants to call `ibv_post_recv()` just once, on the first work request in the list. The HCA will automatically retrieve all following WRs. The unsigned integer `wr_id` is optional and can be used to identify the resulting completion queue entry.

```
1 struct ibv_recv_wr {
2         uint64_t                wr_id;
3         struct ibv_recv_wr     *next;
4         struct ibv_sge         *sg_list;
5         int                     num_sge;
6 };
```

Listing 2.2: The composition of `struct ibv_recv_wr`.

Send work requests A send work request, displayed in Listing 2.3, is a larger structure and tells a lot about the several options (some) InfiniBand adapters offer. The first four elements are identical to those of the `ibv_recv_wr` C structure. They provide a way to match a CQE with a WR, offer the possibility to create a list of WRs, and enable the user to specify a pointer to and the length of a list of scatter/gather elements.

The fifth element, `opcode`, defines the operation which is used to send the message. Which operations are allowed depends on the type of the queue pair the present work request will be sent to; Table 2.5 shows all possible operations together with the service types they are allowed in. `send_flags` can be set to a bitmap of the following flags:

- `IBV_SEND_FENCE`: The WR will not be processed until all previous *RDMA read* and *atomic* WRs in the send queue have been completed.
- `IBV_SEND_SIGNALED`: If a QP is created with `sq_sig_all=1`, completion queue entries will be generated for every work request that has been submitted to the SQ. Otherwise, CQEs will only be generated for WRs with this flag explicitly set.

  This only applies to the send queue. Signaling cannot be turned off for the receive queue.
- `IBV_SEND_SOLICITED`: This flag must be set if the remote node is waiting for an event (subsection 2.3.2), rather than actively polling the completion queue.

This flag is valid for *send* and *RDMA write* operations and will wake up the remote node if it is waiting for a solicited message.

- `IBV_SEND_INLINE`: If this flag is set, the data to which the scatter/gather element points is directly copied into the WQE by the CPU. That means that the HCA does not need to independently copy the data from the host's main memory to its own internal buffers. Consequently, this saves an additional main memory access operation and, since the HCA's DMA engine will not access the main memory, the local key that is defined in the scatter/gather element will not be checked. Sending data inline is not defined in the original IBA and thus not all RDMA devices support it. Before sending a message inline, the maximum supported inline size has to be checked by querying the QP attributes using `ibv_query_qp()`.

  This flag is frequently used in the remainder of the present work because it offers a potential latency decrease and the buffers can immediately be released for re-use after the send WR got submitted.

Table 2.5: Supported operations with various service types. Although Reliable Datagram (RD) theoretically supports all operations, it is not supported by the OFED™ stack.

Operation	UD	UC	RC	RD
send	✓	✓	✓	
send with immediate	✓	✓	✓	
RDMA write		✓	✓	
RDMA write with immediate		✓	✓	
RDMA read			✓	
atomic compare & swap			✓	
atomic fetch & add			✓	

The 32-bit `imm_data` variable is used with operations that send data *with immediate* (Table 2.5). The data will be sent in the data packet's ImmDt field (Table 2.2). Besides sending 32 bit of data to the remote's completion queue—for example, as identifier—the immediate data field can also be used for notification of *RDMA writes*. Usually, the remote host does not know whether an *RDMA write* message is written to its memory and thus does also not know when it is finished. Since *RDMA write with immediate* consumes a receive WQE and subsequently generates a CQE on the receive side, this operation can be used as a way to synchronize and thus make the receiving side aware of the received data.

The fields `rdma`, `atomic`, and `ud` are part of a union, hence, mutually exclusive. The first two structs are used together with the operations with the same name

from Table 2.5. The content of the `rdma` C structure defines the remote address and the remote key, which first have to be acquired through a normal *send* operation. The `atomic` C structure includes the remote address and key, but also a compare and swap operand. The `ud` structure is used for unreliable datagram. As mentioned before, QPs in UD mode are not connected and the consumer has to explicitly define the *Address Handle* (AH) of the remote QP in every WR. The AH is included in the WR through the `*ah` pointer, and can, for example, be acquired with the RDMA communication manager which is presented in subsection 2.3.3. The `remote_qpn` and `remote_qkey` variables are used for the queue pair number and queue pair key of the remote QP, respectively.

```
1  struct ibv_send_wr {
2          uint64_t                wr_id;
3          struct ibv_send_wr      *next;
4          struct ibv_sge          *sg_list;
5          int                     num_sge;
6          enum ibv_wr_opcode      opcode;
7          int                     send_flags;
8          uint32_t                imm_data;
9          union {
10                 struct {
11                         uint64_t        remote_addr;
12                         uint32_t        rkey;
13                 } rdma;
14                 struct {
15                         uint64_t        remote_addr;
16                         uint64_t        compare_add;
17                         uint64_t        swap;
18                         uint32_t        rkey;
19                 } atomic;
20                 struct {
21                         struct ibv_ah   *ah;
22                         uint32_t        remote_qpn;
23                         uint32_t        remote_qkey;
24                 } ud;
25         } wr;
26 };
```

Listing 2.3: The composition of `struct ibv_send_wr`.

2.3.2 Event channels

Usually, completion queues (subsection 2.2.2) are checked for new entries by actively polling them with `ibv_poll_cq()`; this is called *busy polling*. In order for this to return a CQE as soon as one appears in the completion queue, polling has to be done continuously. Although this is the fastest way to get to know if a new CQE

is available, it is very processor intensive: a CPU core with a thread which continuously polls the completion queue will always be utilized 100 %. If minimal CPU utilization outweighs performance, the OFED™ user verbs collections offers *Completion Channels* (CCs). Here, an instance of the `ibv_comp_channel` C structure is created with `ibv_create_comp_channel()` and is, on creation of the completion queue, bound to that queue. After creation and every time after an event is generated, the completion queue has to be armed with `ibv_req_notify_cq()` in order for it to notify the CC about new CQEs. To prevent races, events have to be acknowledged using `ibv_ack_cq_event()`. Events do not have to be acknowledged before new events can be received, but all events have to be acknowledged before the completion queue is destroyed. Since this operation is relatively expensive, and since it is possible to acknowledge several events with one call to `ibv_ack_cq_event()`, acknowledgments should be done outside of the datapath.

The completion channel is realized with help of the Linux system call `read()` [Ker10]. In default mode, `read()` tries to read a file descriptor `fd` and blocks the process until it can return. Hence, as long as `fd` is not available, the operating system hibernates the process, which enables it to schedule other processes to the CPU. Because `read()` is used, the C structure of the channel, displayed in Listing 2.4, is not much more than a mere file descriptor and a reference counter. The blocking function which is used to wait for a channel is `ibv_get_cq_event()`; this function is a wrapper around `read()`.

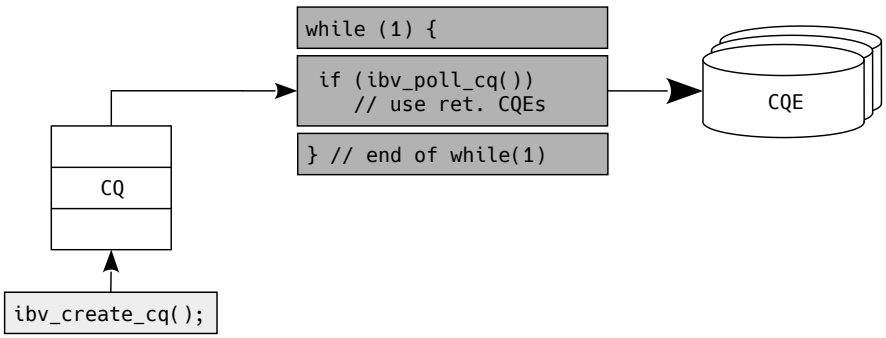
```
1 struct ibv_comp_channel {
2         struct ibv_context     *context;
3         int                     fd;
4         int                     refcnt;
5 };
```

Listing 2.4: The composition of `struct ibv_comp_channel`.

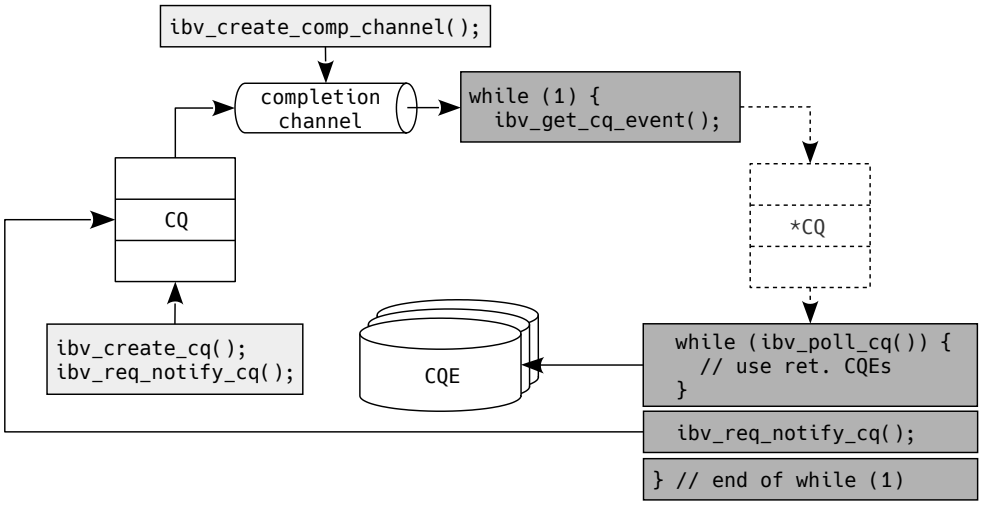
Figure 2.21 depicts a comparison between busy polling and polling after an event channel returns (*event based polling*). Figure 2.21a depicts busy polling, in which `ibv_poll_cq()` is placed in an endless loop and continuously polls the completion queue. In order to achieve low latencies—in other words in order to poll as often as possible—this takes place in a separate thread. If `ibv_poll_cq()` returns a value `ret > 0`, it was able to retrieve `ret` completion queue entries. These can now be processed, for example, to release the buffers they are pointing to.

Event based polling, depicted in Figure 2.21b, is a little bit more complex. As described above, first, a completion channel is created and is bound to the completion queue during initialization. Then, the CQ must be informed with `ibv_req_notify_cq()` about the fact that it should notify the completion channel whenever a CQE arrives. After initialization, the completion channel will be read with `ibv_get_cq_event()`. This happens again in a separate thread, this time because `ibv_get_cq_event()` will block the thread as long as no CQE arrives in the

completion queue. Whenever the function returns, it also returns a pointer to the original CQ, which in turn can be used to busy poll the queue for limited amount of time. However, there are two important differences to regular busy polling: when the CQ is polled the first time, it is ensured that it will return at least one CQE. Furthermore, after it has been polled the first time, the thread will continue to poll it, but as soon as `ibv_poll_cq()` returns 0, the process will re-arm the CQ and return to the blocking function. (Acknowledging with `ibv_ack_cq_event()` is omitted from this example for the sake of simplicity; it has to be called at least once before the completion queue is destroyed.)



(a) The working principle of busy polling.



(b) The working principle of event based polling.

Figure 2.21: A comparison between busy polling and polling after an event channel returns.

2.3.3 RDMA communication manager library

Because communication management can be quite cumbersome in the IBA, Annex A11 of the IBA specification [07] proposes the RDMA IP connection manager, which is implemented by the OpenFabrics Alliance. It offers a socket-like connection model and encodes the connection 5-tuple (i.e., protocol, source and destination IP and ports) into the private data of the CM REQ field (subsection 2.2.8).

RDMA CM over IPoIB The OFED™ `librdmacm`[3] library makes use of *Internet Protocol over InfiniBand* (IPoIB) in its implementation of this communication manager. IPoIB uses an unreliable datagram queue pair to drive communication because this is the only mode which must be implemented by HCAs and because of its multicast support [Kas15]. As can be seen in Figure 2.20, the Linux IPoIB driver enables processes to access the InfiniBand HCA over the TCP/IP stack. On one hand, this eradicates InfiniBand's advantages like kernel bypass. On the other hand, this offers an easy to set up interface to other InfiniBand nodes. All tools capable of working with the TCP/IP stack are also able to work with the TCP/IP stack on top of the IPoIB driver. Because of this, the RDMA communication manager is able to send *Address Resolution Protocol* (ARP) requests to other nodes which support IPoIB on the InfiniBand network. The ARP response will—assuming that a node with the requested IP address is present in the network—include a 20 B *MAC address*. This address consists of—listed from the MSB to the LSB—1 reserved byte, a 3-byte QPN field, and a 16-byte GID field. It is important to note that some applications or operating systems may have problems with the length of IPoIB's MAC addresses since an EUI-48–which has a length of 6 B instead of 20 B—is mostly used in IEEE 802 [17].

Thus, after the IPoIB drivers are loaded and the interface is properly configured using tools like `ifconfig` or `ip`, the RDMA CM is able to retrieve the queue pair number and global identifier of a remote queue pair with the help of a socket like construct.

Communication identifier & events The abovementioned socket like construct is realized through so-called *communication identifiers* (`struct rdma_cm_id`). Unlike conventional sockets, these identifiers must be bound to a local HCA before they can be used. During creation of the identifier with `rdma_create_id()`, an event channel, conceptually similar to the channels presented in subsection 2.3.2, can be bound to the identifier. If such a channel is present, all results of operations (e.g., resolve address, connect to remote QP) are reported asynchronously, otherwise the identifier will operate synchronously. In the latter case, calls to functions that usually cause an event on the channel will block until the operation completes. The former case makes use of a function similar to `ibv_get_cq_event()`: `rdma_get_cm_event()` also implements a blocking function that only returns when an event occurs on

[3] https://github.com/linux-rdma/rdma-core/blob/master/librdmacm

the channel. This function can be used in a separate thread to monitor events that occur on the identifier and to act on them. It is possible to switch between synchronous and asynchronous mode.

Queue pairs can be allocated to an `rdma_cm_id`. Because the identifier keeps track of the different communication events that occur, it will automatically transition the QP through its different states; explicitly invoking `ibv_modify_qp()` is no longer necessary.

2.4 Real-time optimizations in Linux

This section introduces optimizations that can be applied to systems running on the Linux operating system. It expands upon techniques that were applied to the Linux environment all benchmarks and VILLASnode instances were executed on and upon memory optimizations of the code. Of course, the optimizations in this section are a mere subset of all possibilities. The first subsection (2.4.1) elaborates on memory optimization, the second subsection (2.4.2) specifically on non-uniform memory access, the third subsection (2.4.3) on CPU isolation and affinity, the fourth subsection (2.4.4) on interrupt affinity, and finally, the last subsection (2.4.5) elaborates on the `tuned` daemon.

This section will not expand on the `PREEMPT_RT` patch [RH07] because it could not be used together with the current OFED™ stack. Possible opportunities of this real-time optimization with regards to InfiniBand applications are further expanded upon in section 7.1.

2.4.1 Memory optimizations

There are lots of factors that determine how efficiently memory is used: they can be on a high level—e.g., the different techniques that are supported by the OS—but also on a low level—e.g., by changing the order of certain memory accesses in the actual algorithm. Exploring all these different techniques is beyond the scope of the present work; rather, some techniques that are used in the benchmarks and in the implementation of the *InfiniBand* node-type are discussed in this subsection. The interested reader is referred to Drepper's publication [Dre07], which provides a comprehensive overview of methods that can be applied to optimize memory access in Linux.

Hugepages Most modern operating systems—with Linux being no exception—support *demand-paging*. In this method, every process has its own *virtual memory* which appears to the process as a large contiguous block of memory. The OS maps the *physical addresses* of the actual physical memory (or even of a disk) to *virtual addresses*. This is done through a combination of software and the *memory management unit* (MMU) which is located in the CPU.

Memory is divided into *pages*. It is the smallest block of memory that can be accessed in virtual memory. For most modern operating systems, the smallest page size is 4 KiB; in a 64-bit architecture these 4 KiB can hold up to 512 words. If a process tries to access data at a certain address in the virtual memory which is not yet available, a *page fault* is generated. This exception is detected by the MMU, which in turn tries to map the complete page from the physical memory (or from a disk) into the virtual memory.

Page faults are quite expensive and it is beneficial for performance to cause as little as possible page faults [Dre07]. One possible solution to achieve this is to increase the size of the pages: Linux supports so-called *hugepages*. Although there are several possible sizes for hugepages, on x86-64 architectures they are usually 2 MiB [18b]. Compared to the 512 words that can fit into a 4 KiB page, the hugepage can fit 262 144 words into one page, which is 512 times as much. Since more data can be accessed with less page faults, this will increase performance; Drepper [Dre07] reports performance gains up to 57 % (for a working set of 2^{20} B).

Additionally, with hugepages, more memory can be mapped with a single entry in the *translation lookaside buffer* (TLB). This buffer is part of the MMU and caches the most recently used page table entries. If a page is present in the TLB (*TLB hit*), resolution of a page in the physical memory is instantaneous. Otherwise (*TLB miss*) up to four memory accesses in x86-64 architectures are required [Gan+16]. Since the TLB size is limited, larger pages result in the instantaneous resolution of a larger range of addresses with the same size TLB.

Using hugepages is not an all-in-one solution; it has some disadvantages that have to be considered. When page sizes are becoming bigger, it gets harder for the MMU to find contiguous physical memory sectors of this size. This goes hand in hand with external fragmentation of the memory. Furthermore, the size of hugepages makes them more prone to internal fragmentation, which means that more memory is allocated than is actually needed.

Alignment A memory address a is *n-byte aligned* when

$$a = C \cdot n = C \cdot 2^i, \qquad \text{with } i \geq 0, \quad C \in \mathbb{Z}. \tag{2.12}$$

An n-byte aligned address needs to meet the sufficient condition that $log_2(n)$ LSBs of the address are `'0'`.

Figure 2.22 shows a simple example for a 32-bit system with the three primitive C data types from Listing 2.5. In Figure 2.22a the data is *naturally aligned*: the compiler added padding between the data types to ensure alignment to the memory word boundaries. In the structure definition of Listing 2.5b, the compiler is compelled to omit additional padding: the data types are not aligned to word boundaries. Note that equation (2.12) holds in Figure 2.22a, but not in Figure 2.22b. Furthermore, for Figure 2.22a, additional 1-bit characters could be placed at `0x0001`, `0x0002`, `0x0003`, and `0x000A` in this example. Additional 2-bit shorts could be placed at `0x0002` and `0x000A`.

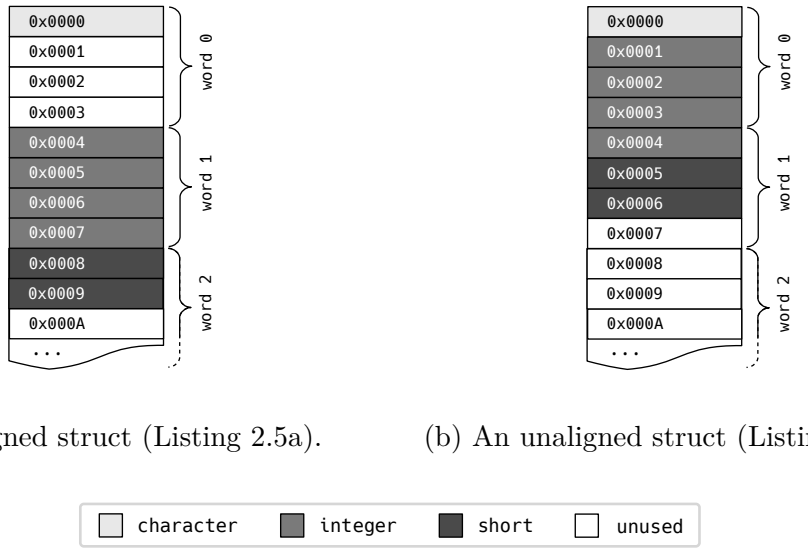
```c
struct a {
    char c;
    int i;
    short s;
}
```

a: Struct with padding.

```c
struct __attribute__((__packed__)) b {
    char c;
    int i;
    short s;
}
```

b: Packed struct without padding.

Listing 2.5: Two C structures with an 1-bit character, a 4-bit integer, and a 2-bit short.



(a) An aligned struct (Listing 2.5a).

(b) An unaligned struct (Listing 2.5b).

Figure 2.22: An example of an 1-bit character, a 4-bit integer, and a 2-bit short from Listing 2.5 in memory with a word size of 32 B.

Similar to pages, a system can only access one whole word at a time. In Figure 2.22a, this translates to one memory access per data type. In Figure 2.22b, however, this is no longer possible. To access the integer, the operating system first has to access the word at address `0x0000` and then the word at address `0x0004`. Subsequently the value in the first word must be shifted one position and the value in the second word three positions. Finally, both words have to be merged. These additional operations cause additional delay when trying to access the memory. Moreover, atomicity becomes more difficult to guarantee, since the OS needs to access two memory locations to access one data type.

Alignment is not only relevant for memory words. Not aligning allocated memory to cache lines significantly slows down memory access [Dre07]. Furthermore, due to the way the TLB works, alignment can speed up resolution of addresses in the physical memory.

Pinning memory The process of preventing the operating system from swapping out (parts of) the virtual address space of a process is called *pinning memory*. It is invoked by calling `mlock()` to prevent parts of the address space from being swapped out, or `mlockall()` to prevent the complete address space from being swapped out.[4]

Explicitly pinning buffers that are allocated to use as source or sink for data by an HCA is not necessary: when registering a memory region (subsection 2.2.7), the registration process automatically pins the memory pages [15].

2.4.2 Non-uniform memory access

If different memory locations in the address space show different access times, this is called *non-uniform memory access* (NUMA). A common example of a NUMA system is a computer system with multiple CPU sockets and thus also multiple system buses. Because a NUMA node is defined as memory with the same access characteristics, here, this is the memory which is closest to the respective CPU. Accessing memory on a remote NUMA node adds up to 50 percent to the latency for a memory access [Lam13].

Figure 2.23 depicts an example with two NUMA nodes and the interconnect between them. It is beneficial for the performance of processes to access only memory which is closest to the processor that executes the process. Furthermore, regarding the InfiniBand applications later presented in the present work, it is beneficial to run processes that need to access a certain HCA on the same NUMA node as the HCA. An HCA is connected to the system bus through the *Peripheral Component Interconnect Express* (PCI-e) bus, hence, access of memory in the same NUMA node will be faster than access of memory on a remote NUMA node. Thus, in case of Figure 2.23, if a process needs to access HCA 0, it should be scheduled on one or more cores on processor 0 and should be restricted to memory locations of memory 0.

To set the memory policy of processes, tools like `numactl`,[5] which are based on the system call `set_mempolicy()`,[6] can be used. These tools will not be further elaborated upon here since the next subsection will introduce a more general tool to constrain both CPU cores and NUMA nodes to processes.

2.4.3 CPU isolation & affinity

Isolcpus It is beneficial for the performance of a process if one or more CPU cores (in the remainder of the present work often simply referred to as *cores* or *CPUs*) are completely dedicated to its execution. Historically, the `isolcpus`[7] kernel parameter is used to exclude processor cores from the general balancing and scheduler algorithms on symmetric multiprocessing architectures. With this exclusion, processes

[4] http://man7.org/linux/man-pages/man2/mlock.2.html
[5] http://man7.org/linux/man-pages/man8/numactl.8.html
[6] http://man7.org/linux/man-pages/man2/set_mempolicy.2.html
[7] https://www.kernel.org/doc/Documentation/admin-guide/kernel-parameters.txt

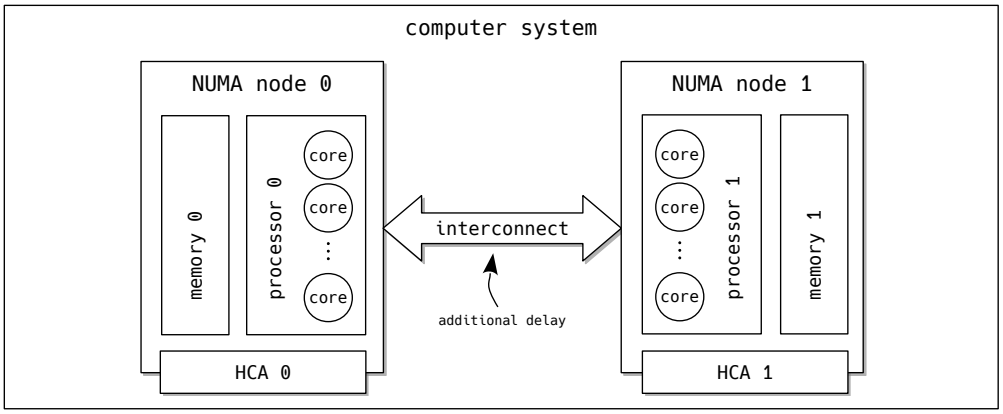


Figure 2.23: Two non-uniform memory access (NUMA) nodes with HCAs on the respective PCI-e buses.

will only be moved to excluded cores if their affinity is explicitly set to these cores with the system call `sched_setaffinity()` [Ker10]. The tool `taskset`,[8] which relies on the aforementioned system call, is often used to set the CPU affinity of running processes or to set the affinity of new commands.

The major advantage of `isolcpus` is at the same time its biggest disadvantage: the exclusion of cores from the scheduling algorithms causes threads, that are created by a process, to always be executed on the same core as the process itself. Take the example of busy polling: if a thread that must busy poll a completion queue is created and is executed on the same core as the primary thread, this has an adverse effect on the performance of the latter. So, it is desired to isolate CPU cores that are dedicated to certain explicitly defined processes, but simultaneously enable efficient scheduling of threads of these processes among the isolated cores.

Cpusets A possible solution to this problem is offered by *cpusets* [Der+04] which uses the generic *control group* (cgroup) [MJL08] subsystem. If this mechanism is used, requests by a task to include CPUs in its CPU affinity or requests to include memory nodes are filtered through the task's cpuset. That way, the scheduler will not schedule a task on a core that is not in its `cpuset.cpus` list and not use memory on NUMA nodes which are not in the `cpuset.mems` list.

Cpusets are managed through the *cgroup virtual file system* and each cpuset is represented by a directory in this file system. The root cpuset is located under `/sys/fs/cgroup/cpuset` and includes all memory nodes and CPU cores. A new cpuset is generated by creating a directory within the root directory. Every newly created directory automatically includes similar files to the root directory. These files shall be used to write the cpuset's configuration to (e.g., with `echo`[9]) or to

[8] http://man7.org/linux/man-pages/man1/taskset.1.html
[9] http://man7.org/linux/man-pages/man1/echo.1.html

read the current configuration from (e.g., with `cat`[10]). The following settings are available for every cpuset [Der+04]:

- `cpuset.cpus`: list of CPUs in that cpuset;
- `cpuset.mems`: list of memory nodes in that cpuset;
- `cpuset.memory_migrate`: if set, pages are moved to cpuset's nodes;
- `cpuset.cpu_exclusive`: if set, cpu placement is exclusive;
- `cpuset.mem_exclusive`: if set, memory placement is exclusive;
- `cpuset.mem_hardwall`: if set, memory allocation is hardwalled;
- `cpuset.memory_pressure`: measure of how much paging pressure in cpuset;
- `cpuset.memory_pressure_enabled`[11]: if set, memory pressure is computed;
- `cpuset.memory_spread_page`: if set, page cache is spread evenly on nodes;
- `cpuset.memory_spread_slab`: if set, slab cache is spread evenly on nodes;
- `cpuset.sched_load_balance`: if set, load is balanced among CPUs;
- `cpuset.sched_relax_domain_level`: searching range when migrating tasks.

Once all desired cpusets are created and everything is set up by writing settings to the abovementioned files, tasks can be assigned by writing their *process identifier* (PID) to `/sys/fs/cgroup/cpuset/<name_cpuset>/tasks`.

Cpuset tool Since the process of manually writing tasks to the *tasks-file* can be quite cumbersome, there are several tools and mechanisms to manage which processes are bound to which cgroups.[12] A rudimentary tool that is used in the present work is called *cpuset*.[13] It was developed by Alex Tsariounov and is a Python wrapper around the file system operations to manage cpusets. The following examples on how to create cpusets, how to move threads between cpusets, and how to execute applications in a cpuset are all based on this tool. However, the exact same settings can also be achieved by writing values manually to the virtual file system.

Listing 2.6 shows how to create different subsets. (Here, and in the remainder of the present work, the octothorpe indicates that the commands must be executed by a superuser. A dollar sign indicates that the command can be executed by a normal user.) In this example, an arbitrary machine with 24 cores and two NUMA nodes (subsection 2.4.2) is assumed. The first cpuset, *system*, may use 16 of these cores exclusively and may use memory in both NUMA nodes. This will become the default cpuset for non-time-critical applications. The second and third cpuset, called *real-time-0* and *real-time-1* in this example, may use four cores each. These

[10] http://man7.org/linux/man-pages/man1/cat.1.html
[11] exclusive to root cpuset
[12] Although the libcgroup package was used in the past, systemd is nowadays the preferred method for managing control groups.
[13] https://github.com/lpechacek/cpuset

are exclusively reserved for time-critical applications. In this example, it is assumed that the CPUs 16, 18, 20, and 22 reside in NUMA node 0 and the CPUs 17, 19, 21, and 23 in NUMA node 1; the real-time cpusets are thus constrained to their respective nodes.

```
# cset set -c 0-15 -s system --cpu_exclusive
# cset set -c 16,18,20,22 -s real-time-0 --cpu_exclusive --mem=0
# cset set -c 17,19,21,23 -s real-time-1 --cpu_exclusive --mem=1
```

Listing 2.6: Creating cpusets for system tasks and real-time tasks.

The exclusiveness of a CPU to cpuset only applies to its siblings; tasks in the cpuset's parent may still use the CPU. Therefore, Listing 2.7 shows how to move threads and movable kernel threads from the root cpuset to the newly created *system* cpuset. Now, the execution of these tasks and of all their children exclusively takes place on CPUs that range from 0 to 15.

```
# cset proc --move -f root -t system --kthread --thread --force
```

Listing 2.7: Moving all tasks, threads, and moveable kernel threads to *system*.

This leaves the two real-time cpusets exclusively for high-priority applications. Listing 2.8 shows how new applications with their arguments can be started within the real-time cpusets.

To ensure that the load is balanced among the CPUs in a cpuset—a feature that is not supported by `isolcpus`—`cpuset.sched_load_balance` must be `'1'`. It is not necessary to explicitly set this value since its default value is already `'1'`.

```
# cset proc --set=real-time-0 --exec ./<application> -- <args>
# cset proc --set=real-time-1 --exec ./<application> -- <args>
```

Listing 2.8: Execute `<application>` with the arguments `<args>` in the real-time cpusets.

Non-movable kernel threads Kernel threads are background operations performed by the kernel. They do not have an address space, are created on system boot, and can only be created by other kernel threads [Lov10]. Although some of them may be moved from one CPU to another, this is not generally the case. Some kernel threads are pinned to a CPU on creation. Although it is not possible to completely exclude kernel threads from getting pinned to cores which will be shielded, there is a workaround which might minimize this chance.

By setting the kernel parameter `maxcpus`[14] to a number smaller than the total amount of CPU cores in the system, some cores will not be brought up during bootup. Hence, these processors will not be used to schedule kernel threads. Later, when all movable kernel threads are moved to a shielded cpuset, the remaining CPUs can be activated with the command from Listing 2.9. Then, these CPUs can be added to an exclusive cpuset. Although it is inevitable that some necessary threads will be spawned on these cores once they are brought up, most of the non-movable kernel threads cannot move from a processor that was available during bootup to a processor that was activated after bootup.

```
1 # echo 1 > /sys/devices/system/cpu/<cpuX>/online
```

Listing 2.9: Bring up a CPU `<cpuX>` which was disabled during bootup.

2.4.4 Interrupt affinity

In most computer systems, hardware *interrupts* provide a mechanism for I/O hardware to notify the CPU when it has finished the work it was assigned. When an I/O device wants to inform the CPU, it asserts a signal on the bus line it has been assigned to. The signal is then detected by the *interrupt controller* which decides if the targeted CPU core is busy. If this is not the case, the interrupt is immediately forwarded to the CPU, which in turn ceases its current activity to handle the interrupt. If the CPU is busy, for example, because another interrupt with a higher priority is being processed, the controller ignores the interrupt for the moment and the device keeps asserting a signal to the line until the CPU is not busy anymore [TB14].

Hence, if a CPU is busy performing time-critical operations—e.g., busy polling (Figure 2.21a)—too many interrupts are detrimental for the performance. Thus, it can be advantageous to re-route interrupts to CPUs that do not perform time-critical applications.

Listing 2.10 shows how to obtain the *interrupt request* (IRQ) affinity of a certain interrupt request `<irqX>`. The value `smp_affinity` is a *bitmap*, which means that the indices that are set represent the allowed CPUs [Bow+09]. E.g., when `smp_affinity` for a certain IRQ is `'10'`, it means that CPU 1 is allowed; if the affinity is `'11'`, it means that CPU 1 and 0 are allowed.

```
1 $ cat /proc/irq/<irqX>/smp_affinity
```

Listing 2.10: Get the IRQ affinity of interrupt `<irqX>`.

[14] https://www.kernel.org/doc/Documentation/admin-guide/kernel-parameters.txt

Listing 2.11 demonstrates how the IRQ affinity of a certain interrupt request can be set. In the case of Listing 2.11, it is set to CPU 0–15, which corresponds to the *system* cpuset from the previous paragraph. `<irqX>` will no longer bother the CPUs 16–23. To re-route all IRQs, a script (e.g., in Bash) that loops through `/proc/irq` can be used.

```
1 # echo FFFF > /proc/irq/<irqX>/smp_affinity
```

Listing 2.11: Set the IRQ affinity of interrupt `<irqX>` to CPU 0–15.

2.4.5 Tuned daemon

Red Hat based systems support the `tuned` daemon,[15] which uses `udev` [Kro03] to monitor devices and, on the basis of its findings, adjusts system settings to increase performance according to a selected profile. The daemon consists of two types of plugins: monitoring and tuning plugins. The former can, at the moment of writing the present work, monitor the disk load, network load, and CPU load. The tuning plugins currently supported are: cpu, eepc_she, net, sysctl, usb, vm, audio, disk, mounts, script, sysfs, and video.

Although it is possible to define custom profiles, `tuned` offers a wide range of predefined profiles, of which *latency-performance* is eminently suitable for low-latency applications. This profile does, among others, disable power saving mechanisms, set the CPU governor to *performance*, and lock the CPU to a low C-state. A complete overview of all settings in the *latency-performance* profile can be found in appendix B.

Management of different tuning profiles can be done with the command line tool `tuned-adm`.[16]

[15] https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/performance_tuning_guide/chap-red_hat_enterprise_linux-performance_tuning_guide-tuned

[16] https://linux.die.net/man/1/tuned-adm

3 Architecture

The first section of this chapter (3.1) explains the concept and internals of a VILLASnode instance. In the second section (3.2), a brief introduction on the configuration of node-type instances is given. Then, in section 3.3, 3.4, and 3.5, the adaptions that had to be made to the interface of node-types, the memory management of VILLASnode, and the finite-state machine of nodes are explained, respectively.

3.1 Concept

The functioning principles and general structure of VILLASframework, of which VILLASnode is a sub-project, were already presented in subsection 1.1.2. This section solely focuses on the structure of VILLASnode.

Table 1.1 presented the different *node-types* that VILLASnode supported at the time of writing the present work. One VILLASnode instance—in the remainder of the present work often referred to as *super-node*—may have several *nodes* which act as source and/or sink of simulation data. A node is defined as an instance of a node-type. Accordingly, a super-node can serve as a gateway for simulation data. Node-types can roughly be divided into three categories:

- *internal node-types*, which enable communication with node-types on the same host (e.g., writing data to a file descriptor through a *file* node);
- *server-server node-types*, which enable communication with nodes on different hosts (e.g., communicating with a *socket* node on a remote host);
- *simulator-server node-types*, which enable communication with simulators (e.g., acquiring data from an OPAL-RT simulator).

(In the remainder of this work, names of node-types and nodes are written in a cursive font, for example, *file* node, *socket* node, or *InfiniBand* node-type.)

Within a super-node, so called *paths* connect different nodes. A path starts at a node from which it acquires data. Immediately after data is obtained, it is optionally sent through a *hook*, which can be seen as an extension to manipulate the data (e.g., to filter or transform it). Then, the data is written into a FIFO (also called: *queue*), which holds it until it can be passed on. Subsequently, the data is sent through a *register*, which can multiplex and mask it. Before the data is placed into the output queue and right before the sending node obtains it, it can be manipulated by more hooks. Finally, if the output node is ready, the data is moved from the output queue to the output node, which then sends it to a given destination node.

Data is transmitted in *samples*, which store the simulation data for a given point in time, send and receive timestamps, and a sequence number. The sample structure is deliberately kept simple because it is the smallest common denominator of all supported simulators.

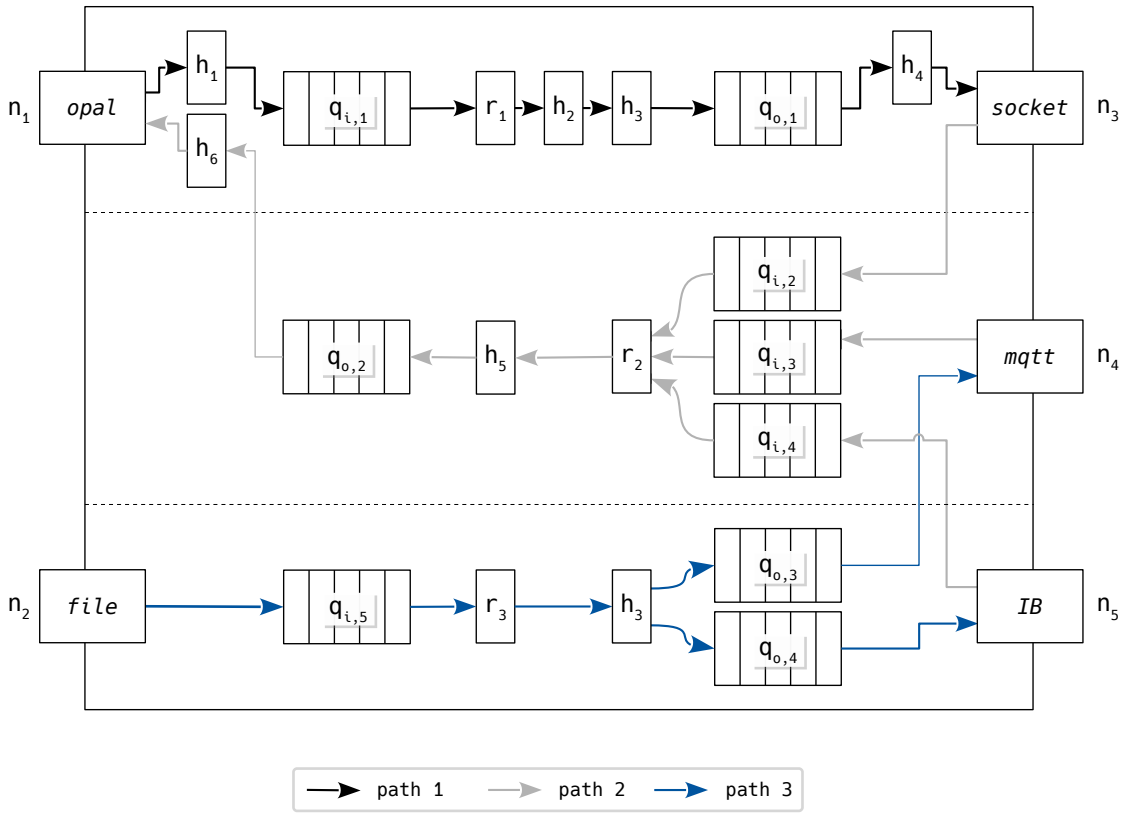


Figure 3.1: The internal VILLASnode architecture [Vog+17]. Depicted is one VILLASnode instance (*super-node*) that includes three *paths*, which connect five node-type instances (*nodes*) with each other.

Figure 3.1 depicts the internal connections of an example super-node. This VILLASnode instance includes five node-type instances: *opal* (n_1), *file* (n_2), *socket* (n_3), *mqtt* (n_4), and a yet to be implemented *InfiniBand* (n_5) node. On receive, data from the *opal* node n_1 is modified by hook h_1 before it is placed in queue $q_{i,1}$. Path 1 continues through register r_1, hook h_2, and hook h_3, before it enters the output queue $q_{o,1}$. Before the *socket* node n_3 sends the data from the queue to another *socket* node, it is modified one last time by hook h_4.

Path 2 connects a *socket* node (n_3), an *mqtt* node (n_4), and an *InfiniBand* node (n_5) with an *opal* node n_1. In this path, the register r_2 determines the forwarding conditions for $q_{i,2}$, $q_{i,3}$, and $q_{i,4}$; it could, for example, depending on the data available

in the queues, mask them. Before the data is placed in the output queue $q_{o,2}$ and right before the *opal* node sends the data, it is modified by hook h_5 and h_6, respectively.

Path 3 connects a *file* node n_2, which reads data from a local file, with an *mqtt* node n_4 and *InfiniBand* node n_5.

3.2 Configuration of nodes

Listing 3.1 shows an example of a stripped down VILLASnode configuration file. The first part of the configuration consists of a list of nodes to be initialized (comparable with $n_{1...5}$ in Figure 3.1). In this example, an instance of a *file* node-type (`node_1`) and an instance of an *InfiniBand* node-type (`node_2`) would be instantiated. Besides the type, a user can specify a range of settings for every node. These can be divided into global settings for the complete instance, settings only for the input part of the node, and settings only for the output part. The supported settings for every node-type can be found on the VILLASframework documentation pages.[1]

The *paths* section describes how nodes are connected within the super-node (compare with path 1, path 2, and path 3 in Figure 3.1). In this case, there is a path between `node_1` and `node_2`. This means that data is read from a file, which would be specified in the in-section of `node_1`, and then placed in a buffer in the super-node. Then, after it is sent through possible hooks—which are not defined in this configuration file—it is copied to the memory that is allocated as output buffer for the *InfiniBand* node. The super-node then sends these samples to the write-function of that node, which in turn sends the samples to a remote node as specified in its out-section.

3.3 Interface of node-types

To ensure interoperability between different node-types and VILLASnode, the VILLASframework specification defines an interface to use between the super-node and node-types. It is realized as a fixed set of functions with a given set of parameters that every node-type can implement. These functions have to be registered with the framework by passing it the pointers of the respective functions. Examples of functions to be implemented are `start()` and `stop()`, as well as `read()` and `write()`. Since their parameters had to be changed to efficiently support an *InfiniBand* node-type, this section will expand upon the latter.

Not every function is mandatory; some functions will simply be ignored if they are not implemented. A complete list of all functions a node-type should implement, together with a brief description, is presented in appendix C.

[1] https://villas.fein-aachen.org/doc/node-types.html

```
nodes = {
    node_1 = {
        type = "file",

        // Global settings for node_1

        in = {
            // Settings for node input, e.g., file to read from
        }
    },
    node_2 = {
        type = "infiniband",

        // Global settings for node

        in = {
            // Settings for node input, e.g., address of local
            // InfiniBand HCA to use
        },

        out = {
            // Settings for node output, e.g., remote InfiniBand
            // node to write to
        }
    }
},

paths = (
    {
        in = "node_1",
        out = "node_2"
    }
)
```

Listing 3.1: Structure of the configuration file of a *file* node and an *InfiniBand* node with a path connecting them.

3.3.1 Original implementation of the read- and write-function

Listing 3.2 shows the variables which were originally used in the `node_type` C structure (appendix D.3) to save the function pointers to the read- and write-function. Since this listing shows the initial parameters, it helps to understand the working principles of both functions and their weaknesses.

For both functions, `*n` is a C structure that holds information about the node-type instance. It contains, among others, information about the state, the number of generated or received samples, the configuration of the node and a field for node-type specific virtual data. The node structure is displayed in appendix D.2; the present work will not expand further upon this struct.

```
1 int (*read)(struct node *n, struct sample *smps[], unsigned cnt);
2 int (*write)(struct node *n, struct sample *smps[], unsigned cnt);
```

Listing 3.2: Original parameters of `read()` and `write()`

Read-function The working principle of the read-function is displayed in Figure 3.2. The *_read()* box represents the function to which the `(*read)` pointer (line 1 in Listing 3.2) of a given node-type points and is often simply referred to as *read-function* in the remainder of the present work. The box thus depicts a part of the interface between the super-node and the node.

In order to retrieve data from a node, the super-node starts by allocating `cnt` ≥ 1 empty samples. A sample contains fields for, i.a., an origin timestamp, a receive timestamp, a sequence number, a reference counter, and a field to save the actual signal. The signal can contain unsigned 64-bit integers, 64-bit floating-point numbers, booleans, or complex numbers. Appendix D.1 presents the `sample` C structure. Since this structure contains some host specific information, it contains more data than will actually be sent.

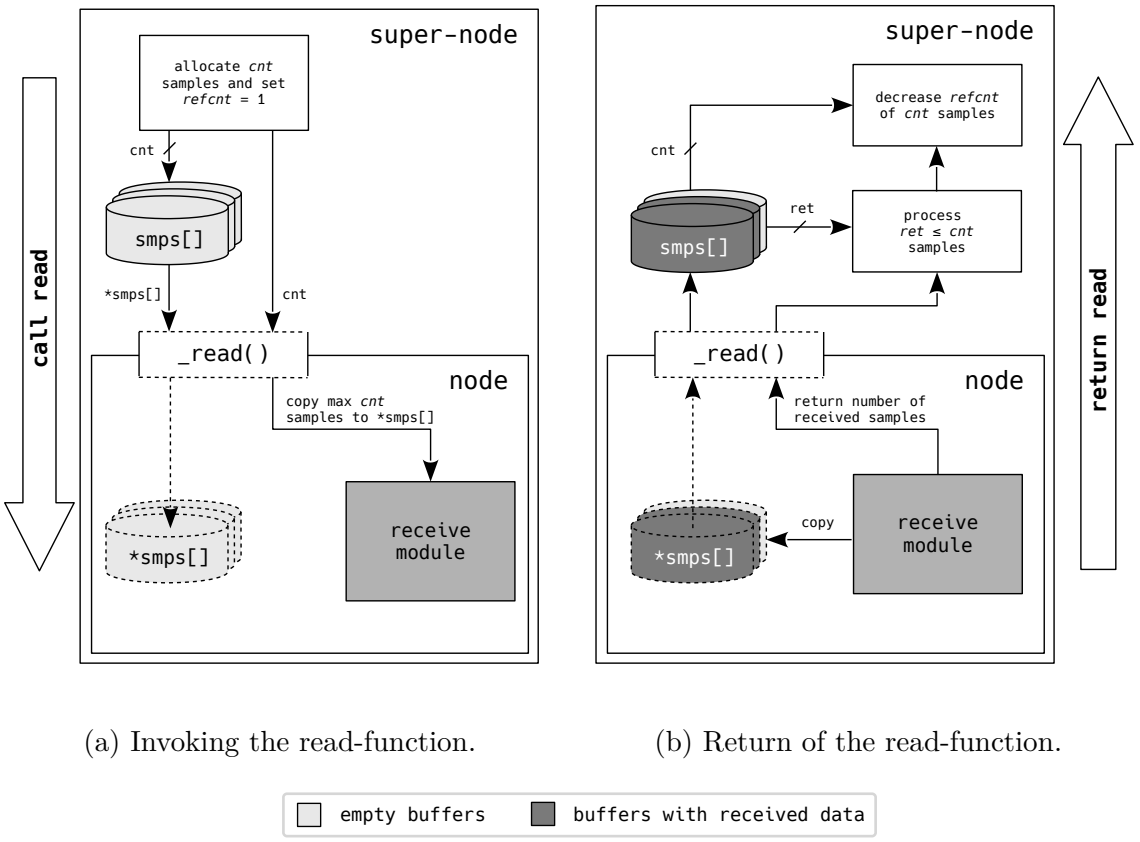
After samples have been allocated, their reference counter (*refcnt*) is increased by one. Samples in VILLASnode cannot be destroyed unless the reference counter is 1 when the release-function is called. When $refcnt > 1$, other instances within VILLASnode still rely on the sample; calling the release-function on such a sample will cause the reference counter to be decremented by 1. In the remainder of the present work, *releasing a sample* and *decreasing the reference counter of a sample by one* is used interchangeably.

After memory to hold the samples has been allocated, a pointer to the first sample (`*smps[]`) and the total number of allocated samples (`cnt`) is passed to the node by calling the read-function (Figure 3.2a). The node then tries to receive a maximum of `cnt` values to subsequently copy them to the allocated memory.

The return of the read-function is depicted in Figure 3.2b. After the receive module, which is blackboxed here, has filled up *ret* $\leq$ `cnt` samples, it lets the read-function return with *ret*. The super-node then processes *ret* samples (e.g., sending them through several hooks, before sending them to another node). Finally, all `cnt`—thus not only *ret*—samples are released. So, after a read cycle, the reference counter of all samples is decreased by 1, and in that way the samples are usually destroyed.

Write-function The write-function works in a similar fashion as the read-function and has identical parameters (line 2 in Listing 3.2). The working principle of this function is depicted in Figure 3.3. When a super-node's path needs to write data to a node, it calls the write-function (Figure 3.3a) and passes the total number of samples and the pointer to the first sample as arguments.

When the write-function is called, the node starts processing the samples by copying `cnt` samples to its send module and instructing it to send the data. The



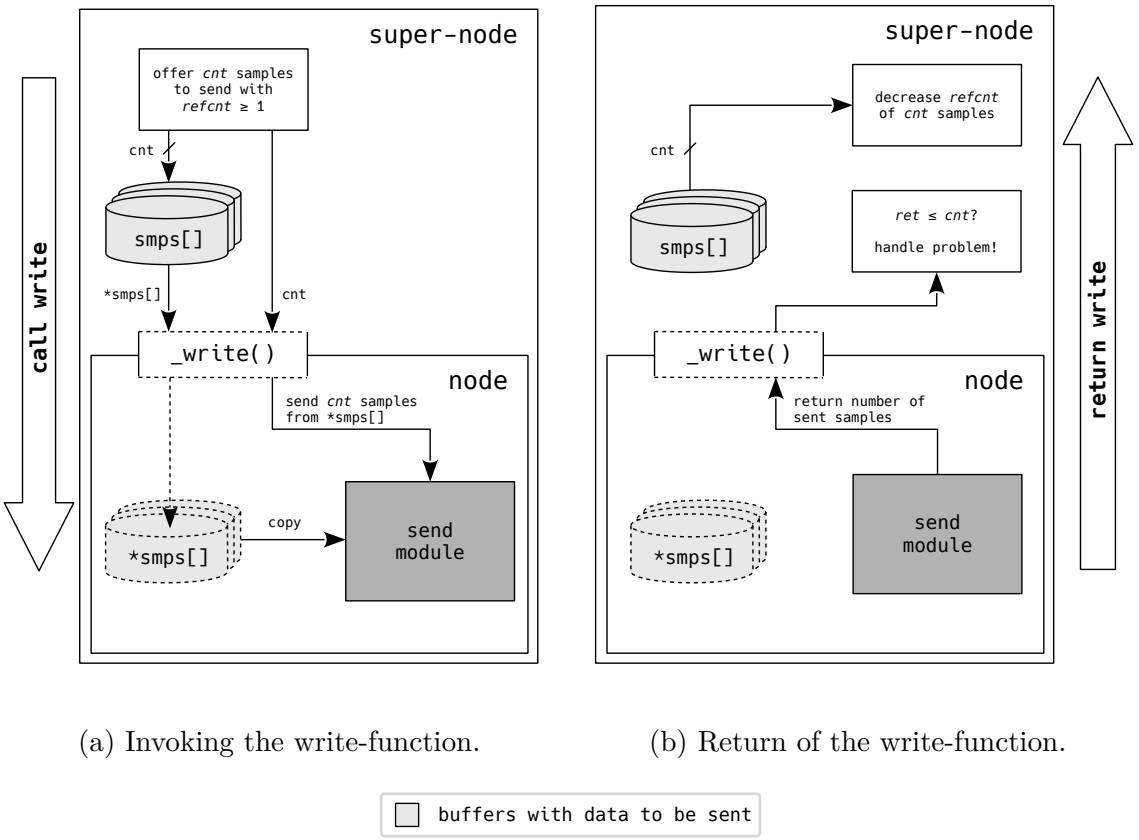
(a) Invoking the read-function. (b) Return of the read-function.

Figure 3.2: A depiction of the working principle of the read-function in VILLASnode. This function is part of the interface between a super-node and a node.

send module does not return until all samples are copied to the send module, and in case of many nodes, not until the data is successfully sent. When the send module is done, depicted in Figure 3.3b, it lets the write-function return with the number of samples that have been successfully sent. Ideally, the returned value *ret* is equal to the number of passed samples `cnt`. If this is not the case, the super-node will detect this and act upon a possible error. In all cases, the reference counter of all `cnt` samples is decremented by 1.

3.3.2 Requirements for the read- and write-function of an InfiniBand node

As discussed in the previous section, the reference counters of all samples that have been sent into the read- or write-functions are decreased after the functions return. For nodes with either a receive module that has a local buffer or with a send module which does not return until it has made a copy of the data or actually sent the data, this approach works exactly as intended. But, as soon as the modules are implemented by an architecture which is based on the VIA—in this particular case



(a) Invoking the write-function.

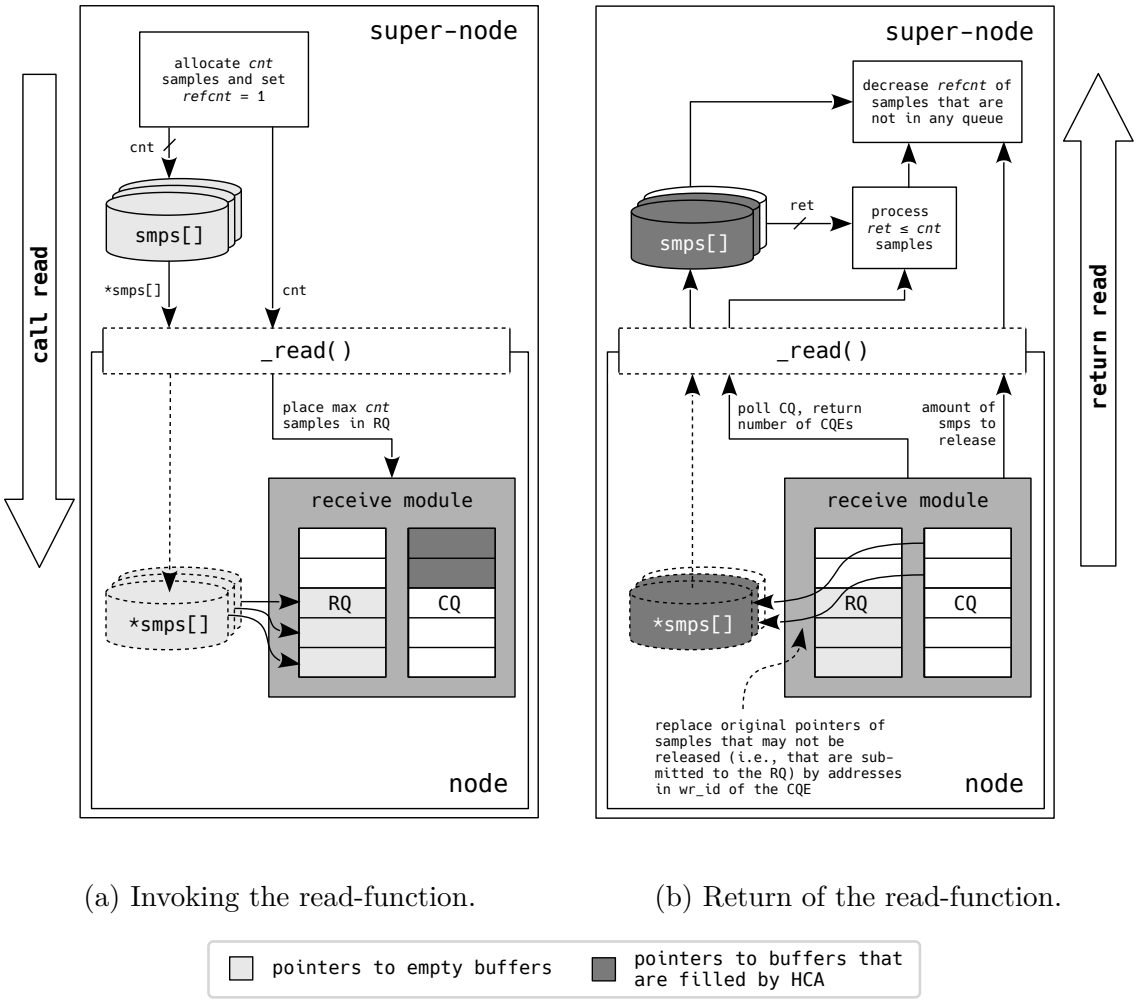
(b) Return of the write-function.

Figure 3.3: A depiction of the working principle of the write-function in VILLASnode. This function is part of the interface between a super-node and a node.

the IBA—the implementation causes problems. To adhere to the zero-copy principle of the VIA, data should not be copied from the super-node's buffer to a local buffer or the other way around. Rather, a pointer to, and the length of, a memory location should be passed to the network adapter, which then independently copies the data from the host's memory to its local buffers or the other way around.

In the following, the ideal situation for a read and write operation for the InfiniBand Architecture is presented. Although this approach is specifically for the IBA, it can relatively easily be translated to other VIAs. After the desired approach has been discussed, the next subsection will discuss the shortcomings of the parameters in Listing 3.2, that obstruct the implementation of this approach.

Read-function Figure 3.4 depicts a super-node that reads from a node-type instance whose communication is based on the IBA. The receive module in this figure relies on the receive queue of an InfiniBand QP. As explained in subsection 2.2.2, a queue pair cannot receive data unless its RQ holds receive WQEs. Hence, work requests that point to buffers of the super-node have to be submitted.



(a) Invoking the read-function. (b) Return of the read-function.

Figure 3.4: A depiction of the working principle of the read-function in an *InfiniBand* node. The RQ is part of a complete QP, but the SQ is omitted for the sake of simplicity.

An important requirement for this node-type was that it should be compatible with the original node-type interface; or at least that the changes would be minimal. Hence, in order to acquire pointers to samples from the super-node, the `*smps[]` parameter from the read-function is used. Like the super-node in Figure 3.2a, the super-node in Figure 3.4a starts by allocating $cnt \geq 1$ empty samples, increasing their reference counters, and passing their pointers to the node's read-function. The node, in turn, takes the addresses of the samples, wraps them up in scatter/gather elements, places them in work requests, and submits them to the RQ. Now, when the HCA receives a message, it will write the data directly into the allocated memory within the super-node. In this way, an additional copy between the node and the super-node is avoided.

Since the receive module of an *InfiniBand* node does not copy data to the passed samples, the returning of function in Figure 3.4b works fundamentally different from the returning of the function in Figure 3.2b. If there are no CQEs in the completion queue, thus if the HCA did not receive any data, the return value *ret* of the node shall be 0. In that way, the super-node knows that the set of previously allocated `smps[]` does not hold any data. The reference counters of none of the buffers shall be decreased since they are all submitted to the RQ and the HCA will thus write data to them.

If CQEs are available, pointers to samples which are submitted to the RQ (light gray in Figure 3.4) are replaced by the pointers to the buffers that are filled by the HCA (dark gray in Figure 3.4). The return value *ret* shall be the number of pointers that have been replaced since these buffers now contain valid data that was sent to this node. The reference counters of these buffers must be decreased after they have been processed by the super-node.

Consequently, in order for the *InfiniBand* node to be able to receive data, the super-node has to invoke the read-function at least once without acquiring any data. To store the pointers to the buffers in the CQEs, the WR C structure member `wr_id` can be used (see subsection 2.3.1).

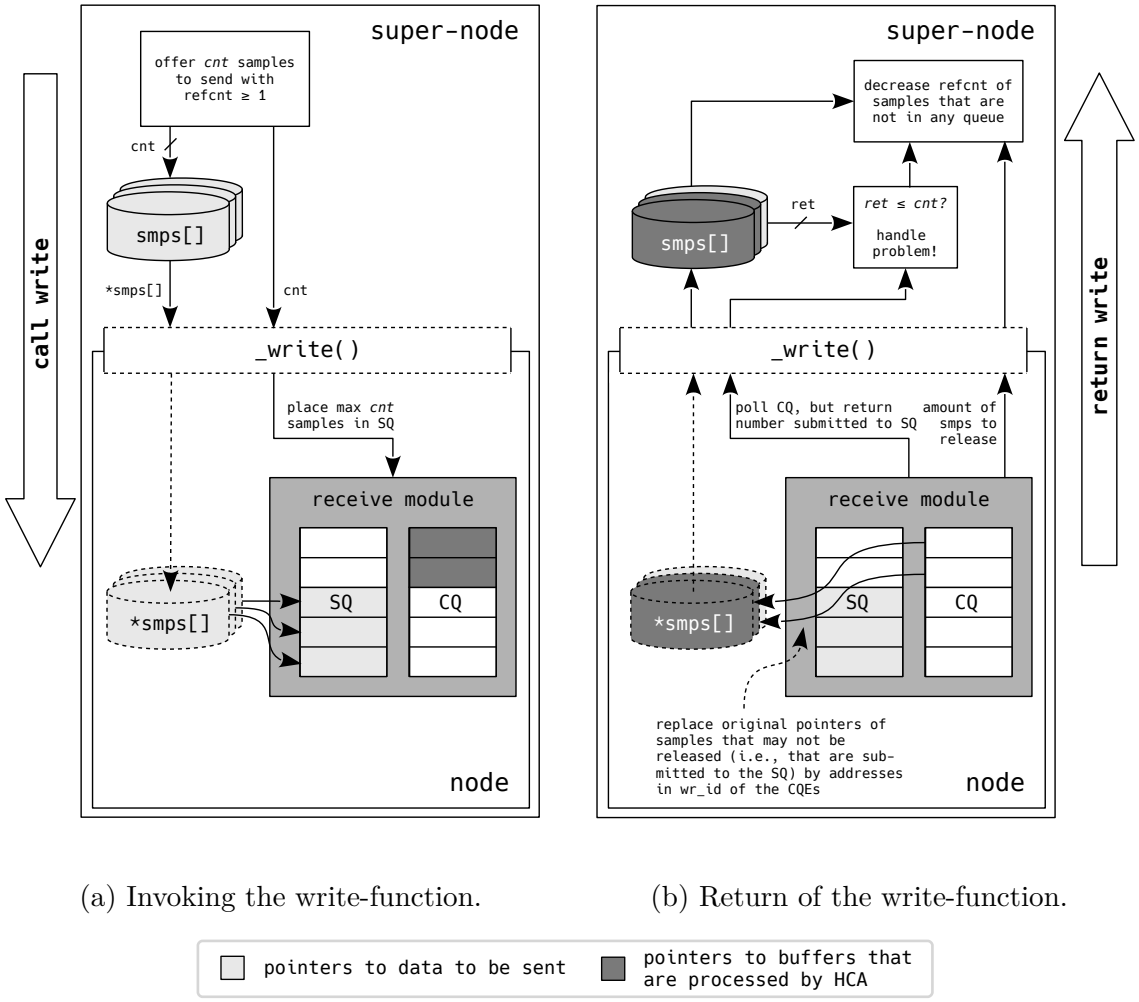
Write-function The write-function, depicted in Figure 3.5, has to adhere to similar conventions as the read-function in order to realize zero-copy. Again, the addresses of the samples are passed to the node as arguments of the write-function, to be subsequently submitted to the SQ. The HCA will then process the submitted work requests and take care of the necessary memory operations.

When the pointers are successfully submitted to the SQ, the function shall return the total number of submitted pointers *ret*. If the completion queue is empty, none of these pointers may be released because the HCA has yet to access the memory locations. If the completion queue contains entries, that means that previously submitted send WRs are finished; these pointers can be released. So, in order to release them, the initial pointers to the data to be sent (light gray in Figure 3.5) are replaced by pointers to buffers which were submitted to the SQ in a previous call of the write-function. The super-node has to be notified that it must only decrease the reference counter of pointers that were yielded by the CQEs.

3.3.3 Proposal for a new read- and write-function

Apparently, the major shortcoming of the functions from Listing 3.2 is the lack of an interface to pass the number of samples to be released to the super-node. There is no way the super-node can predict how many samples may be released; this becomes even more difficult if it is taken into account that some samples may be sent inline—thus can be released immediately after submitting the WR—and that some work requests may not be successfully submitted to the SQ.

Therefore, new parameters for the read- and write-function are proposed in Listing 3.3. The additional parameter in each function lets a node decide how many



(a) Invoking the write-function. (b) Return of the write-function.

Figure 3.5: A depiction of the working principle of the write-function in an *InfiniBand* node. The SQ is part of a complete QP, but the RQ is omitted for the sake of simplicity.

items of `smps[]` should actually be released. The several distinctions which have to be considered are further elaborated upon in section 4.2.

```
1 int (*read)(struct node *n, struct sample *smps[], unsigned cnt,
2              unsigned *release);
3
4 int (*write)(struct node *n, struct sample *smps[], unsigned cnt,
5               unsigned *release);
```

Listing 3.3: Proposal for an additional parameter in `read()` and `write()`.

3.4 Memory management

Originally, memory that was allocated within the framework could be allocated with a fixed set of settings called *memory-types*. The VILLASnode internal `alloc()` could be called, for example, with `memory_hugepage`, which pins memory and maps it to hugepages (see subsection 2.4.1), or with `memory_heap`, which allocates aligned memory on the heap. These embedded memory-types are not sufficient for the *InfiniBand* node-type. Subsection 3.3.2 already showed that the HCA will directly access the memory that is allocated by the super-node. Thus, as follows from subsection 2.2.7, the buffer must be registered with a memory region and the WRs that are submitted to either queue of the QP must contain the local key.

Since embedding a memory-type for every node-type in the VILLASnode source code would go against the principle of modularity, this is not an option. Consequently, the most obvious solution is to allow every node-type to register its own memory-type if necessary. In that way, every node-type can exactly define what the `alloc()` and `free()` functions implement. For `alloc()`, a node-type can, for example, define how memory should be allocated, whether the pages should be aligned, how big the pages should be, and if the memory should be registered with a memory region. It is also possible for a node-type to implement certain functions which interact with the memory that is allocated by the memory-type; this can, for example, be used within the *InfiniBand* node to acquire the local key of a sample that is passed as an argument of the read- or write-function.

With this method, every node-type may define a `memory_type` C structure, which it must register in the same fashion as it registers the interface functions with the super-node (line 39, Listing D.3). By enabling node-types to register their own memory-type, the super-node knows what type of memory to use for input and/or output buffers that are connected to nodes of this type ($q_{i,x}$ and $q_{o,x}$ in Figure 3.1).

If no memory-type is specified, the super-node will assume `memory_hugepage`.

3.5 VILLASnode finite-state machine

Initially, a node could reside in one of the six states displayed in Listing 3.4. The super-node transitions the node through the states depending on the results of functions from appendix C. E.g., when the super-node calls a node's start-function, the transition *checked* $\rightarrow$ *started* is performed if the function returns successfully.

These states were sufficient for the node-types which existed up to now (Table 1.1); when a node resided in *started*, this meant it was ready to send and receive data. This is not the case for node-types that are based (descendants of) the Virtual Interface Architecture. Here, a node can be initiated—for which the *started* state can be used—but not connected and thus not able to send data to another node. Accordingly, the introduction of a new state *connected* would be appropriate. Furthermore, architectures that are based on the VIA rely on descriptors (called work requests in the IBA) in a send and receive queue. Hence, in order to be able to receive data

```
enum state {
        STATE_DESTROYED         = 0,
        STATE_INITIALIZED       = 1,
        STATE_PARSED            = 2,
        STATE_CHECKED           = 3,
        STATE_STARTED           = 4,
        STATE_STOPPED           = 5
};
```

Listing 3.4: The six states a node could originally reside in.

directly after the connection has been established, descriptors have to be present in the RQ at this moment. For this reason, in (descendants of) the VIA, it is possible to prepare elements in the receive queue prior to the actual connection.

These considerations yield the finite-state machine in Figure 3.6. The states which are indicated with dashed borders, *pending connect* and *connected*, may be set by the node after the super-node transitioned the instance to the *started* state. The use of both states is not mandatory. If a node is in one of these two states, the super-node interprets it as were the node in the *started* state. But, they can be used within the node itself to distinguish between a node being started, being in a pending connect state, or actually being connected. This state machine shows similarities with the VIA's finite-state machine in Figure 2.2. It can therefore be used for future node-types that are based on the VIA—other than the *InfiniBand* node-type that is presented in the present work—as well.

Although it is necessary to execute the transition *checked* → *started*, it is possible to transition to *stopped* and *destroyed* from any of the three states in the dashed square.

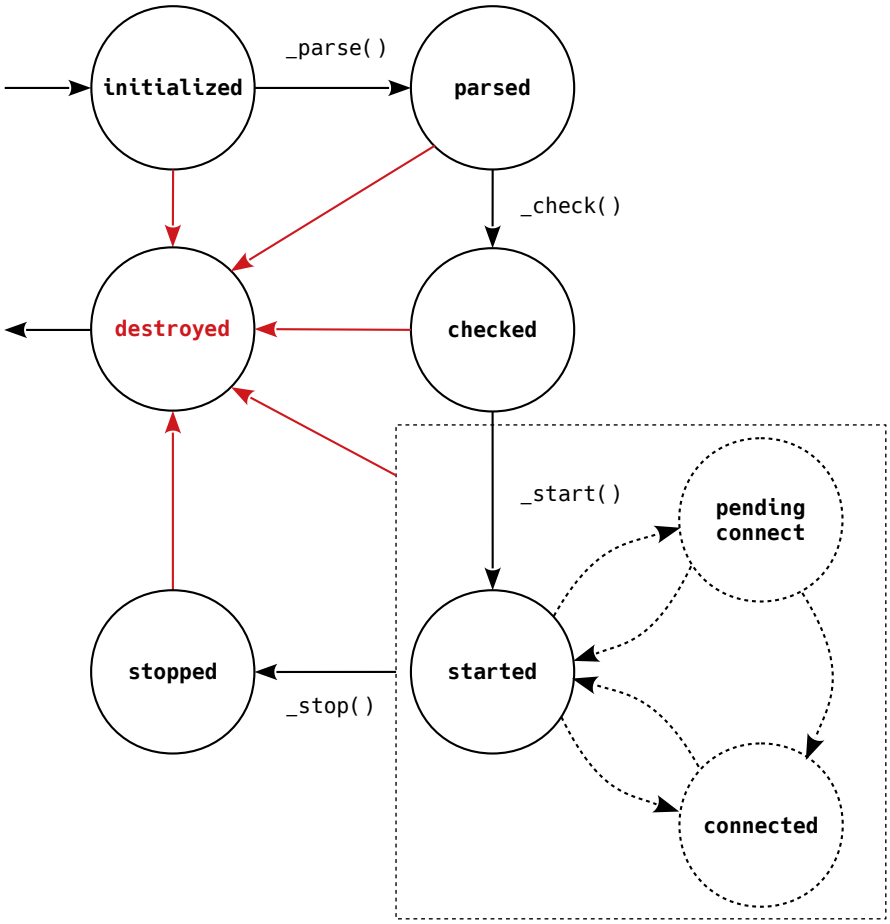


Figure 3.6: The VILLASnode state diagram with the two newly introduced states *pending connect* and *connected*.

4 Implementation

The first section of this chapter (4.1) describes the implementation of the benchmark which was used to measure latencies between InfiniBand host channel adapters. Then, section 4.2 describes how the *InfiniBand* node-type for VILLASnode was implemented. Subsequently, section 4.3 describes the characteristics and implementation of the benchmark that was used to analyze VILLASnode node-types. Thereafter, section 4.4 describes how UC support was added to the RDMA CM library. Finally, section 4.5 briefly describes what tools and techniques were used to process and analyze the acquired data.

If not stated otherwise, all software that is discussed in this chapters is written in the C programming language [KR78].

4.1 Host channel adapter benchmark

The developed host channel adapter benchmark was inspired by the measurements which were done by MacArthur and Russel [MR12], which were already presented in section 1.2. Although this work will likewise analyze the influence of variations in operation modes, settings, and message sizes on latencies, it will not focus on their influence on throughput.

The objective of this benchmark is to measure—as accurately as possible—how long data resides in the actual InfiniBand Architecture when it is sent from one host channel adapter to another host channel adapter. So, if latency is defined as

$$t_{lat} = t_{subm} - t_{recv}, \tag{4.1}$$

the time data actually spends in the IBA can be approximated by setting t_{subm} to the moment on which the send WR is submitted, and t_{recv} to the moment the receive node becomes aware of the CQE in the completion queue which is bound to the receive queue.

Subsection 4.1.1 first introduces how and where in the source code the timestamps t_{subm} and t_{recv} are measured. Then, subsection 4.1.2 describes what tests the benchmark is capable of running.

4.1.1 Definition of measurement points

Many benchmarks factually measure the latency of the round-trip and divide it by two in order to approximate the one-way time between two host channel adapters. This is necessary if the HCAs are not part of the same host system. The latency of

messages between InfiniBand HCAs is usually under 5 µs; there are even reports of one-way times as small as 300 ns [MR12]. Hence, if both HCAs are part of different systems, even small deviations between the endnodes' system clocks could cause significant skews in t_{lat} and make the results useless. This problem is nonexistent if both timestamps t_{subm} and t_{recv} are generated by the same system clock.

A possible disadvantage of using the round-trip delay to approximate the one-way delay is the additional (software) overhead. Lets assume that a message is sent from node A to node B, back to node A. Then, there can be an additional time penalty which is introduced by software on node B, that is necessary to submit a work request in order to return the received message.

Furthermore, it is possible that latency benchmarks—e.g., `ib_send_lat` and `ib_write_lat` in the OFED™ Performance Tests[1]—yield distorted, possibly idealized, results. Although these are well suited for hardware and software tuning, the results can deviate from the actual latencies that can be seen when implementing an application with the OFED™ verbs.

The present work therefore implements a custom benchmark that assumes two HCAs in the same host system. It thereby prevents the skewness that is caused by deviations between different endnodes's system clocks and the additional software overhead for round-trip delays. Furthermore, it makes sure that the latencies correspond to the latencies that can be seen in actual applications.

Generation of timestamps In this benchmark, `clock_gettime()` [Ker10] is used to generate timestamps. Its parameters are a variable of type `clockid_t` and a reference to an instance of `struct timespec` (Listing 4.1) to which the function will write the current time.

```
1 struct timespec {
2         time_t   tv_sec;        /* seconds */
3         long     tv_nsec;       /* nanoseconds */
4 };
```

Listing 4.1: The composition of `struct timespec`.

The former parameter, `clockid_t`, is particularly interesting. Usually, this is set to `CLOCK_REALTIME`, on which `clock_gettime()` returns the system's best guess of the current time. This means that this clock can change during operation because it is adapted by the *Network Time Protocol* (NTP). Therefore, this timestamp is not suitable for the calculation of time differences with a nanosecond resolution. However, if the `CLOCK_MONOTONIC` is requested, `clock_gettime()` will return a strictly linearly increasing timestamp starting at an unspecified point in the past. Since linearity is guaranteed between timestamps for this `clockid_t`, it is best suited to calculate t_{lat} from equation (4.1).

[1] https://github.com/linux-rdma/perftest

Location of timestamps in code This benchmark takes timestamps on three different locations in the code:

- t_{subm} is acquired right before an already prepared work request is submitted to the send queue with `ibv_post_send()`. The timestamp will be the message's payload. For that reason, it is important that the address to which the scatter/gather element points is valid until the message is actually send and that the timestamp is not overwritten in a next iteration. The pseudocode for this case is displayed in Listing 4.2.
- t_{recv} is measured on the receiving node. It is acquired right after `ibv_poll_cq()` on the completion queue that is bound to the receive queue returns with a positive value. The pseudocode for this case is displayed in Listing 4.3.

  The function that is displayed in Listing 4.3 lies in the datapath, and the moment on which the timestamp and the identifier of the message are saved to be evaluated later, are time-critical. For one, this is optimized by using 2 MiB hugepages instead of conventional 4 KiB pages. For example, when 8000 messages are received, 8000 8-byte timestamps (64 KiB) and 8000 4-byte identifiers (32 KiB) must be saved. These 96 KiB fit into one single hugepage, however, it would require 24 conventional pages and in turn 24 potential page faults. For the sake of readability of the code, the timestamps and message identifiers are spread among two hugepages.

  Furthermore, it is made sure that the pages are immediately touched after initialization with `mmap()` to prevent page faults from happening in the datapath. After allocating the memory, the pages are locked with `mlockall()`. More information on memory optimization can be found in subsection 2.4.1.
- t_{comp} is measured in the same fashion as t_{recv}, but on the sending node. `ibv_poll_cq()` polls the completion queue that is bound to the send queue. It gives an indication of the time that passes before the sending node gets a confirmation that the message has been sent. Similar to equation (4.1), the latency before a confirmation of transmission is available can be defined as:

$$t_{lat}^{comp} = t_{subm} - t_{comp}. \tag{4.2}$$

  This timespan is relevant because buffers in the main memory cannot be reused as long as there is no confirmation that the HCA copied the data from the host's main memory to its internal buffers. (This is not the case for data that is sent inline, see subsection 2.3.1.)

There is one special case which has not been discussed yet. t_{subm} is set to the time right before the WR is submitted to the send queue. Since it will take a certain amount of time before the HCA will copy the data (i.e., the timestamp) from the host's main memory to its internal buffers, it is possible to continue to alter the value after the work request has been posted. This benchmark offers a

```
// 'int messages' represents the number of messages to be sent
struct timespec tp[messages];

for (int i = 0; i < messages; i++) {
    /**
     * Prepare WR with an sge that points to tv_nsec of tp[i].
     * By using an array of timespecs, it is guaranteed that
     * the timestamp will not be overwritten.
     */

    clock_gettime(CLOCK_MONOTONIC, &tp[i]);
    ibv_post_send();
}
```

Listing 4.2: Pseudocode which records the moment a messages is submitted to the Send Queue (SQ).

```
struct timespec tp;

while (1) {
    ibv_get_cq_event();  // Only necessary for event based polling

    while (ibv_poll_cq()) {
        clock_gettime(CLOCK_MONOTONIC, &tp);

        /**
         * Save tp and message identifier in an array and
         * return as soon as possible, so that as little
         * as possible time is lost before polling goes on.
         */
    }

    ibv_req_notify_cq(); // Only necessary for event based polling
}
```

Listing 4.3: Pseudocode which records the moment a Completion Queue Entry (CQE) becomes available in the Completion Queue (CQ).

function to measure t_{send}, which approximates the moment the HCA copies the data to its internal buffer. The delta

$$\Delta t_{inline} \approx \tilde{t}_{lat}^{send} - \tilde{t}_{lat}, \tag{4.3}$$

approximates the amount of time which will be saved by sending the data inline. In equation (4.3), $\tilde{t}_{lat}^{send}$ is the median latency measured with *send*-timestamps, and $\tilde{t}_{lat}$ is the median latency measured with *submit*-timestamps. The pseudocode of Listing 4.2 must be replaced with the pseudocode of Listing 4.4 to transmit t_{send} instead of t_{subm}.

```
// Global variable
struct timespec tp;

void * t_function(void * ctx)
{
    while (1) {
        clock_gettime(CLOCK_MONOTONIC, &tp);
    }

    return NULL;
}

pthread_t t_thread;
pthread_create(&t_thread, NULL, t_function, NULL);

// 'int messages' represents the number of messages to be sent
for (int i = 0; i < messages; i++) {
    /**
     * Prepare WR with sge that points to tp.tv_nsec. It will
     * continue to change since the thread continues to run in the
     * background.
     */

    // No need to invoke clock_gettime() here
    ibv_post_send(); // Post prepared WR
}
```

Listing 4.4: Pseudocode which continues to update an instance of the `timespec` C structure in a separate thread, whilst a pointer to this instance has already been submitted to the Send Queue (SQ).

4.1.2 Supported tests

The list below provides an overview of the different settings that can be applied. Later, section 5.1 will present the results for different combinations of these settings.

- **The service type** (Table 2.1) can be varied between RC, UC, and UD.
- **The poll mode** (Figure 2.21) can be set to *busy polling* or *wait for event*. The poll mode can be set independently for t_{recv} and t_{comp}.
- **Inline mode** (subsection 2.3.1) can be turned on for small messages.
- **Unsignaled completion** can be enabled. When this switch is set, send WRs will not generate WQEs when the HCA has processed them.
- **The operation** (Table 2.5) can be set to *send with immediate* or *RDMA write with immediate*. Both operations are only supported *with immediate* in

this benchmark since the ImmDt header is used to identify the order of the messages at the receive side.

- **The burst size** represents the number of messages that will be sent during one test and is limited to the maximum size of a QP in the HCA. The benchmark is built in a way that it will continuously send messages, until this value is reached. It can be varied between 1 and 8192.
- **An intermediate pause** (in nanoseconds) can be set. The benchmark will sleep for this amount of time in between the `ibv_post_send()` calls.
- **Either the send or submit time** can be measured. This switch determines whether t_{subm} or t_{send} is measured.
- **The message size** S_M can be set to

$$S_M = 8 \cdot 2^i \,\mathrm{B},\ i \in [0, 12], \tag{4.4}$$

where 8 B is the minimum size of a message with a timestamp. A maximum of 32 768 B (32 KiB) is chosen because messages in VILLASnode are unlikely to be bigger than 32 KiB.

Although the possibility to submit linked lists of scatter/gather elements and work requests to the send queue will be used in the VILLASframework *InfiniBand* node-type, its influence on latency will not be examined in this benchmark. Linking scatter/gather elements can become handy if data from different locations in the memory must be sent. Submitting combined work requests can be convenient if a whole batch of WRs has to be posted and it is not necessary that a WR is posted immediately after its generation (e.g., creating a set of receive WRs in a loop and posting the linked list right after the closing bracket of the loop). However, the lowest latency is achieved by passing only one memory location to the HCA and by sending a message immediately after generation of the timestamp.

4.2 VILLASframework InfiniBand node-type

Chapter 3 already introduced the architecture of node-types in VILLASframework and concepts to enable compatibility of VIAs—and in particular the IBA—with VILLASframework. The key objective of the development of an *InfiniBand* node-type was the implementation of all functions in appendix C with as little as possible alterations to the pre-existing architecture. Other than the proposed changes from subsection 3.3.3, the VILLASframework architecture was not modified with regards to the node-type interface and the memory management.

The implementation of the more apparent functions, e.g., `parse()`, `check()`, `reverse()`, `print()`, `destroy()`, and `stop()`, will not be discussed. This section mainly focuses on non-obvious functions, which are either InfiniBand specific (i.e., the start-function in subsection 4.2.1) or had to be optimized to make full use of the

kernel bypass InfiniBand offers (i.e., the read- and write-functions in subsection 4.2.3 and 4.2.4, respectively). The complete source code of the *InfiniBand* node-type can be found on VILLASnode's public Git repository.[2]

4.2.1 Start-function

After a configuration file, which is set by a user, is interpreted by the parse-function and reviewed by the check-function, the super-node will invoke the start-function to initialize all necessary structures. It starts with the creation of a communication event channel with `rdma_create_event_channel()` and the initialization of an RDMA communication identifier with `rdma_create_id()`. The latter is bound to both a local InfiniBand device that was defined in the configuration file and the event channel.

Before the node allocates the protection domain with `ibv_alloc_pd()`, the communication identifier tries to resolve the remote address with `rdma_resolve_addr()` (in case of an active node) or places itself into a listening state with `rdma_listen()` (in case of a passive node). Whether the node becomes an active or passive node depends on the presence of a remote host address to connect to in the configuration file. Finally, the start-function creates a separate thread with `pthread_create()` [Ker10] to monitor all asynchronous events on the `rdma_cm_id`.

When everything is set up successfully, the start-function will return 0, to indicate success. The super-node then moves the node to the *started* state (Figure 3.6).

4.2.2 Communication management thread

The function that is executed by the thread that is spawned by the start-function is kept busy by a while loop until the node is moved to the *started* state. This avoids races and ensures that the state transitions from Figure 3.6 are obeyed.

The remainder of this function consists of a while loop that monitors the communication identifier in a blocking manner with `rdma_get_cm_event()` (subsection 2.3.3). Within this loop, the different events are handled by a switch statement. The loop, the switch statement, and a short description of what happens for every case are displayed in Listing 4.5. Before expanding on the different operations of every case, a note on the blocking characteristics of `rdma_get_cm_event()` has to be made. This function enables the OS to suspend further execution of the thread for an indefinite amount of time, which usually results in difficulties when trying to cancel (or kill) the thread. However, `read()`, which lies at the heart of `rdma_get_cm_event()`, is a required cancellation point. A thread, for which cancelability is enabled, only acts upon cancellation requests when it reaches a cancellation point [Ker10]. Furthermore, as defined in IEEE Std 1003.1™-2017 [18a]: "[when] a cancellation request is made with the thread as a target while the thread is suspended at a cancellation point, the thread shall be awakened and the cancellation

[2] https://git.rwth-aachen.de/acs/public/villas/VILLASnode/

request shall be acted upon." Thus, even though the thread is suspended, it can be canceled with `pthread_cancel()` if necessary.

Active node As defined in the previous subsection, an active node is a node that tries to connect to another node. The first event that should appear after the start-function has been called is `RDMA_CM_EVENT_ADDR_RESOLVED`. This event denotes that the address has been resolved and that the QP and two CQs—one for the receive and one for the send queue—can be created. These instances are created using `rdma_create_qp()` and `ibv_create_cq()`, respectively. It is important for the functioning of the *InfiniBand* node-type's write-function (subsection 4.2.4) that the QP's initialization attribute `sq_sig_all` is set to `'0'`.

After all necessary structures have been initialized, `rdma_resolve_route()` will be invoked. Then, when the route has successfully been resolved, the event channel will unblock again and return `RDMA_CM_EVENT_ROUTE_RESOLVED`. This means that everything is set up, and `rdma_connect()` may be called to invoke a connection request. The state of the active node is then set to *pending connect*.

When the remote node accepts the connection, `RDMA_CM_EVENT_ESTABLISHED` occurs and the state of the node is set to *connected*.

If the node operates with the UD service type, the last mentioned event structure contains the Address Handle (AH), which includes information to reach the remote node. This value is saved because in UD mode it has to be defined in every work request (subsection 2.3.1). Although the node is not really connected—after all UD is an unconnected service type—the node will be transitioned to the *connected* state. In the context of VILLASnode, this state implies that data can be send, either because the QPs are connected or because the remote AH is known.

Passive node As mentioned before, a passive node listens on the communication identifier and waits until another node reaches out to it. If another node calls `rdma_connect()` on it, the channel will unblock and return the event `RDMA_CM_EVENT_CONNECT_REQUEST`. Thereon, the node will build its QP, its CQs, and accept the connection with `rdma_accept()`. If the service type of the node is a connected service type (i.e., UC or RC), the node will move to the *pending connect* state. If the service type is unconnected (i.e., UD), it will move directly to the *connected* state.

In case of a connected service type, the `RDMA_CM_EVENT_ESTABLISHED` event occurs when the connection has successfully been established. The state is then set to *connected*.

Error events Error events which are caused because a remote node could not be reached are not necessarily fatal for the complete node. In this case, a fallback function which sets the node into the listening mode instead of the active mode will be invoked. This behavior is configurable; if a user sets the appropriate flag in the configuration file these errors can be made fatal.

```
struct rdma_cm_event *event;

while (rdma_get_cm_event(event_channel, &event) == 0) {

    switch (event->event) {
        case RDMA_CM_EVENT_ADDR_RESOLVED:
            // Create QP, receive CQ, and send CQ.
            // Call rdma_resolve_route()
            // State: STARTED
        case RDMA_CM_EVENT_ADDR_ERROR:
            // Try fallback and set mode rdma_cm_id to listening
            // State: STARTED
        case RDMA_CM_EVENT_ROUTE_RESOLVED:
            // Call rdma_connect()
            // State: PENDING_CONNECT
        case RDMA_CM_EVENT_ROUTE_ERROR:
            // Try fallback and set mode rdma_cm_id to listening
            // State: STARTED
        case RDMA_CM_EVENT_UNREACHABLE:
            // Try fallback and set mode rdma_cm_id to listening
            // State: STARTED
        case RDMA_CM_EVENT_CONNECT_REQUEST:
            // Create QP, receive CQ, and send CQ.
            // Call rdma_accept()
            // State: PENDING_CONNECT
        case RDMA_CM_EVENT_CONNECT_ERROR:
            // Try fallback and set mode rdma_cm_id to listening
            // State: STARTED
        case RDMA_CM_EVENT_REJECTED:
            // Try fallback and set mode rdma_cm_id to listening
            // State: STARTED
        case RDMA_CM_EVENT_ESTABLISHED:
            // In case of UD, save address handle from event struct
            // State: CONNECTED
        case RDMA_CM_EVENT_DISCONNECTED:
            // Release all buffers and destroy everything
            // State: STARTED
        case RDMA_CM_EVENT_TIMEWAIT_EXIT:
            break;
        default:
            // Error message: unkown event
    }

    rdma_ack_cm_event(event);
}
```

Listing 4.5: The events that are monitored by the communication management thread. Although not explicitly stated in this listing, every case block ends with a `break`.

4.2.3 Read-function

This subsection focuses on the implementation of the read-function which was previously proposed in section 3.3. Contrary to the functioning principle in Figure 3.2, which suggests that all samples that are passed to the read-function will definitely be submitted and must thus be held, there is a chance that some samples will not be submitted successfully. These samples must be released again.

Figure 4.1 shows a decision graph for the algorithm that is implemented by the read-function. The example case, depicted by the red path, assumes that 5 empty samples are passed to the read-function, and that there are at least *threshold* WQEs in the RQ. This threshold, which is set in the configuration file, is necessary to ensure that a node can always receive samples because there are always at least *threshold* pointers in the RQ. If this threshold has not yet been reached, all passed samples are submitted to the receive queue and `*release` is set to 0 (depicted by the black path). Then, the function returns with $ret = 0$, without ever polling the completion queue.

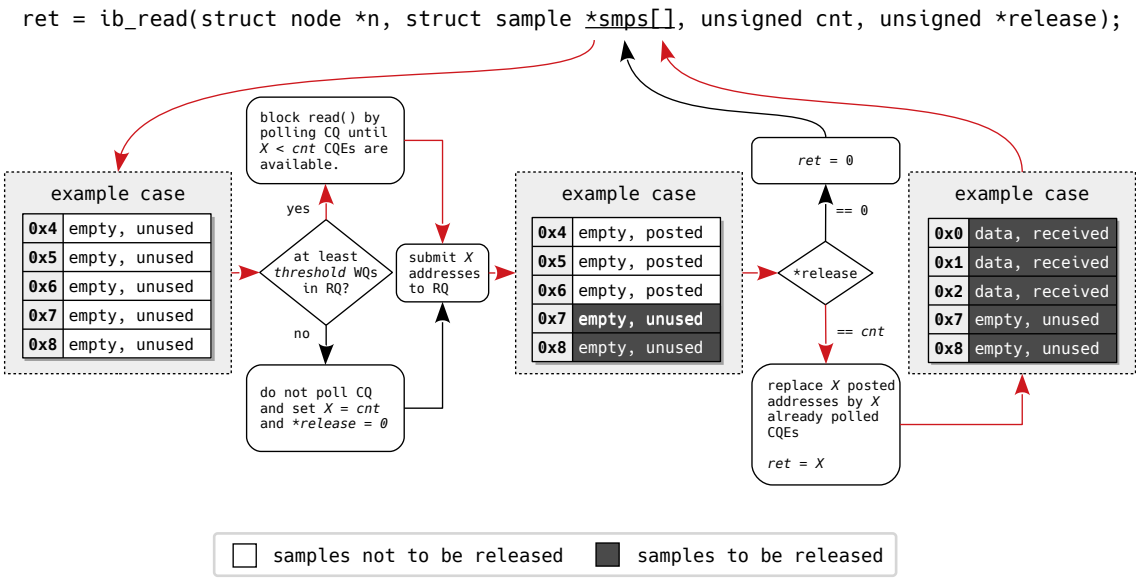


Figure 4.1: The decision graph for the read-function in the *InfiniBand* node. Prior to invoking the read-function, `*release` is always set to `cnt` by the super-node.

If the threshold has been reached, the red path is followed. The completion queue is polled in a while loop until at least one, but not more than `cnt`, CQEs are available. This will block further execution the read-function, which is the intended behavior. After all, when a certain amount of WQEs resides in the RQ, it is undesired to continue to submit new WRs. At a certain moment, the queue would be full and it would not be possible to submit new addresses that the node got from the super-node. However, this is necessary to free places in `*smps[]`, which can only hold up

to `cnt` values. So, if this blocking behavior was not in place, the super-node would keep passing new addresses until the receive queue would overflow and no addresses from CQEs could be returned to the super-node anymore.

Because `ibv_poll_cq()` does not rely on any of the system calls that are listed in [18a], the thread that contains this while loop would not notice if a cancellation request is sent. Therefore, `pthread_testcancel()` [Ker10] should regularly be called within this loop.

The addresses in `*smps[]` are not immediately swapped with the X addresses that are returned with `ibv_poll_cq()`. First, after the poll-function has indicated that X CQEs with addresses are available, X addresses from `*smps[]` are submitted to the RQ. This ensures that the RQ does not drain and it makes room for the addresses from the CQEs in `*smps[]`. Finally, the polled addresses are swapped with the addresses that were posted to the receive queue, and the read-function returns with $ret = X$. Note that `*release` remains untouched: all values in `*smps[]` are either received or not used, and must thus be released.

4.2.4 Write-function

The write-function, depicted in Figure 4.2, is a bit more complex than the read-function. This time, the algorithm depicted in the decision graph includes four example cases.

Immediately after the write-function is invoked by the super-node, it tries to submit all `cnt` samples to the SQ. While going through `*smps[]`, the node dynamically checks whether the data can be sent inline (subsection 2.3.1) and whether an AH must be added. The node has to distinguish among four cases:

- the samples will be submitted normally and may thus not be released by the super-node until a CQE with the address appears;
- the samples will be submitted normally, but some samples will be immediately marked as *bad* and must thus be released by the super-node;
- the samples will be sent inline and, because the CPU directly copies them to the HCA's memory, must thus be released by the super-node;
- an arbitrary combination of all abovementioned cases.

For samples that are sent normally, the WR's `send_flags` (Listing 2.3) must be set to `IBV_SEND_SIGNALED`. These samples may only be released after the HCA has processed them, which must not necessarily be in the same call of the write-function. The only way for the HCA to let the node know that it is done with a sample, is through a completion queue entry. Since the QP is created with `sq_sig_all=0`, the generation of CQEs for samples must explicitly be requested.

When a sample is sent inline, `send_flags` must only be set to `IBV_SEND_INLINE`. It is not desired to get a CQE for an inline WR since it can be—and thus will be—released immediately after being submitted to the SQ. After all, it is not possible to release a sample twice.

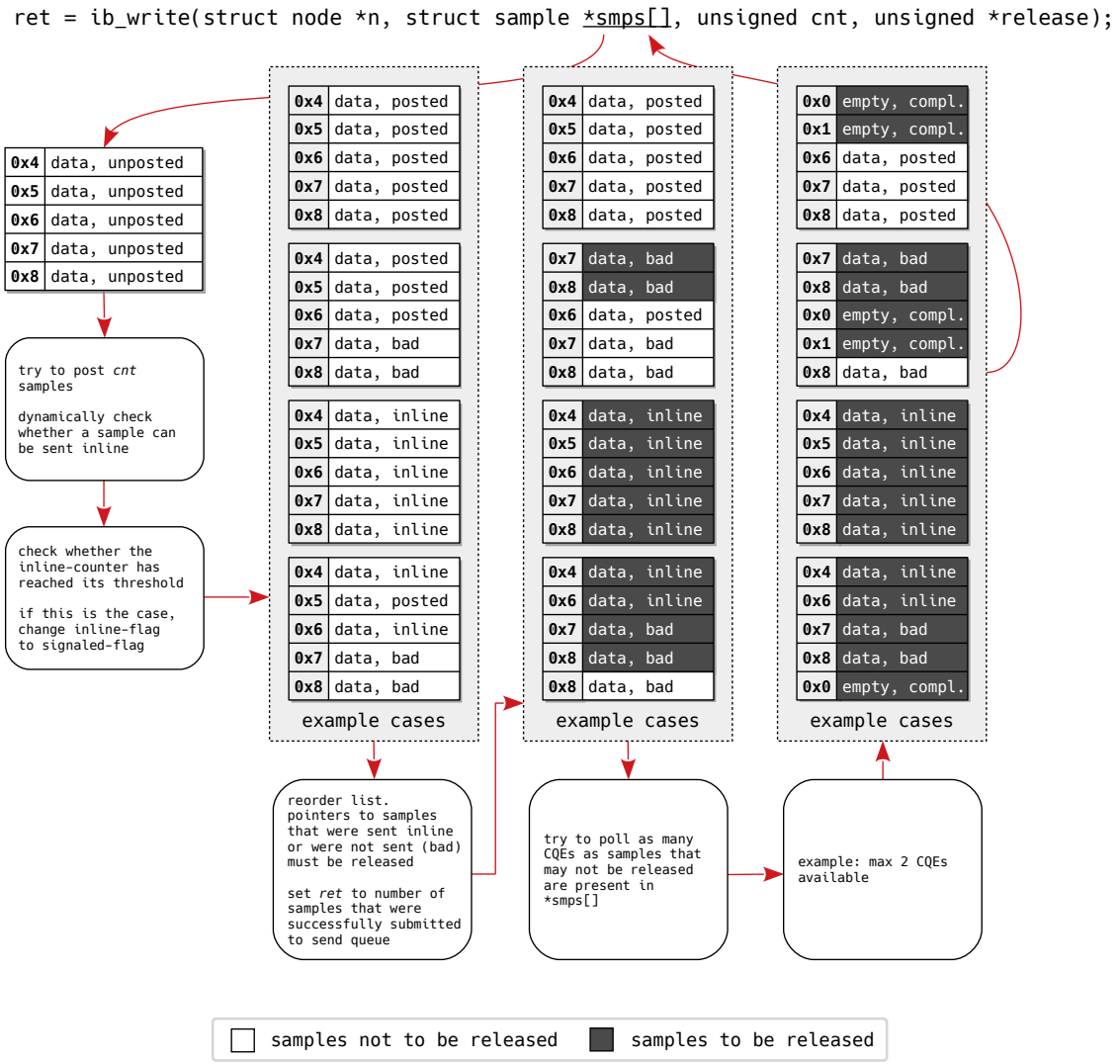


Figure 4.2: The decision graph for the write-function in the *InfiniBand* node. Prior to invoking the write-function, `*release` is always set to `cnt` by the super-node.

There is one exception to this, however. Although no notifications will be generated if the *signaled* flag is not set, the send queue will start to fill up nonetheless. Therefore, when a lot of subsequent WRs are submitted with the *inline* flag set, occasionally a WR with the *signaled* flag must be submitted. For this reason, the write-function contains a counter which, when reaching a configurable threshold, changes an `IBV_SEND_INLINE` to an `IBV_SEND_SIGNALED`.

When all samples have been submitted to the SQ, the value *ret*, which will be returned to the super-node when the write-function returns, is set to the total number of samples that were successfully posted to the send queue.

Now, because the node can only use `*release` to communicate how many samples to release, `*smps[]` must be reordered. All samples that must be released, i.e., samples that were not successfully submitted to the send queue or samples that were sent inline, must be placed at the top of the list.

In the next step, the write-function shall try to poll

$$C_{poll} = \texttt{cnt} - C_{release} \tag{4.5}$$

CQEs, which corresponds to the number of places in `*smps[]` that are still free. Here, `cnt` is the total number of samples in `*smps[]` and $C_{release}$ the number of samples that have already been marked to be released when the write-function returns. It is certain that all addresses that return from the CQ must be released, since samples that were sent inline will not generate a CQE.

4.2.5 Overview of the InfiniBand node-type

Figure 4.3 summarizes all components in the VILLASnode *InfiniBand* node-type. Every component that is marked with an asterisk is listed in Table 4.1. Here, the sections that describe the respective basics (chapter 2), architecture (chapter 3), and implementation (chapter 4) are summarized.

Table 4.1: *InfiniBand* node-type components from Figure 4.3 and the respective sections of the present work that elaborate upon these components.

Component	Basics	Architecture	Implementation
HCA	subsection 2.2.1		
Queue pair	subsection 2.2.2		
Protection domain	subsection 2.2.7		
Event channels	subsection 2.3.2		
Communication identifier	subsection 2.3.3		
Buffers	subsection 2.2.7	section 3.4	
VILLASnode		section 3.1	
Read-function		section 3.3	subsection 4.2.3
Write-function		section 3.3	subsection 4.2.4
Start-function			subsection 4.2.1
Management thread			subsection 4.2.2

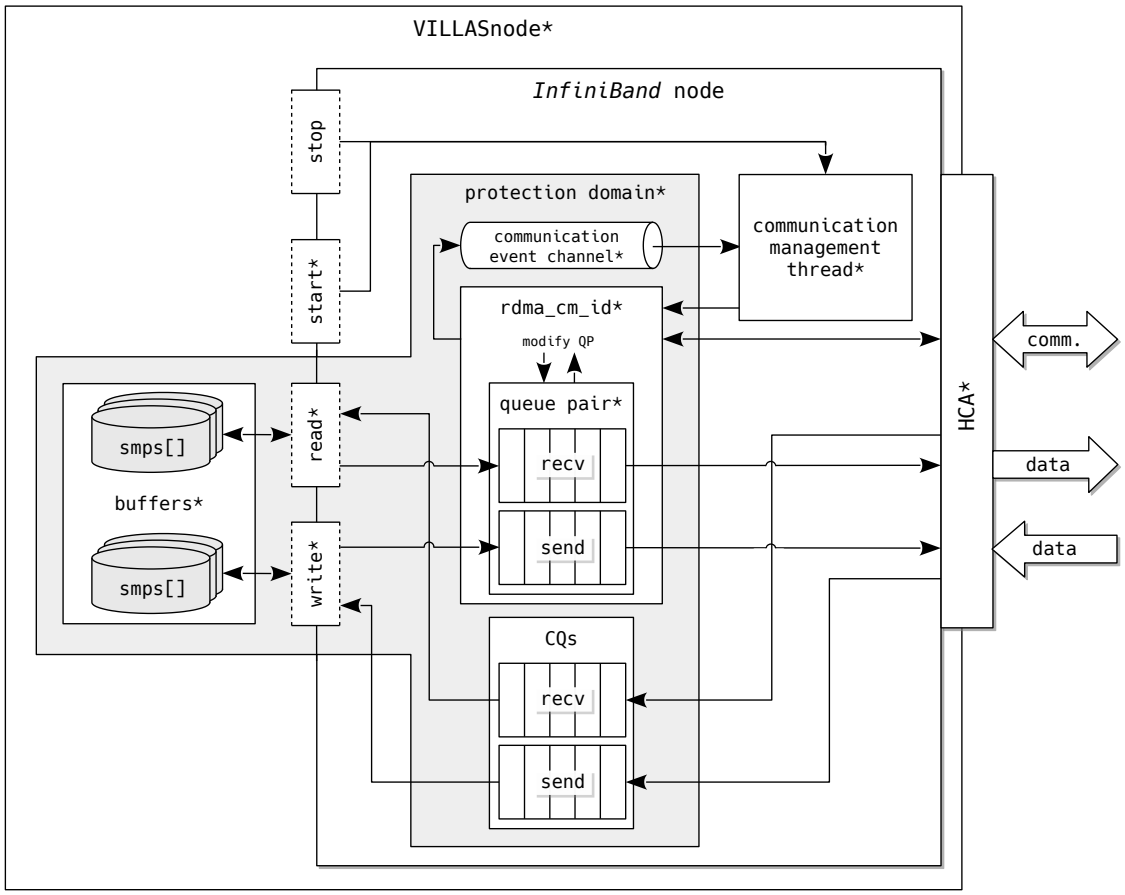


Figure 4.3: An overview of the VILLASnode *InfiniBand* node-type and its components.

4.3 VILLASnode node-type benchmark

The VILLASnode node-type benchmark is intended to compare different node-types with each other. The structure of the benchmark is depicted in Figure 4.4. The node-type under test could be, for example, the *InfiniBand* node-type. The benchmark is completely based on existing mechanisms within VILLASnode.

First, a *signal* node generates samples which, as aforementioned, also include timestamps. These samples are then sent to a *file* node, which in turn writes them to a *comma-separated values* (CSV) file, here called *in*. Simultaneously, the samples are sent to a sending instance of the node-type that is being tested. Eventually, a receiving instance of that node-type adds a receive timestamp and sends the samples to a second *file* node. This node writes the samples to a CSV file called *out*.

Although the *out* log file will contain both the generation timestamp and the receive timestamp, the *in* log file is necessary to monitor and analyze lost samples. This benchmark is meant to analyze the latencies of the different node-types, but also to discover their limits. Because it is possible that the signal generation misses

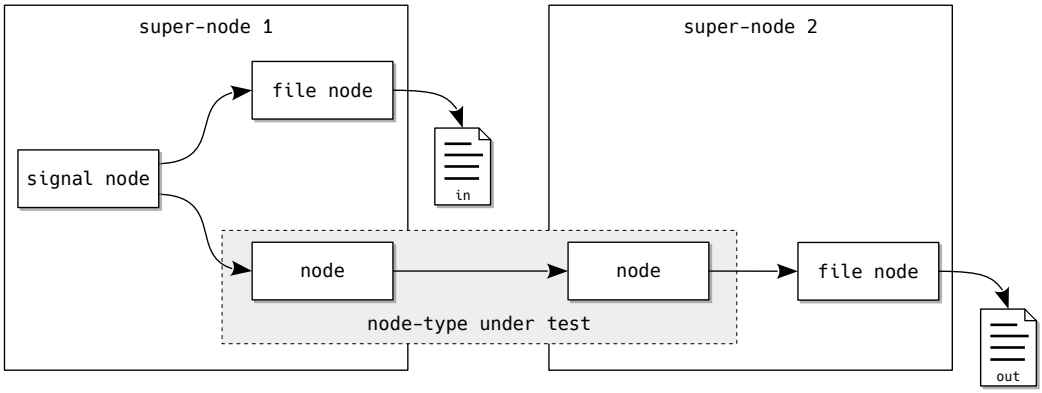


Figure 4.4: The VILLASnode node-type benchmark is formed by connecting a *signal* node, two *file* nodes, and two instances of the node-type that shall be tested.

steps at high frequencies (more on that in the next subsection), a missing sample in the *out* log file does not necessarily mean that something went wrong within the nodes that were tested. By comparing the *in* and *out* log file, the benchmark can decide which samples were missed by the *signal* node, and which samples were missed by the node that was tested.

4.3.1 Signal generation rate

In order for the benchmark to create an environment similar to the real use cases VILLASnode, the *signal* node must be time-aware and insert samples at a given rate. This injection rate of samples must be adjustable. Although this work only focused on rates between 100 Hz and 100 kHz, lower and higher rates are theoretically possible.

Listing 4.6 displays a simplified version of the *signal* node-type's read-function. When a super-node that holds a *signal* node tries to acquire samples from it, it calls its read-function. This function blocks further execution until a function `task_wait()` returns. Assuming that the super-node would usually call the read-function at an infinite high frequency, the wait-function ensures that it now only returns after a fixed amount of time.

The wait-function returns an integer `steps`, which indicates the number of steps between the timestamps of the samples. Lets assume that

$$t^{i+1}_{\texttt{task\_wait()}} > t^{i}_{sample} + \frac{1}{f_{signal}}\,\mathrm{s}, \tag{4.6}$$

when attempting to generate the sample with the timestamp t^{i+1}. Here, $t^{i+1}_{\texttt{task\_wait()}}$ is the moment `task_wait()` is called, t^{i}_{sample} the moment the last sample was generated, i the iteration of `signal_generator_read()`, and f_{signal} the frequency the *signal* node is set to. When the condition in equation (4.6) holds, `task_wait()`

cannot wait until t_{sample}^{i+1} since that time has already passed. Hence, the function must wait until

$$t_{sample}^{i+2} = t_{sample}^{i} + 2 \cdot \frac{1}{f_{signal}} \text{ s}, \tag{4.7}$$

in order to stay synchronized with the set frequency. Now, instead of 1 step, 2 timesteps have passed since the last call of the wait-function. In other words, 1 step is missed.

After the missed steps have been counted and the timestamp has been calculated, the actual samples are generated. These will be returned to the super-node through the `*smps[]` parameter of the read-function. This behavior is similar to that of the read-function of the *InfiniBand* node-type.

```
1  int signal_generator_read(struct node *n, struct sample *smps[],
2                            unsigned cnt, unsigned *release)
3  {
4    struct signal_generator *s = (struct signal_generator *) n->_vd;
5    struct timespace ts;
6    int steps;
7
8    /* Block until 1/p->rate seconds elapsed */
9    steps = task_wait(&s->task);
10
11   if (steps > 1 && s->monitor_missed) {
12     warn("Missed steps: %u", steps-1);
13
14     s->missed_steps += steps-1;
15   }
16
17   ts = time_now();
18
19   /**
20    * Generate sample(s) with signal and timestamp ts .
21    * Return this sample via the *smps[] parameter of
22    * signal_generator_read()
23    */
24 }
```

Listing 4.6: Simplified version of the read-function of the *signal* node-type.

This subsection will expand on two different methods to implement `task_wait()` and thus to monitor the rate with which samples are sent. Although the first method is the easier and preferred method, it does not work for high frequencies such as 100 kHz on which the *InfiniBand* node can operate. For these frequencies to work, the second method is introduced.

Timer expiration notifications via a file descriptor Linux provides an API for timers. The function `timerfd_create()` creates a new timer object and returns

a file descriptor that refers to that timer. Once the timer's period is set with `timerfd_settime()`, the file descriptor can be read with `read()` [Ker10].

Listing 4.7 shows the implementation of `task_wait()` with a Linux timer object. When `read()` is called on the timer's file descriptor (line 6, Listing 4.7), it will write the number of elapsed periods since the last modification of the timer or since the last read to `steps`. If no complete period has gone by when `read()` is called, the function will block until this is the case.

```
1  uint64_t task_wait(struct task *t)
2  {
3    int ret;
4    uint64_t steps;
5
6    ret = read(t->fd, &steps, sizeof(steps));
7
8    if (ret < 0)
9      return 0;
10
11   return steps;
12 }
```

Listing 4.7: Implementation of `task_wait()` by waiting on timer expiration notifications via a file descriptor.

Although Linux' API for timer notifications via a file descriptor offers a convenient way of keeping track of elapsed time periods, it is not suited for high-frequency signals. On the one hand, `read()` causes a system call which is relatively expensive since it causes a switch between user and kernel mode. On the other hand, the operating system is inclined to suspend the process when the read-function blocks. Since it takes a certain amount of time to wake up the process when a period has been elapsed, this can cause a potential timing violation for the next sample according to equation (4.6).

Busy polling the x86 Time Stamp Counter All x86 CPUs since the Pentium era contain a 64-bit register called *Time-Stamp Counter* (TSC). Since the Pentium 4 era, this counter increments at a constant rate which depends on the maximum core-clock to bus-clock ratio or the maximum resolved frequency at which the processor is booted [18c]. The nominal frequency can be calculated using:

$$f_{nominal}^{TSC} = \frac{\texttt{CPUID.15H.ECX}[31:0] \cdot \texttt{CPUID.15H.EBX}[31:0]}{\texttt{CPUID.15H.EAX}[31:0]}. \tag{4.8}$$

In his white paper [Pao10], Paoloni describes how the TSC can be used to measure elapsed time during code execution. In his work, the *Read Time-Stamp Counter* (RDTSC) and *Read Time-Stamp Counter and Processor ID* (RDTSCP) instructions

that are described in [18d] are used to read the TSC. Listing 4.8 shows the inline assembler that was used in VILLASnode to acquire the timestamp.

The functioning of both instructions is largely the same. After the `rdtsc`/`rdtscp` instruction is invoked, the 32 MSB of the timestamp are placed in `rdx` and the 32 LSB in `rax`. To get a valid 64-bit variable, `rdx` is shifted left 32 bit and subsequently disjuncted with `rax`. The resulting value is set as output variable `tsc`, which is also returned by both functions in Listing 4.8. During this operation, the high-order 32 bit of `rax`, `rdx`, and `rcx` are cleared. When hard-coded registers are clobbered as a result of the inline assembly code, this must be revealed up front to the compiler (line 12, Listing 4.8).

```
1  static inline uint64_t rdtsc()
2  {
3    uint64_t tsc;
4
5    __asm__ __volatile__(
6      "lfence;"
7      "rdtsc;"
8      "shl $32, %%rdx;"
9      "or %%rdx,%%rax"
10     : "=a" (tsc)
11     :
12     : "%rcx", "%rdx", "memory"
13   );
14
15   return tsc;
16 }
```

a: RDTSC.

```
1  static inline uint64_t rdtscp()
2  {
3    uint64_t tsc;
4
5
6    __asm__ __volatile__(
7      "rdtscp;"
8      "shl $32, %%rdx;"
9      "or %%rdx,%%rax"
10     : "=a" (tsc)
11     :
12     : "%rcx", "%rdx", "memory"
13   );
14
15   return tsc;
16 }
```

b: RDTSCP.

Listing 4.8: The RDTSC instruction with fencing and the RDTSCP instruction, written in inline assembler. Both functions must be placed inline and thus be preceded by `__attribute__((unused,always_inline))`.

The main difference between RDTSC and RDTSCP is that, unlike the former, the latter waits until all previous instructions have been executed and all previous loads are globally visible. One consequence of this, among others, was described by Paoloni [Pao10]. He demonstrated that RDTSC showed a standard deviation of 6.9 cycles, whereas RDTSCP only showed a standard deviation of 2 cycles.

Since not all x86 processors support RDTSCP, VILLASnode nonetheless includes RDTSC. However, to improve its behavior, the *Load Fence* (LFENCE) instruction [18e] is executed prior to the actual read instruction. This type of fence serializes all load-from-memory instructions prior to its call. Furthermore, no instructions that are placed after the load fence execute until the fence has completed.

Listing 4.9 shows the implementation of `task_wait()` based on the TSC. During the $(i+1)^{th}$ call of `task_wait()`, the counter is busy polled until the desired

timestamp t_{sample}^{i+1} is reached. Then, it updates the next timestamp t_{sample}^{i+2} and simultaneously calculates whether t_{sample}^{i+1} is actually only one step after t_{sample}^{i} or if some steps were missed. The period can be calculated according to:

$$T = \frac{f_{nominal}^{tsc}}{\texttt{rate}}. \tag{4.9}$$

```
1  uint64_t task_wait(struct task *t)
2  {
3    int ret;
4    uint64_t steps, now;
5
6    do {
7      now = rdtscp();
8    } while (now < t->next);
9
10   for (steps = 0; t->next < now; steps++)
11     t->next += t->period;
12
13   return steps;
14 }
```

Listing 4.9: Implementation of `task_wait()` by busy polling the x86 Time-Stamp Counter (TSC).

The advantage of this implementation of `task_wait()` is that given periods can be approximated very accurately ($\sigma = 2$ clock cycles, [Pao10]). Now, complications will rather arise because `signal_generator_read()` is not called frequently enough because datapaths are too long.

4.3.2 Further optimizations of the benchmark's datapath

Before the *signal* node from Figure 4.4 generates a sample, it checks whether steps were missed. Then, after it has generated a sample, the super-node has to write it to the file node and an instance of the node-type that is being tested. Only then, the *signal* node can generate the next sample. Both the time that is spent on this check and the time that is spent in the file node are part of the datapath and affect the time it takes before `task_wait()` is invoked again. Increasing $t_{\texttt{task\_wait()}}^{i+1}$ accordingly increases the chance of a timing violation according to equation (4.6). It is thus desirable that the time that is spent on the check and in the file node is minimized.

Suppressing information to the standard output Originally, a file node always kept track of the total number of missed steps, and wrote a message to the standard output as soon as one or more steps were missed. Especially the latter is relatively

expensive since `printf()` [Ker10] invokes a system call. For high rates, it can cause a snowball effect: this situation only occurs when the generation rate is already too high so that timing requirements are not met, and now, additionally, the time that is spent in the datapath is increased even more by adding system calls to write to the standard output. Since the missed steps can also be derived from the *in* and *out* log file, it is made configurable to disable internal logging of missed steps. Now, when minimal latency is required, like in the case of the VILLASnode node-type benchmark, a flag can be set in the configuration file.

Buffering the file stream Usually, each call to the *stdio* library—which is used by the file node-type to read from and write to files—results in a system call. Although it is not possible to get rid of these system calls completely—after all, they are necessary to write to the *in* and *out* log files—they should be reduced to an absolute minimum in the datapath. To achieve this, the file node-type was modified so that the buffering of the file stream can be configured. Now, a user can define the size of a buffer in the configuration file. Buffering is controlled with `setvbuf()` [Ker10], which enables an instance of the file node-type to read or write data in units equal to the size of that buffer.

4.4 Enabling UC support in the RDMA CM

The RDMA CM does not officially support unreliable connections. However, by modifying small parts of the `librdmacm` library and by re-compiling it, it is possible to facilitate UC anyway. This enables the present work to also analyze the unreliable connection with the custom and the VILLASnode node-type benchmark.

To enable support, the `rdma_create_id2()` function of the `librdmacm` has to be made non-static. As a result, this function can directly be accessed, whereas it is normally only accessible through the wrapper `rdma_create_id()`. Now, also the QP type can be passed on the the RDMA CM library, and by passing `RDMA_PS_IPOIB` as `port_space` and `IBV_QPT_UC` as `qp_type`, a managed UC QP will be created.

4.5 Processing data

In order to analyze the generated comma-separated value dumps, several Python 3.7 scripts were developed in Jupyter Notebook.[3,4] Jupyter Notebook (formerly IPython Notebook) is part of Project Jupyter and allows a user to interactively explore Python scripts. On the one hand, it enables (stepwise) execution of Python code in a web browser, based on IPython [PG07]. On the other hand, rich text documentation, written in Markdown,[5] can directly be included in the document. The

[3] https://python.org
[4] https://jupyter.org
[5] https://daringfireball.net/projects/markdown/

documentation, together with the source code, can be exported to several formats, e.g., to `.py`, `.tex`, `.html`, `.md`, and `.pdf`.

Jupyter Notebook's command line API makes it also highly suitable for automatic analysis of large datasets of timestamps. It is, for example, included in the CI/CD pipeline of VILLASnode to automatically analyze the performance impact of certain changes in the source code and to compare node-types against each other. Furthermore, the scripts are included in the present work's build automation, which makes it possible to easily convert raw data from the benchmarks to convenient graphs.

Besides several standard libraries, NumPy[6]—which adds support for numerical calculations in Python—and matplotlib[7]—which adds a comprehensive toolset to create 2D plots—were used.

4.5.1 Processing the host channel adapter benchmark's results

Histograms The first type of graph that is used in chapter 5 and appendix F is a histogram. The Python script that generates this graph first needs the path that contains the timestamps. This can be passed on through the command line or directly in the notebook. Then, the script loads the JSON file that must be present in every data directory. It contains settings on how to process the data, but also information about the plots, e.g., dimensions of the figure and labels.

When all preparatory work is done, the Python script loads all timestamps as defined in subsection 4.1.1. To keep the minimum message size as low as possible, this benchmark only sends the 8-byte `long tv_nsec` from Listing 4.1. However, the complication with only sending this long integer is that it overflows from 999 999 999 ns to 0 ns. But, since transmissions cannot take longer than 1 s—assuming no severe errors occur—this overflow is resolved by adding 1 s to t_{recv} and t_{comp} if they are smaller than t_{subm}/t_{send}.

Subsequently, all data is displayed in a histogram. To be able to see differences in the distribution of latencies at a glance and thus to make the comparison of the results easier, all histograms range from 0 ns to 10 000 ns. A small box in the top left or top right corner then provides information on the percentage of values above this limit and about the maximum value. A red, vertical line indicates the median value of the data set.

This script is able to compare data sets from the same run—for example, t_{lat} and t_{lat}^{comp}—or data sets from different runs—for example, t_{lat} from various runs with distinct settings. In the present work, the former and the latter first occur in Figure 5.2 and Figure 5.5, respectively.

Median plot with variability indication Histograms are great for getting a more comprehensive view of the distribution of latencies and the effect specific changes have on this distribution. However, this type of plot is not suitable for displaying

[6] http://numpy.org
[7] https://matplotlib.org

many different setups in one comprehensible graph. Therefore, a simple line chart is used to display the median values of several data sets. In order to add information about dispersion of latency, error bars are added to every marker. In the present work's line charts, these indicate an 80 % interval around the median value. Thus, for every marker, 10 % of the values are bigger than the upper limit of the error bar and 10 % of the values are smaller than the lower limit.

In the present work, this type of graph first occurs in Figure 5.8.

4.5.2 Processing the VILLASnode node-type benchmark's results

As discussed in section 4.3 and depicted in Figure 4.4, the VILLASnode node-type benchmark results in two files with data: an *in* and *out* file. For every sample, the former includes a generation timestamp, a sequence number, and the actual values of the sample. Additionally, the latter includes a receive timestamp which is computed by the receiving instance of the node-type that is being benchmarked.

The VILLASnode node-type benchmark serves two purposes. On the one hand, there must be a graph that shows the performance of all node-types in one glance and makes comparison of node-types easy. For this purpose, the line graph from the previous subsection is well suited.

On the other hand, the benchmark should give a comprehensive insight in the latency distribution and the maxima of a certain node-type. For this purpose, the histogram from subsection 4.5.1 is better suited. However, as described in section 4.3, these graphs should also provide information about the limitations of node-types. Not all node-types will be limited to the same maximum frequency. Therefore, the graph should provide additional information about the missing samples in the *in* and *out* file. By comparing these files, it can be determined if samples were not transmitted by the node-type that was tested.

3D surface plot To be able to wrap up all information up in one plot, a third type of graph is introduced: the 3D surface plot. With this type of graph, it is possible to vary both the message size and sample generation rate, whilst still displaying all data in a comprehensible manner. In addition to the median latencies of size/generation rate combinations, an indication of the percentage of missed steps is plotted. In that way, it is easy to identify which combinations were detrimental for the sample generation.

In the present work, this type of graph first occurs in Figure 5.11.

5 Evaluation

This chapter discusses the results of the previously presented benchmarks. Section 5.1 starts with an evaluation of the custom one-way HCA benchmark from section 4.1. After these results have been analyzed, section 5.2 will compare them to the results of `ib_send_lat` of the OFED™ Performance Test package. Subsequently, section 5.3 discusses the several VILLASnode node-types that were benchmarked.

Table 5.1 lists the hardware, the operating system, the OFED™ stack version, and the VILLASnode version that were used for all benchmarks. Fedora was selected as OS because of its support for the `tuned` daemon (subsection 2.4.5) and because of its easy-to-set-up support for `PREEMPT_RT`-patched kernels (section 7.1). At the time of writing the present work, the chosen Fedora and kernel version was the latest combination that was seamlessly supported by this version of the Mellanox® variant of the OFED™ stack.

Table 5.1: Dell PowerEdge T630 test system for benchmarks.

CPU	2× Intel® Xeon® E5-2643 v4, 20 MiB cache, 3.40 GHz base frequency
Chipset	Intel® C610
RAM	32 GB, DDR-4 2400 MHz, ECC buffered
Motherboard	Dell PowerEdge T630 System Board NT78X
Storage	Intel ® SSD DC P3700 Series, PCI-e (Gen 2) ×8, 400 GB
HCA	2× Mellanox® ConnectX®-4 MT27700, PCI-e (Gen 3) ×16, 100 Gbit/s
Physical link	0.5 m Mellanox® MCP100-E00A Passive copper Cable, 100 Gbit/s
OS	Fedora 27 @ Linux kernel 4.13.9-200
OFED™	MLNX OFED Linux 4.4-2.0.7.0
VILLASnode	Compiled version on commit 0819207c55ef06c7b98ddfe98637eb2b5e1e5d0b

The system was optimized using to the techniques from section 2.4. Unless stated otherwise, all analyses that are presented in this chapter have been run under these circumstances. Figure 5.1 shows the distribution of CPUs among cpusets (subsection 2.4.3). The CPUs in the two *real-time-<X>* cpusets are limited to the memory locations in their NUMA node (subsection 2.4.2). These memory locations are also the same as those the respective HCAs will read from or write to. Finally, the system is optimized by setting the `tuned` daemon to the *latency-performance* profile (subsection 2.4.5).

Thus, all time-critical processes that needed to use the HCAs `mlx5_0` and `mlx5_1` were run on the CPUs 16, 18, 20, and 22 and 17, 19, 21, and 23, respectively.

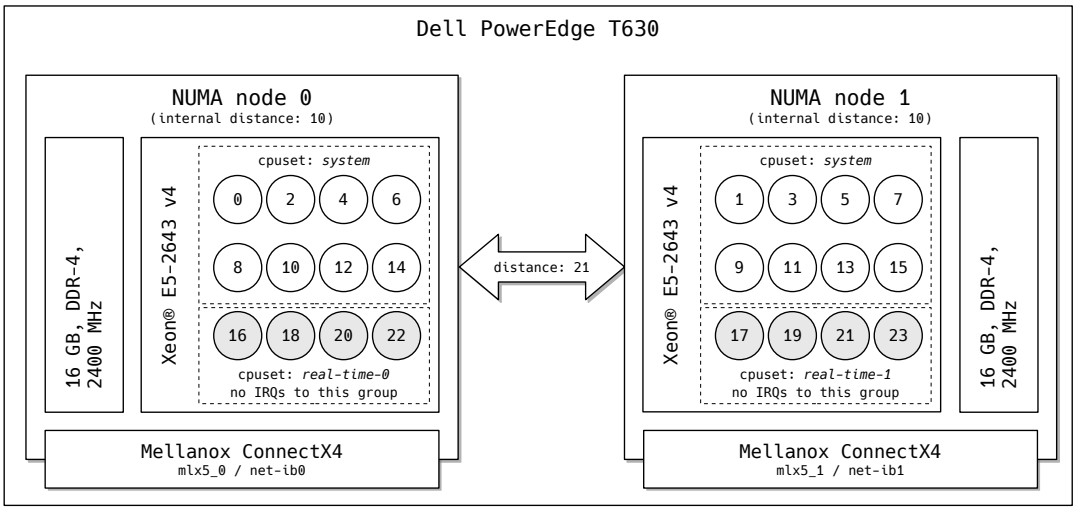


Figure 5.1: The configuration of the Dell PowerEdge T630 from Table 5.1, which was used in the present work's evaluations. NUMA specific data is acquired with `numactl`.

5.1 Custom one-way host channel adapter benchmark

This section examines different possible configurations of communication over an InfiniBand network using the benchmark presented in section 4.1. It is intended to help make a well considered choice regarding the configuration of the InfiniBand VILLASnode node-type and to get a ballpark estimate of the latency this communication technology will show in VILLASnode.

5.1.1 Event based polling

The first analyses that were performed were meant to examine the characteristics of event based polling (Figure 2.21b). Since event channels are designed to be CPU efficient, in this case, the optimizations from subsection 2.4.3 ("CPU isolation & affinity") and subsection 2.4.4 ("Interrupt affinity") were not applied and Figure 5.1 is not relevant. Instead of improving latency, the aforementioned optimizations had an adverse effect and actually increased latency. However, the `tuned` profile *latency-performance* and memory optimization techniques were applied nevertheless.

Table 5.2 shows the settings that were used with the custom one-way benchmark. These settings were introduced in subsection 4.1.2. Gray columns in Table 5.2, and in all following tables that list benchmark settings, indicate that the settings of

these columns were varied during the different runs. Consequently, all settings in the white columns stayed constant whilst performing the different tests. The graphs that were generated from the resulting data are shown in Figure 5.2.

Table 5.2: The benchmark's settings which were used to analyze the latency of messages sent whilst both the sending and receiving node were waiting for an event.

	service type	polling (send)	polling (recv)	inline mode	unsignaled	operation	burst size	repetitions	loop pauses	timestamp	message size
Figure 5.2a	RC	event	event	✗	✗	send	8000	20	0 ns	t_{subm}	32 B
Figure 5.2b	UC	event	event	✗	✗	send	8000	20	0 ns	t_{subm}	32 B
Figure 5.2c	UD	event	event	✗	✗	send	8000	20	0 ns	t_{subm}	32 B
Figure 5.2d	RC	event	event	✗	✗	send	8000	1	1×10^9 ns	t_{subm}	32 B
Figure 5.2e	UC	event	event	✗	✗	send	8000	1	1×10^9 ns	t_{subm}	32 B
Figure 5.2f	UD	event	event	✗	✗	send	8000	1	1×10^9 ns	t_{subm}	32 B

In the first three subfigures of Figure 5.2, $25 \cdot 8000$ messages of 32 B were bursted for RC, UC, and UD. This message size was chosen in most of the following tests because it is the minimum size of a message in the VILLASnode *InfiniBand* node-type. Every sample that is sent from one VILLASnode *InfiniBand* node to another contains at least one 8-byte value and always carries 24 B of metadata.

The first thing that catches the eye are the relatively high median latencies (equation (4.1)) of all service types: $\tilde{t}_{lat}^{RC} = 3608\,\text{ns}$, $\tilde{t}_{lat}^{UC} = 3598\,\text{ns}$, and $\tilde{t}_{lat}^{UD} = 3389\,\text{ns}$. These were caused by the event channels that were used for synchronization: with abovementioned settings, the benchmark waits until a `read()` system call returns before it tries to poll the completion queue. Therefore, in the meantime, other processes can be scheduled onto the CPU and it will take a certain amount of time to wake the benchmark up again. So, event based polling results in a lower CPU utilization compared to busy polling, but, in return, yields a higher latency.

Maxima The maximum latencies that can be seen were mainly caused by initial transfers immediately after the process started or after a period of hibernation. This is sometimes referred to as the *warm up effect*. Potential solutions for this problem are introduced in section 7.1.

The custom one-way benchmark includes another potential cause for latency maxima. As mentioned in subsection 4.1.1, the function that measures and saves the receive timestamps (Listing 4.3) lies in the time-critical path. The worst case situation, in which the two memory regions were only initialized by `mmap()` but were not yet touched and thus allocated, was examined. This caused maxima of more than 700 µs. When the pages were present in the virtual memory, the latency of both save operations was determined to be approximately 40 ns together.

Thus, in order to make full use of the capabilities and low latencies of InfiniBand, it is important to carefully pick the operations that lie in the datapath.

Minima The small peaks at the left side of the graphs, between approximately 900 ns and 2900 ns, were caused by how this benchmark implements event based polling. Figure 2.21b already showed that after a completion channel notifies the process that a new CQE is available, the CQ must be polled with `ibv_poll_cq()` to acquire CQEs. After polling, this benchmark does not immediately return control to `ibv_get_cq_event()`; rather it tries to poll again to see if new messages arrived in the meantime. If this was the case, these messages did not have to wait until a `read()` system call returned before they got processed, for that reason, their latency was lower.

Sent confirmations Subsection 2.2.2 already discussed at which moment CQEs at the send side are generated. In case of a reliable connection (Figure 5.2a), entries showed up in the completion queue when a message was delivered to a remote CA and when that CA acknowledged that it received the message. Naturally,

$$t_{lat}^{comp} = t_{comp} - t_{subm} > t_{recv} - t_{subm} = t_{lat} \tag{5.1}$$

was almost certainly true for every message that was sent.

This was different for the unreliable service types (UC and UD, Figure 5.2b and Figure 5.2c), where the HCA is only responsible for sending a message. Hence, in these cases, the HCA generated a CQE immediately after a message was sent. Thus, for more messages,

$$t_{lat}^{comp} < t_{lat} \tag{5.2}$$

was true. In Figure 5.2b, this cannot be identified yet, but the difference between the median values $\tilde{t}_{lat}^{comp}$ and $\tilde{t}_{lat}$ is getting smaller. For messages that were sent as unreliable datagrams, equation (5.2) usually holds, and in Figure 5.2c,

$$\tilde{t}_{lat}^{comp} < \tilde{t}_{lat} \tag{5.3}$$

is even true.

Comparison of the service types It can be seen that the median latencies of the unreliable service types were barely different from the median latency of the reliable connection. With 3598 ns and 3521 ns, the median latencies of UC and UD were just slightly lower than the median latency of 3608 ns of the RC service type. As expected, this was caused by the absence of acknowledgment messages between the two channel adapters. However, the variability of the three service types differed. With regards to t_{lat}, UD had the highest ($t_{lat} > 10\,000$ ns in 0.1665 % of the cases) and UC the lowest ($t_{lat} > 10\,000$ ns in 0.0595 % of the cases) dispersion. In the remainder of this section, 10 000 ns and 10 µs will be used interchangeably with regards to the significant figures.

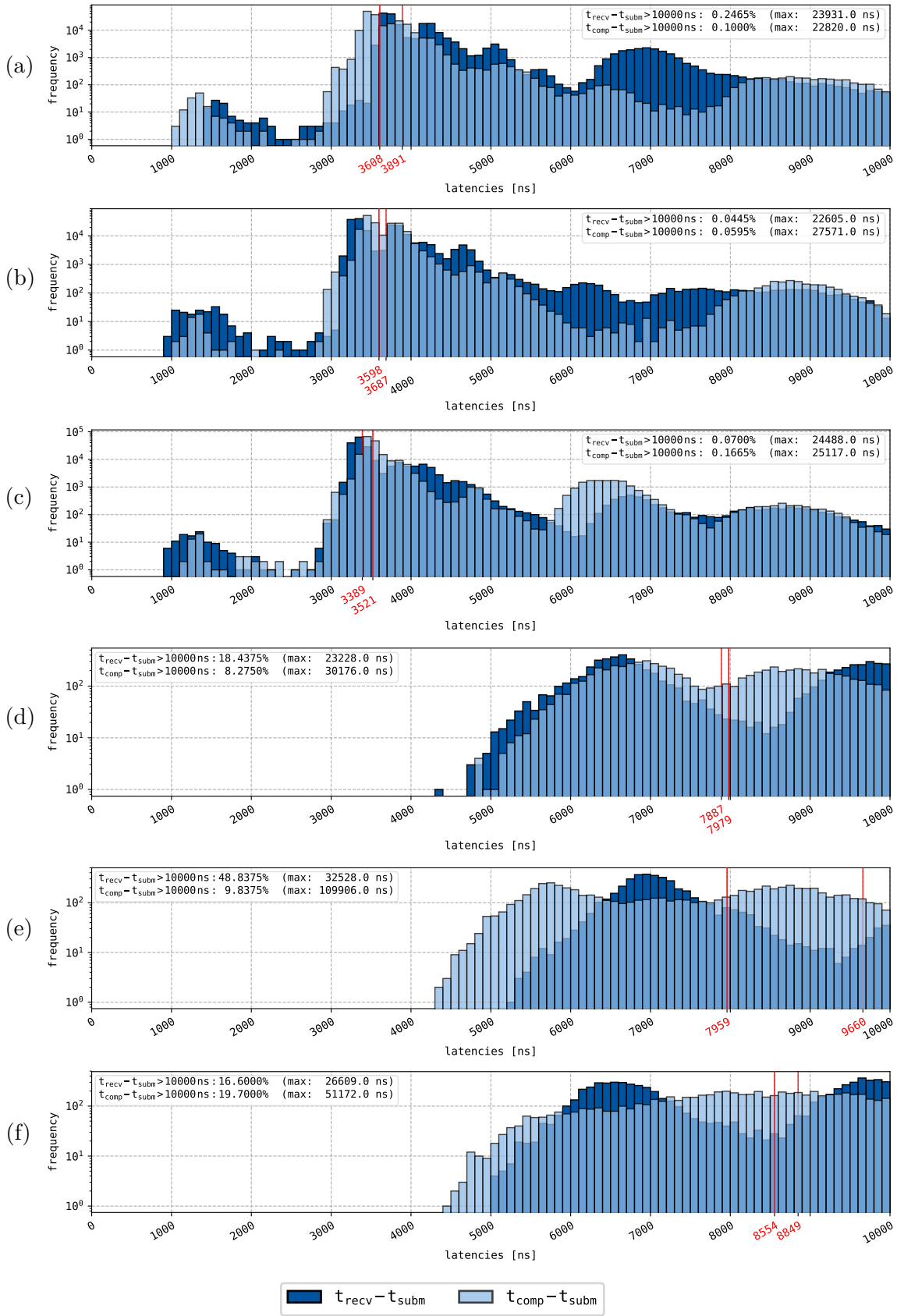


Figure 5.2: Results of the one-way benchmark with the settings from Table 5.2. These were used to analyze latencies with event based polling.

Intermediate pauses The last three subfigures of Figure 5.2 show the results of the same test, but with an intermediate pause of 1 000 000 000 ns (1 s) and with just $1 \cdot 8000$ messages per run. One can see that the latency almost doubled. The pause of 1 s was long enough for the OS to swap out the waiting process, and it took a considerable amount of time to re-activate the process after the `read()` system call returned. Furthermore, the peaks at the left side of the graphs completely disappeared because now there could never be a second entry in the CQ after the first entry was acquired.

5.1.2 Busy polling

Event based polling is suitable for semi-time-critical applications in which minimal CPU utilization outweighs maximum performance and thus minimal latency. However, if minimal latency is the topmost priority, busy polling (Figure 2.21a) should be used.

To be able to compare apples to apples, the settings in Table 5.3 are very much alike those in Table 5.2, but with a different polling mode. Since busy polling is a CPU intensive task, all tests were performed in the optimized environment that was presented at the beginning of this chapter. The results of the tests are displayed in Figure 5.3.

Table 5.3: The benchmark's settings which were used to analyze the latency of messages sent whilst both the sending and receiving node were busy polling.

	service type	polling (send)	polling (recv)	inline mode	unsignaled	operation	burst size	repetitions	loop pauses	timestamp	message size
Figure 5.3a	RC	busy	busy	✗	✗	send	8000	20	0 ns	t_{subm}	32 B
Figure 5.3b	UC	busy	busy	✗	✗	send	8000	20	0 ns	t_{subm}	32 B
Figure 5.3c	UD	busy	busy	✗	✗	send	8000	20	0 ns	t_{subm}	32 B
Figure 5.3d	RC	busy	busy	✗	✗	send	8000	1	1×10^9 ns	t_{subm}	32 B
Figure 5.3e	UC	busy	busy	✗	✗	send	8000	1	1×10^9 ns	t_{subm}	32 B
Figure 5.3f	UD	busy	busy	✗	✗	send	8000	1	1×10^9 ns	t_{subm}	32 B

In the first three subfigures of Figure 5.3, again, $25 \cdot 8000$ messages of 32 B were bursted for RC, UC, and UD. It is immediately visible that the median latencies $\tilde{t}_{lat}^{RC} = 1269$ ns, $\tilde{t}_{lat}^{UC} = 1251$ ns, and $\tilde{t}_{lat}^{UD} = 1273$ ns are approximately 65 % lower than the same latencies for event based polling. This is in line with the findings of MacArthur and Russel [MR12], who reported a decrease of almost 70 % in their work.

Since the completion queues on the send side were also busy polled, their latencies also went down. Now, equation (5.2) holds for both unreliable service types. Note

that it could be, depending on the use case, beneficial to busy poll the receive CQ but to rely on a completion channel that is bound to the send queue. In that way, less CPU cores are fully utilized by busy polling, but low latencies are achieved between the sending and receiving node anyway. This approach would naturally result in:

$$t_{lat}^{comp} \gg t_{lat}, \tag{5.4}$$

and is suitable for applications that do not need to release the send buffers virtually instantaneous (subsection 3.3.2 & subsection 3.3.3).

Maxima The maximum latencies did not decrease with the same proportions as the median latencies, but still notably. With regards to $\max t_{lat}$, the results for the reliable service type decreased with approximately 14 % and for the unreliable service types with approximately 36 %. The main reason for the maxima was likely the same as for event based polling: the warm up effect caused peaks at the beginning of the transmission. This conjecture is strengthened by the tests that were done with an intermediate pause of 1 s. For these runs, the yielded maximum latencies were only slightly lower, which indicates that the maxima were not caused by congestion but rather by the scheduling of the polling process. After all, the tests that were performed with an intermediate pause of 1 s between transmissions are unlikely to have been subject to congestion.

Minima Latency minima as could be seen with event based polling could not arise here. This polling mode polls continuously all the time, so no peaks can arise because of short periods of time during which another polling mode was used.

Variability The number of messages for which it took more than 10 µs to arrive at the receiving host was almost one order of magnitude lower for the RC and UD service types, and approximately 5 times lower for the UC service type. This considerably reduced variability naturally implies a higher predictability. When sending messages in an environment that is based on busy polling, the maximum latency can be estimated with more certainty.

Intermediate pauses This shows another important difference between event based polling and busy polling. Whereas the runs with event based polling showed more than double the latency when intermediate pauses occurred between transfers, runs that relied on busy polling showed a much smaller difference. Latencies of tests with intermediate pauses were about 20 % higher than latencies of tests without any pauses when busy polling was applied. The same comparison for tests that relied on event based polling yielded a difference of 120 %.

Although the median latencies with intermediate pauses when busy polling were substantially better than when waiting for an event, they were still higher than anticipated. Since the process continuously polled the completion queue, and the operating system should thus not have suspended it, it was expected that $\tilde{t}_{lat}$ would

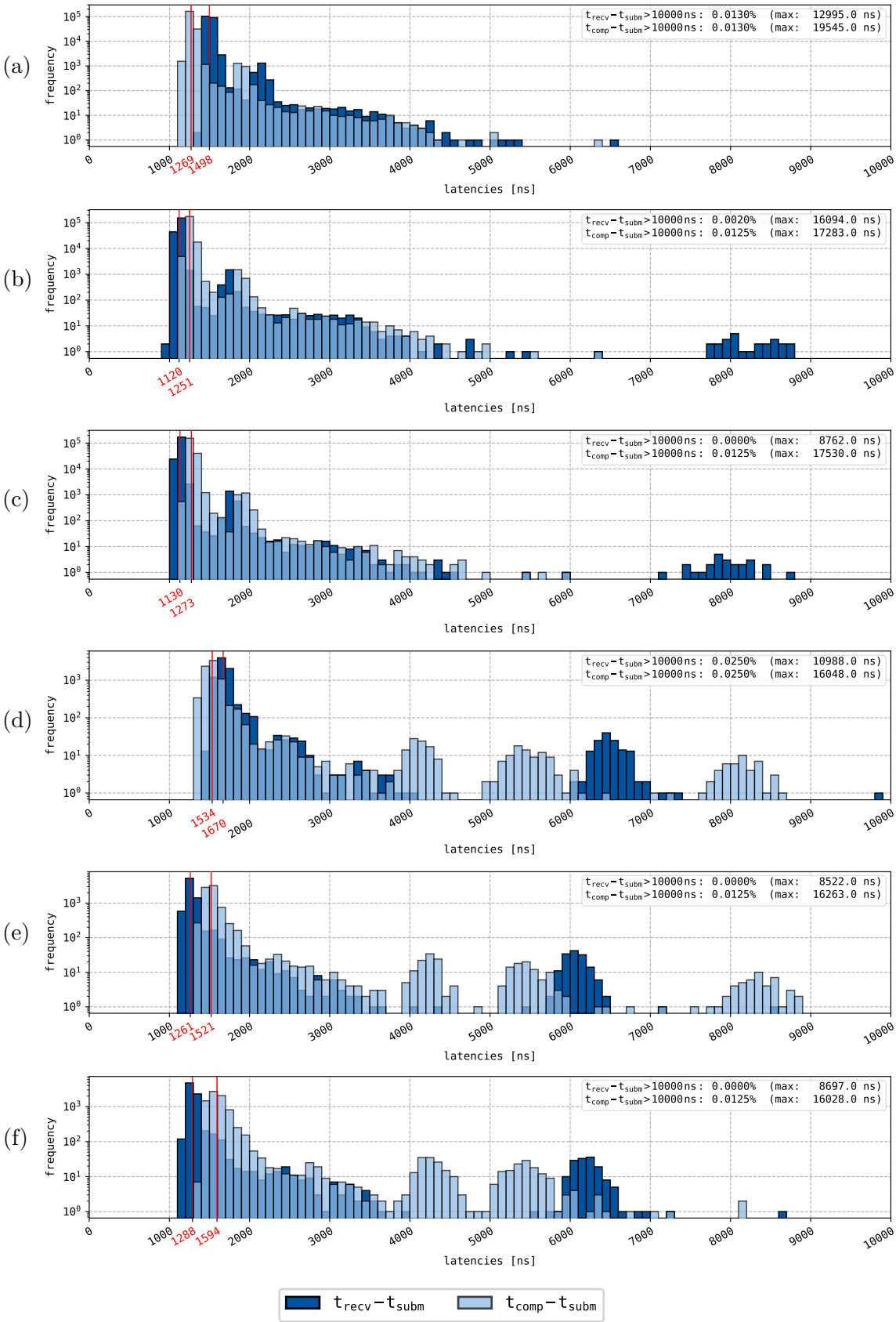


Figure 5.3: Results of the one-way benchmark with the settings from Table 5.3. These were used to analyze latencies with busy polling.

be lower for scenarios with less traffic on the link. However, for these cases, $\tilde{t}_{lat}$ was slightly higher in Figure 5.3.

It was first suspected that *Active State Power Management* (ASPM), which is described in the PCI-e Base Specifications [10], caused this additional latency. This technique sets the PCI-e link to a lower power state when the device it is connected to—which would in this case be the HCA—is not used. However, when the tests from Table 5.3 were repeated with ASPM explicitly turned off, the results remained the same.

The second suspicion was related to power saving levels of the CPU: the so-called *C-states*. After ensuring that all power savings were turned off—i.e., C0 was the only allowed state—a maximum response latency of 0 µs was written to `/dev/cpu_dma_latency`. This virtual file forms an interface to the *Power Management Quality of Service* (PM QoS),[1] and writing `'0'` to it expresses to the OS that the minimum achievable DMA latency is required. However, this did also not improve $\tilde{t}_{lat}$.

Nevertheless, busy polling is still the more suitable technique for real-time applications. The next sections will explore other techniques to reduce the latency even more. For the methods that are likely to have a similar impact on the different service types, only the UC service type was used for the sake of brevity. The unreliable connection was chosen because it showed the best results so far.

5.1.3 Differences between the submit and send timestamp

This subsection explores the difference between the moment a work request is submitted to the send queue and the moment the HCA actually sends the data. The feature of the benchmark that measures this difference is based on Listing 4.4: the sending node keeps updating the timestamp until the HCA copies the data to one of its virtual lanes.

Table 5.4: The benchmark's settings which were used to analyze the difference in time between the moment that a Work Request (WR) is submitted to the Send Queue (SQ) and the moment the corresponding message is actually sent.

	service type	polling (send)	polling (recv)	inline mode	unsignaled	operation	burst size	repetitions	loop pauses	timestamp	message size
Figure 5.4	UC	busy	busy	✗	✗	send	8000	20	0 ns	t_{subm}	32 B
Figure 5.4	UC	busy	busy	✗	✗	send	8000	20	0 ns	t_{send}	32 B

[1] https://www.kernel.org/doc/Documentation/power/pm_qos_interface.txt

Table 5.4 shows the settings of the two tests that were performed. The results of both are plotted in Figure 5.4.

In the results of this test, and in the results of all following tests of this type, all data regarding t_{lat}^{comp} is completely omitted. In the previous two subsections, it could be seen that settings that affect the receive CQ will affect the send CQ in a very similar manner. Hence, continuing to plot it would have been redundant. Rather, two similar data sets that must be compared—e.g., $(t_{recv} - t_{send})$ and $(t_{recv} - t_{subm})$—have been plotted in the same graph.

As it turns out, approximately

$$\left(1 - \frac{726\,\text{ns}}{1253\,\text{ns}}\right) \cdot 100\,\% \approx 42\,\% \tag{5.5}$$

of the time that was needed to send a message from one node to another node was spent before the HCA actually copied the data. This timespan includes the notification of the HCA, but also the accessing and copying of the data from the hosts's main memory to the HCA's internal buffers. Note that this test did not measure the time the data spent in the sending node's HCA since it is not possible to update the timestamp as soon as it resided in the HCA's buffers.

This relatively long timespan suggests that the memory access is a bottleneck. The next subsection will discuss a possible solution for small messages.

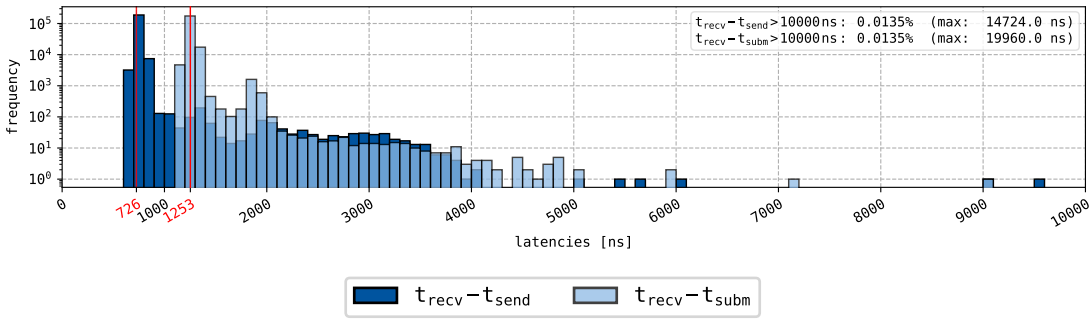


Figure 5.4: Results of the one-way benchmark with the settings from Table 5.4. These were used to analyze the difference between t_{lat} and t_{lat}^{send}.

5.1.4 Inline messages

Equation (4.3) in subsection 4.1.1 already suggested that the difference between $\tilde{t}_{lat}$ and $\tilde{t}_{lat}^{send}$ could be an approximation of the latency decrease that can be achieved by using the *inline* flag that some InfiniBand HCAs—among them the Mellanox® ConnectX®-4—support. By setting this flag, introduced in subsection 2.3.1, relatively small messages ($\lesssim 1\,\text{KiB}$) will directly be included in a work request. Accordingly, the HCA's DMA does not need to access the host's main memory to acquire the data when it becomes aware of the submitted WR. This suggests that posting small

messages inline will eradicate a part of the overhead that was discussed in the last subsection.

Table 5.5 shows which settings were used with the one-way benchmark to analyze this difference. They are almost identical to the settings from Table 5.4, but instead of varying the timestamp that was taken (t_{subm}/t_{send}), the inline mode was varied. The results are depicted in Figure 5.5.

Table 5.5: The benchmark's settings which were used to analyze the influence of sending messages inline on the latency.

	service type	polling (send)	polling (recv)	inline mode	unsignaled	operation	burst size	repetitions	loop pauses	timestamp	message size
Figure 5.5	UC	busy	busy	✗	✗	send	8000	20	0 ns	t_{subm}	32 B
Figure 5.5	UC	busy	busy	✓	✗	send	8000	20	0 ns	t_{subm}	32 B

Being 1264 ns, the median latency for the regularly submitted case was almost identical to the latency in Figure 5.4, which makes it very suitable for comparison. In subsection 5.1.3, it was determined that about 42 % of the time was lost before the HCA actually copied the data to its own buffers. The graph shows that messages that were submitted with the inline flag had a

$$\left(1 - \frac{906\,\text{ns}}{1264\,\text{ns}}\right) \cdot 100\,\% \approx 28\,\% \tag{5.6}$$

lower latency than regularly submitted messages.

Thus, apparently, the additional memory access the HCA had to perform when a 32 B message was not directly included in the work request was accountable for 28 % of the latency. Hence, if possible, it is favorable for latency to include data directly in the work request. Furthermore, as mentioned in subsection 2.3.1, another advantage is the fact that the used buffers can be released immediately after submitting the WR.

5.1.5 RDMA write compared to the send operation

Table 2.5 presented the different operations which are supported for the different service types. So far, all discussed tests relied on *send with immediate*. The second suitable operation to transfer a message to a remote host which also supports an additional 32-bit header as identifier is *RDMA write with immediate*. In the remainder of this chapter, for the sake of brevity, this operation is simply referred to as *RDMA write*.

Table 5.6 describes the settings that were used with the one-way benchmark to compare the *send* operation with *RDMA write*. Note that UD is not included, since

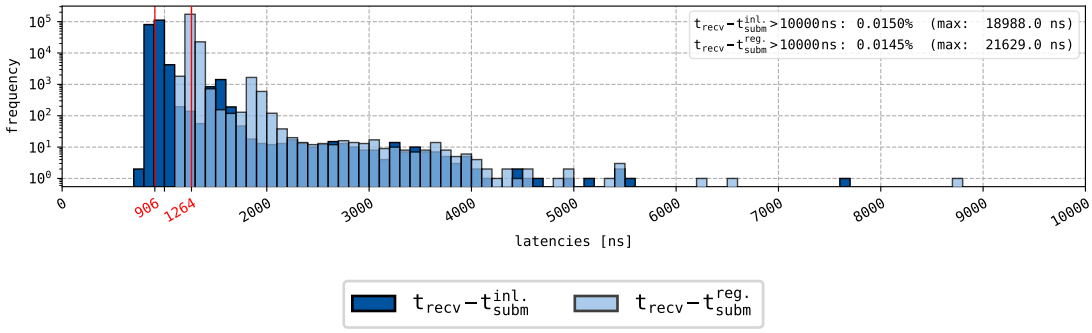


Figure 5.5: Results of the one-way benchmark with the settings Table 5.5. These were used to analyze the difference between messages that are submitted regularly ($t^{reg.}_{subm}$) and that are submitted inline ($t^{inl.}_{subm}$).

Table 5.6: The benchmark's settings which were used to analayze the effect on latency of sending messages through memory semantics instead of channel semantics.

	service type	polling (send)	polling (recv)	inline mode	unsignaled	operation	burst size	repetitions	loop pauses	timestamp	message size
Figure 5.6a	RC	busy	busy	✓	✗	send	8000	20	0 ns	t_{subm}	32 B
Figure 5.6a	RC	busy	busy	✓	✗	rdma	8000	20	0 ns	t_{subm}	32 B
Figure 5.6b	UC	busy	busy	✓	✗	send	8000	20	0 ns	t_{subm}	32 B
Figure 5.6b	UC	busy	busy	✓	✗	rdma	8000	20	0 ns	t_{subm}	32 B

none of the RDMA operations support it. The results of the tests are depicted in Figure 5.6.

In these results, the *RDMA write* operation seems slower than the *send* operation. However, a few remarks have to be made. First, the maximum latency and the variability of the RDMA transfers were lower. In case of the UC service type, sending messages with RDMA resulted in 5× less messages with a latency greater than 10 µs. (In some iterations of the tests, reductions up to 25× could be seen.) So, although the median latency was slightly higher for RDMA, the lower variability makes it a more predictable service type.

Secondly, this test relied on the *RDMA write with immediate*, not *RDMA write*. The actual *RDMA write* operation is probably a little faster, but without synchronization there is no way for a process on the receiving side to know when data is available. Since the only other way of synchronizing would be using an additional *send* operation, *RDMA write with immediate* is the fastest way of sending data with RDMA and signaling to the receiving node that data is available.

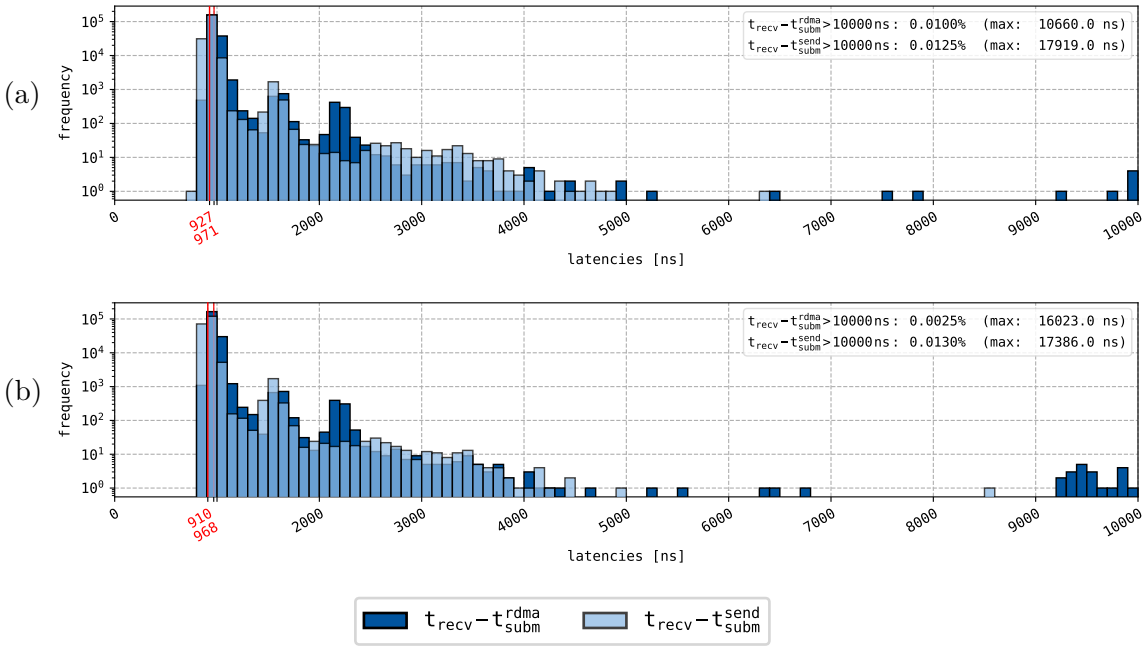


Figure 5.6: Results of the one-way benchmark with the settings from Table 5.6. These were used to analyze the difference between the *RDMA write with immediate* and *send with immediate* operation.

5.1.6 Unsignaled messages compared to signaled messages

Subsection 2.3.1 discussed that the OFED™ verbs allow to submit WRs to the SQ without generating a notification. Thereafter, subsection 4.2.4 presented how this technique was implemented in the node-type's write-function. This was done to prevent file structures from unnecessarily rippling through the completion queue into the write-function, to subsequently be discarded there. Since MacArthur and Russel [MR12] only observed small performance increases but recommended sending unsignaled for inline messages, the following tests were intended to review the performance increase in the present work's environment.

Table 5.7 shows the settings that were used with the one-way benchmark during these tests and Figure 5.7 shows the resulting latencies. The median latency $\tilde{t}_{lat}^{sig.}$ of the messages that were sent inline with signaling approximately corresponds to the number from Figure 5.5. Thus, since Figure 5.7 shows that the median latency of unsignaled messages is:

$$\tilde{t}_{lat}^{uns.} \approx 0.87 \cdot \tilde{t}_{lat}^{sig.}, \tag{5.7}$$

it can be concluded that turning signaling off yields a noteworthy performance increase. By signaling only shortly before the send queue overflows, a decrease in latency of almost 13 % can be seen.

Because previous works [MR12; LR14] were inclined to use *RDMA write* over *send* operations, the same tests as in Table 5.7 were repeated with *RDMA write* as operation mode. Similar to the results in Figure 5.6, the latency for messages that

Table 5.7: The benchmark's settings which were used to analyze the influence of Completion Queue Entry (CQE) creation on latency for *send* operations.

	service type	polling (send)	polling (recv)	inline mode	unsignaled	operation	burst size	repetitions	loop pauses	timestamp	message size
Figure 5.7	UC	busy	busy	✓	✗	send	8000	20	0 ns	t_{subm}	32 B
Figure 5.7	UC	busy	busy	✓	✓	send	8000	20	0 ns	t_{subm}	32 B

were sent over RDMA was worse than for those that were sent normally. However, the relative increase in performance caused by the disabling of the signaling was, being a bit more than 12 %, almost identical to the increase in Figure 5.7.

The settings and the results of these tests can be seen in appendix F.1.

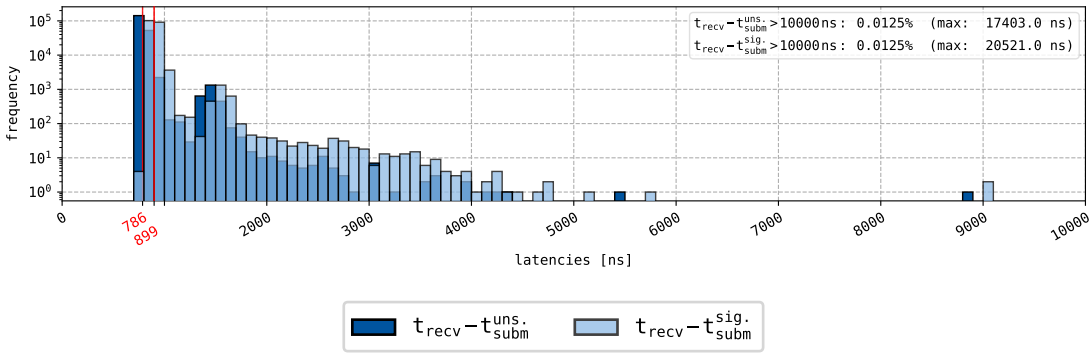


Figure 5.7: Results of the one-way benchmark with the settings from Table 5.7 to analyze the difference in latency between messages that did and did not cause a Completion Queue Entry (CQE). The *send* operation mode was used in this test.

Based on the results from the previous subsections, $\tilde{t}_{lat} = 786\,\text{ns}$ seems to be the lowest achievable median latency for 32-byte messages. This confirms the implementation of the VILLASnode node-type that was presented in subsection 4.2.3 and 4.2.4. In the communication between *InfiniBand* node-types, the *send* operation mode is used, when under a configurable threshold messages are sent inline, and a CQE for inline-messages is only generated when a counter reaches a configurable threshold.

Although sub-microsecond latencies could easily be achieved in the used environment, there was still a considerable deviation from the latencies MacArthur and Russel [MR12] observed, which had minima around 300 ns. A possible explanation for this could be the number of used buffer. The objective of this benchmark was to find the best fit for a VILLASnode node-type. Because a node-type needs a relatively large pool of buffers to be able to process a lot of small samples with high

frequencies, this benchmark also assumed a large pool of buffers. MacArthur and Russel, however, observed that latencies in their environment started to increase when more than 16 buffers were used.

5.1.7 Variation of message size

All aforementioned tests assumed an idealized situation with 32-byte messages. Usually, the packets in a real-time co-simulation framework will be a few powers of two larger. Table 5.8 shows the settings that were used with the one-way benchmark to explore the influence of message size on the latency.

The tests are grouped in three categories: Figure 5.8a exclusively shows the RC, Figure 5.8b the UC, and Figure 5.8c the UD service type. Furthermore, an upward pointing triangle and a dark shade indicate the *send* operation, and a downward pointing triangle and a light shade an *rdma write* operation. Black shades were used for messages that were sent normally and blue shades for messages that were sent inline.

Whenever possible, tests were performed with messages ranging from 8 B to 32 KiB. However, inline work requests and the UD service type do not support messages that big; the adjusted ranges are listed in Table 5.8.

Table 5.8: The benchmark's settings which were used to analyze the influence of message size on the latency, with $i \in [0, 12]$, $j \in [0, 7]$, and $k \in [0, 9]$.

	service type	polling (send)	polling (recv)	inline mode	unsignaled	operation	burst size	repetitions	loop pauses	timestamp	message size
Figure 5.8a ▲	RC	busy	busy	✗	✗	send	8000	10	0 ns	t_{subm}	$8 \cdot 2^i$ B
Figure 5.8a ▲	RC	busy	busy	✓	✗	send	M_{inl} [1]	10	0 ns	t_{subm}	$8 \cdot 2^j$ B
Figure 5.8a ▼	RC	busy	busy	✗	✗	rdma	8000	10	0 ns	t_{subm}	$8 \cdot 2^i$ B
Figure 5.8a ▼	RC	busy	busy	✓	✗	rdma	M_{inl} [1]	10	0 ns	t_{subm}	$8 \cdot 2^j$ B
Figure 5.8b ▲	UC	busy	busy	✗	✗	send	8000	10	0 ns	t_{subm}	$8 \cdot 2^i$ B
Figure 5.8b ▲	UC	busy	busy	✓	✗	send	M_{inl} [1]	10	0 ns	t_{subm}	$8 \cdot 2^j$ B
Figure 5.8b ▼	UC	busy	busy	✗	✗	rdma	8000	10	0 ns	t_{subm}	$8 \cdot 2^i$ B
Figure 5.8b ▼	UC	busy	busy	✓	✗	rdma	M_{inl} [1]	10	0 ns	t_{subm}	$8 \cdot 2^j$ B
Figure 5.8c ▲	UD	busy	busy	✗	✗	send	8000	10	0 ns	t_{subm}	$8 \cdot 2^k$ B
Figure 5.8c ▲	UD	busy	busy	✓	✗	send	M_{inl} [1]	10	0 ns	t_{subm}	$8 \cdot 2^j$ B

[1] The maximum size M_{inl} of a QP for a given message size is dependent on the HCA. In case of the Mellanox ConnectX®-4, each queue of a QP could hold 8000, 8000, 8000, 6552, 5461, 4096, and 2730 WRs for a message size of 8 B, 16 B, 32 B, 64 B, 128 B, 256 B, and 512 B, respectivly.

Constant latency (8 B–256 B) As can be seen in Figure 5.8, all $\tilde{t}_{lat}$ of messages that were smaller than 256 B were virtually the same. The only difference is that, as expected from equation (5.6), messages that were sent inline have a median latency that is approximately 28 % lower than messages that were sent normally. All these $\tilde{t}_{lat}$ were around the values that could be seen for 32 B messages in Figure 5.3, 5.5, and 5.6. This is similar to MacArthur and Russel's results [MR12]. In their publication, they found that messages smaller than 1024 B have a somewhat constant latency. In the present work's finding this is only true for messages up to approximately 256 B.

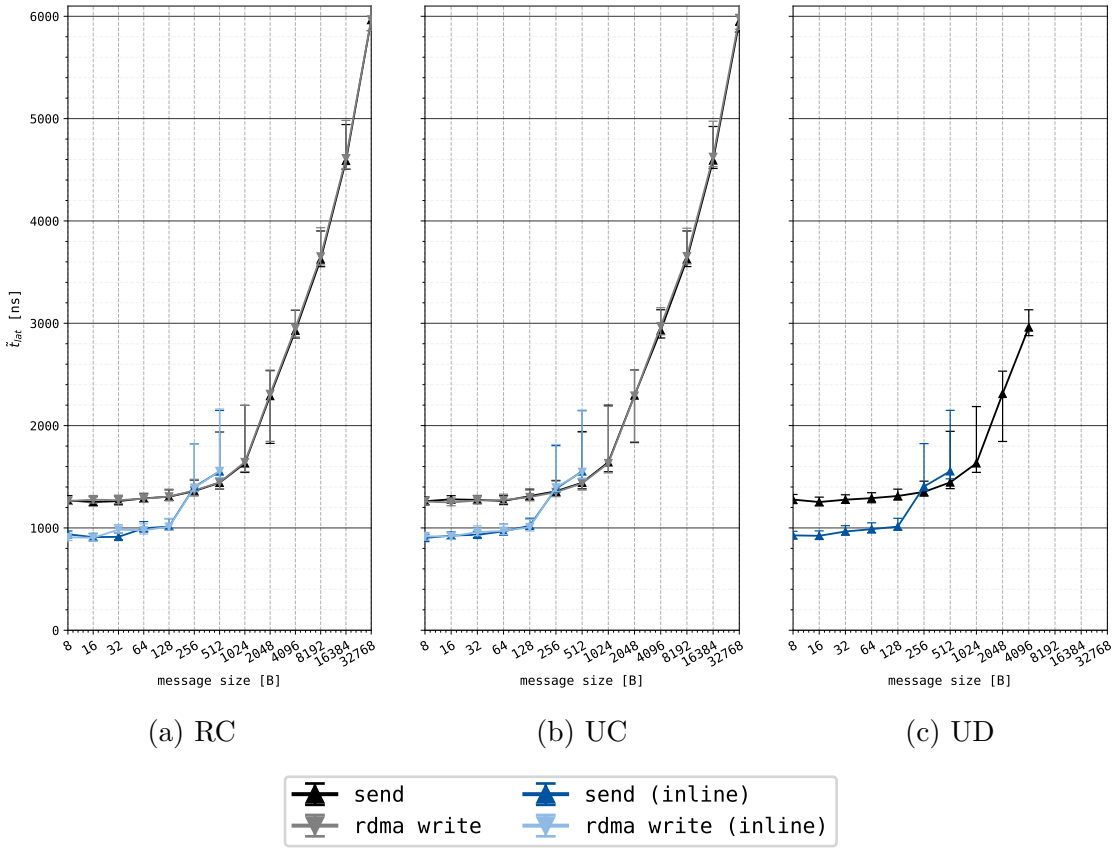
For all these sizes, the variance of the latencies is minimal. The error bars in Figure 5.8 indicate where the boundary to the upper and the lower 10 % of the values lie.

Increasing latency (256 B–32 KiB) When the message size exceeded 256 B, $\tilde{t}_{lat}$ started to gradually go up and the variance increased for messages that were sent normally. At 256 B, $\tilde{t}_{lat}$ for messages that were sent inline even exceeded the median latency of messages that were sent normally. Because not only the message size but also the burst size changed for the blue lines, the inline tests were repeated with a fixed burst size of 2730 messages per burst (appendix F.2). Since this steep slope between 128 B and 256 B is still present for fixed burst sizes, it can be concluded that—although the HCA allows it—sending data inline is not always favorable.

The increasing latency of inline messages around 256 B is in line with the findings of MacArthur and Russel [MR12]. In their work, they claim that this latency step was caused by their adapter's cache line size, which happened to be 256 B. The HCA that was used in the present work, however, had a cache line size of a mere 32 B. Thus, according to their findings, messages that were equal to or bigger than 32 B should have had latencies which were substantially bigger than latencies of messages that were smaller than 32 B. However, this was not the case, as can be seen in Figure 5.8. This leads to the conclusion that the increase is not solely caused by the cache line size.

Decreased variability (4096 B) The second aspect that catches the eye is located at 4096 B, which also happened to be the set MTU in these tests. For all service types—even for UD, which does only support messages up to the MTU—the variability of the latency decreased for messages bigger than or equal to 4096 B. Thus, although the median latency continued to go up, the predictability of the latency also rose.

Further peculiarities There was no meaningful difference between channel semantics and memory semantics with immediate data. Although the *send* operation was always slightly better in terms of median latency, the operation that best suits the requirements of the application should be used.



(a) RC (b) UC (c) UD

Figure 5.8: Results of the one-way benchmark with the settings from Table 5.8. These were used to analyze the influence of message size on the latency. While a triangle indicates $\tilde{t}_{lat}$ for a certain message size, the error bars indicate the upper and lower 10 % of t_{lat} for that message size.

To make sure that the increased median latency was not caused by congestion control (subsection 2.2.6) all tests from Table 5.8 were repeated with an intermediate pause of 5500 ns between calls of `ibv_post_send()`. As appendix F.3 shows, this did not influence the median latency.

5.2 OFED's round-trip host channel adapter benchmark

This section analyzes some assumptions that were made in previous sections. In the first subsection, the results of the round-trip benchmark `ib_lat_send` will be compared to the results from subsection 5.1.7. Then, in the second and third subsection, the influence of the MTU and the QP type on latency will be examined.

5.2.1 Correspondence between round-trip and one-way benchmark

Table 5.9 shows the results for the first tests that were performed with `ib_send_lat`. The median latencies in Table 5.9 approximately correspond to the latencies for the same test in Figure 5.8. It stands out that the same leap in latency between 128 B and 256 B that could be seen in Figure 5.8, also occurred in these results. To rule out that this leap was solely caused by the fact that messages were not sent inline anymore at 256 B, the test was also performed with the inline threshold set to a higher value. In this second test, the leap between 128 B and 256 B turned out to be even higher.

Table 5.9: $\min t_{lat}$, $\tilde{t}_{lat}$, and $\max t_{lat}$ measured with `ib_send_lat`. All communication went over an RC RDMA CM QP and was sent with the normal *send* operation. Every test contained 1000 iterations and messages that were smaller than 188 B were sent inline.

	round-trip benchmark			one-way benchmark
message size [B]	$\min t_{lat}$ [µs]	$\tilde{t}_{lat}$ [µs]	$\max t_{lat}$ [µs]	$\tilde{t}_{lat}$ [µs]
8	0.80	0.83	2.37	0.94
16	0.79	0.83	4.06	0.91
32	0.79	0.82	4.29	0.91
64	0.82	0.86	2.20	1.00
128	0.86	0.90	1.73	1.01
256	1.24	1.30	2.05	1.36
512	1.31	1.35	2.56	1.42
1024	1.45	1.49	2.78	1.63
2048	1.71	1.75	2.87	2.29
4096	2.22	2.27	2.87	2.93
8192	2.55	2.61	3.97	3.62
16 384	3.14	3.21	4.54	4.58
32 768	4.48	4.59	5.78	5.96

Difference in maximum latencies A substantial difference between the results of the round-trip benchmark and the custom one-way benchmark were the maxima. This was, in all likelihood, caused by the used sample size. The description on the OFED™ Performance Tests' Git[2] states that "setting a very high number of

[2] https://github.com/linux-rdma/perftest

iteration may have negative impact on the measured performance which are not related to the devices under test. If [...] strictly necessary, it is recommended to use the -N flag (No Peak)." Therefore, the default setting of the round-trip benchmark was set to 1000 messages per test. Since the custom one-way benchmark was meant to mimic the behavior of InfiniBand hardware in VILLASnode—which would also burst large amounts of small messages at high frequencies—this hint was ignored in the custom benchmark. Every marker in Figure 5.8 includes between 27 300 and 80 000 time deltas.

Foreseeing the analysis of the *InfiniBand* node-type, the one-way benchmark gave a more realistic view of the way the InfiniBand adapters would behave in VILLASnode. For example, all plots in appendix F.4 show low median latencies, however, also latency peaks which are much higher than the median values.

Furthermore, the median latencies the round-trip benchmark yielded were marginally lower than the ones yielded by the custom one-way benchmark. This difference was probably caused by the abovementioned effect as well.

5.2.2 Variation of the MTU

Crupnicoff, Das, and Zahavi [CDZ05] report that the selected MTU does not affect the latency. Since the MTU can affect the latency in other technologies—such as Ethernet—this claim was examined. With `ib_send_lat`, it is fairly easy to change the MTU. All results of this test are displayed in Table 5.10. Since the UC service type is not officially supported by the RDMA CM QP, only results for the RC and UD service type are shown.

The table shows that no extraordinary peaks occurred. The only latency that stands out is marked red. However, since the difference is not substantial, and since this is the only occurrence of such a peak, it can be assumed that the MTU indeed does not affect latency.

5.2.3 RDMA CM queue pairs compared to regular queue pairs

In all implementations presented in the present work, it was assumed that the performance of a regular QP and a QP that is managed by the RDMA CM is almost identical. This assumption was evaluated as well.

Table 5.11 shows that the median latency for smaller messages was slightly smaller for regular QPs. For larger messages, this difference in latency diminished. This inconsiderable difference, however, does not outweigh the ease that comes with the RDMA communication manager. To get a latency decrease of less than 7 % (Table 5.11's worst case), a lot of complexity would have to be added to the source code, in order to efficiently manage the QPs.

Table 5.10: $\tilde{t}_{lat}$ [µs] as reported by `ib_send_lat` for different service types and message sizes with a varying MTU. All communication went over an RDMA CM QP and was sent with the normal *send* operation. Every test contained 1000 iterations and messages that were smaller than 188 B were sent inline.

service type	size	MTU				
		256 B	512 B	1024 B	2048 B	4096 B
RC	32 B	0.83	0.81	0.82	0.82	0.83
	1 kB	1.58	1.58	1.52	1.60	1.61
	4 kB	2.26	2.25	2.25	2.27	2.28
	32 kB	4.56	4.58	4.57	4.58	4.57
UD	32 B	0.86	0.87	0.86	0.87	0.86
	1 kB	✗	✗	1.44	1.54	1.45
	4 kB	✗	✗	✗	✗	2.22

Table 5.11: $\tilde{t}_{lat}$ [µs] as reported by `ib_send_lat` for different service types and queue pair types with a varying message size. All communication was sent with the normal *send* operation. Every test contained 1000 iterations and messages that were smaller than 188 B were sent inline.

service type	QP type	message size					
		32 B	128 B	512 B	2 kB	8 kB	32 kB
RC	regular	0.77	0.86	1.31	1.68	2.60	4.56
	RDMA CM	0.81	0.90	1.33	1.73	2.57	4.57
UD	regular	0.80	0.84	1.26	1.74	✗	✗
	RDMA CM	0.86	0.90	1.31	1.71	✗	✗

5.3 VILLASnode node-type benchmark

Again, all runs of the benchmark in this section were performed in the optimized environment as introduced in Figure 5.1 on the host system from Table 5.1.[3]

Timer of the signal node To find the timer that was best suited for the needs of the analyses that are discussed in this section, separate tests were performed and

[3] A small change to the environment had to be made: all tests that are presented in the following were performed with a customized version of the *latency-performance* `tuned` profile. The reason for this is discussed in the paragraph "Optimized environment" below.

their results are presented below. Since the ability to generate samples at high rates was a requirement for most of the analyses in the remainder of this section, a fixed, high rate of 100 kHz was set for the tests to analyze the timers. Four tests were prepared: two with a VILLASnode instance with a timer object that relies on a file descriptor for notifications (`timerfd`) and two with a timer that relies on the TSC. For the former, as can be seen in Table 5.12, more steps were missed at high rates. In the optimized environment, the file descriptor based implementation missed about 0.68 % of the signals, whereas the TSC based implementation only missed 0.50 % of the steps. Since the implementation with the least missed steps is preferred—after all, when steps are missed, the actual rate that is sent to the node-type under test is lower than the set rate—the TSC was chosen as timer for the following tests.

Appendix F.4 shows the histograms for these four tests, including the missed steps and an indicator for whether samples were not transmitted by the nodes that were tested. When comparing the median latencies of the four cases, it becomes apparent that the `timerfd` timer affected the measured $\tilde{t}_{lat}$ more than the TSC. Since this means that the benchmark's results with the TSC better reflect the actual performance of the node-type under test, this is another advantage of the TSC. Furthermore, in case of the unoptimized environment, the latency's variability with the `timerfd` timer was considerably worse than in the three other cases.

In later tests, it was also discovered that the TSC did not perform well with relatively small rates (≤ 2500 Hz). As it turned out, for the minimum rate of 100 Hz, approximately 8 % of the steps were missed. However, using the the `timerfd` timer for these low rates would noticeably skew the results, and a deviation of 8 Hz is unlikely to influence the latencies of the analyzed nodes. Therefore, the TSC was also used for these low rates.

Table 5.12: Comparison of the performance of timer functions. All tests were performed with a rate of 100 kHz, with 10 64-bit floating-point numbers per sample, RC as service type, and with the InfiniBand node-type as node-type under test. Every test contains 250 000 samples.

optimized environment	**missed samples**	
	`timerfd`	**TSC**
✗	$\frac{12085}{250000} \cdot 100\,\% = 4.83\,\%$	$\frac{3035}{250000} \cdot 100\,\% = 1.21\,\%$
✓	$\frac{1692}{250000} \cdot 100\,\% = 0.68\,\%$	$\frac{1244}{250000} \cdot 100\,\% = 0.50\,\%$

Optimized environment The tests that were done to analyze the behavior of the timers also revealed information about the effect of the optimized and unoptimized environment on latencies. As it turned out, using the *latency-performance* `tuned` profile was detrimental for the latency and the overall performance. This effect occurred regardless of the used environment. For the cases in Figure F.4, median la-

tencies increased about 700 ns, variability and maxima rose, and the `timerfd` timer missed up to 15 % of the steps. Further research has shown that the `force_latency` flag (line 6, Listing B.1) caused this problem. Therefore, in all tests that are presented in the following, a customized version of the *latency-performance* `tuned` profile without this flag was used.

Figure F.4 also reveals that running VILLASnode in the optimized environment was beneficial for latency. However, the difference between both environments was not tremendous. It is likely that the reason for this is that the testsystem from Table 5.1 was fully dedicated to the tests that were run on it. In a real life scenario, the system would be busy with other processes, and the difference in latency for processes in the shielded cpuset and in the normal pool of CPUs would presumably be larger.

Configuration of the InfiniBand nodes It was found that the number of buffers hardly influenced the performance of the *InfiniBand* node-type. Even MacArthur and Russel's "ideal" number of buffers—although impracticable for the purposes of this real-time framework—were investigated [MR12]. Apart from the fact that such a small number of buffers made it impossible to send samples bigger than a few byte at high frequencies, barely any difference in latency could be seen compared to cases with (a lot) more buffers.

A momentous difference, however, could be seen when the size of the receive queue and the number of mandatory work requests in the receive queue was varied. The situation with the lowest latency arose when the size and the number of WRs was chosen to be just big enough to support the highest combination of generation rate and message size. For example, in case of Figure F.4d, latency extrema around 262 µs could be seen with this ideal setup. For an arbitrary large number (e.g., a queue depth of 8192 and 8064 mandatory WRs in the queue), these extrema peaked at more than 3000 µs. This effect was caused by the way the *InfiniBand* node-type's read-function is implemented and probably occurred shortly after the initialization of the receiving *InfiniBand* node. As presented in Figure 4.1, the read-function first fills the receive queue, before it starts polling the queue and processing the data. When the threshold is large, it takes a certain amount of time before data can be processed. However, it is important to keep in mind that a larger receive queue yields a higher stability because overflows will be less likely.

For the send queue, the opposite is true: in order to signal as little as possible, the send queue can be as large as the HCA allows it to be. The signaling threshold, that describes the maximum number of unsignaled WRs before a signaled WR must be sent, is determined according to equation (1.1) in section 1.2. If one sample is sent per call of the write-function, which is true for all following tests,

$$S = \frac{D_{SQ}}{2} \tag{5.8}$$

follows from equation (1.1). Before running any of the following tests, it was verified that this threshold indeed yielded the lowest latency. It turned out that any higher or lower threshold yielded, although marginally, worse latencies.

The settings for the sending and the receiving *InfiniBand* node can be found in appendix E. These settings were used in all tests that are presented in this section.

5.3.1 Comparison between InfiniBand service types

This subsection presents the tests that were performed to examine how the different InfiniBand service types perform within VILLASnode. It solely focuses on the reliable connection and on unreliable datagrams since these two service types are officially supported by the RDMA CM, and thus require no modification of the RDMA CM library.

Varying the sample generation rate In the first set of tests, the rate with which samples were generated was varied between 100 Hz and 100 kHz. All tests were performed until 250 000 samples were transmitted. Each sample that was sent contained 8 random 64-bit floating-point numbers. For the reliable connection, this added up to

$$8 \cdot 8\,\mathrm{B} + 24\,\mathrm{B} = 88\,\mathrm{B} \tag{5.9}$$

per message, taking the 24-byte metadata into account. For unreliable datagrams, this number was

$$88\,\mathrm{B} + 40\,\mathrm{B} = 128\,\mathrm{B} \tag{5.10}$$

because the 40-byte GRH of the sending node was attached to every message. Since the messages were relatively small, they were all sent inline.

Figure 5.9 shows the results the VILLASnode node-type benchmark yielded with the abovementioned settings. Both service types showed an almost identical behavior, regardless of which rate was set: for both types, $\tilde{t}_{lat}$ decreased when the rate was increased. This is in line with prior observations in subsection 5.1.2, where latency increased when pauses between the transmission of messages were increased.

Characteristic for InfiniBand is the (almost) non-existent latency difference between messages on reliable connections and unreliable datagrams. Because, as discussed in section 2.1, reliability is handled in the HCA rather than in the operating system, it causes less overhead.

All tests for the *InfiniBand* node-type were only performed for signal generation rates up to 100 kHz. At higher frequencies, the *signal* node started to miss more and more steps. According to the latencies from section 5.1, the sample rate needs to be a lot higher than 100 kHz before the InfiniBand hardware becomes the bottleneck. Assuming a message resides about 1000 ns in the InfiniBand stack and network, rates up to:

$$\frac{1}{1000\,\mathrm{ns}} = 1\,\mathrm{GHz} \tag{5.11}$$

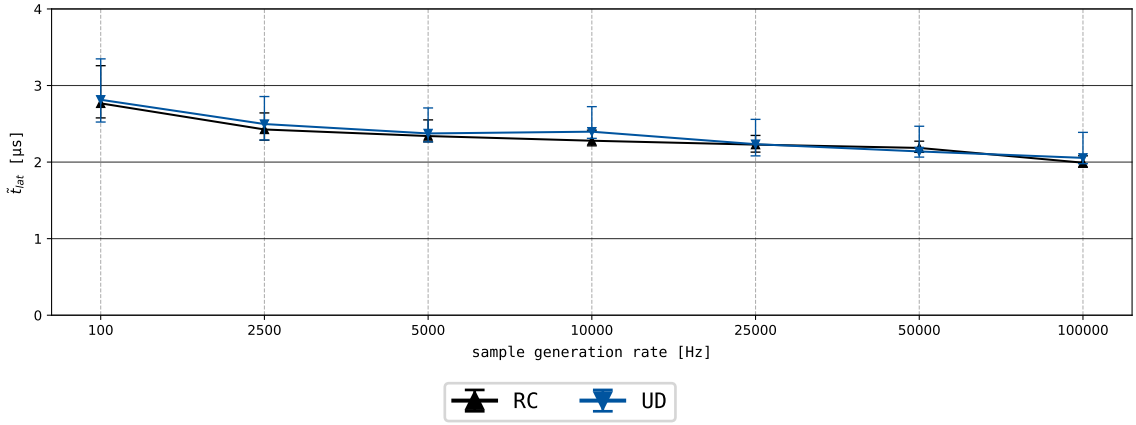


Figure 5.9: Results the benchmark yields for the *InfiniBand* node-type with a fixed message size of 88 B for RC and 128 B for UD. The sample generation rate was varied between 100 Hz and 100 kHz and for every rate, 250 000 samples were sent.

are theoretically possible with the numbers measured in the previous section.[4] However, two problems arise:

- The refresh rate of the buffers in the receive queue is not indefinitely high. As described in section 4.2, for its completion queue to be cleared and its receive queue to be refilled, an *InfiniBand* node depends on the rate with which the read-function is invoked. When the QP is chosen to be big enough, a node should be able to absorb short peaks in the message rate (e.g., 1 GHz) flawlessly. However, if the rate stays high for an extended amount of time, the buffers will overflow in the current setup.

  More on the theoretically achievable rate in subsection 5.3.2.
- Subsection 4.3.2 described optimizations that were applied to the *file* node-type. Even though these optimizations considerably increased the maximum signal generation rate, rates well above 100 kHz were still not achievable. Consequently, to increase this upper limit, the *file* node-type should be optimized further, so that the share it takes in the total datapath decreases.

Varying the sample size In the second set of tests, the generation rate was fixed to 25 kHz. The message size was varied between 1 and 64 values per sample. This resulted in messages between 32 B and 536 B for RC and 74 B and 576 B for UD. Based on the results from Table 5.8 and Figure 5.8, messages smaller than or equal to 188 B were sent inline.[5]

[4] This is only based on the measured time that a message congests the InfiniBand stack and network; it is assumed that WRs can be submitted to the QPs with this rate.

[5] Inline sizes that are powers of two are not supported by the Mellanox HCA used in the present work. The HCA automatically converts it to the closest value that is larger than the set value. In this case, 188 B is the closest value larger than 128 B.

The first observation to be made is the increasing median latency when messages become bigger than approximately 128 B. This is in line with the findings from subsection 5.1.7. Secondly, the variability of the reliable connection was consistently lower than the variability of unreliable datagram. This was not only true for high rates, but also for lower rates. Finally, it can be observed that the RC service type had a lower median latency than UD. This is remarkable, and a reason for this could be the fact that the receiving node's AH must be added to every work request when the UD service type is used. Furthermore, the GRH is added to every message that is sent with the UD service type.

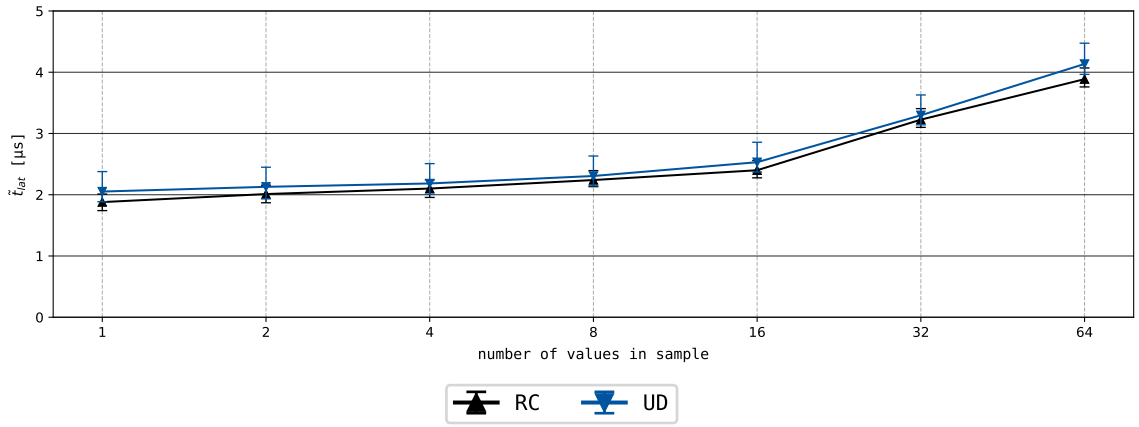


Figure 5.10: Results the benchmark yields for the *InfiniBand* node-type at a fixed sample generation rate of 25 kHz and a message size that was varied between 32 B and 536 B for RC and 74 B and 576 B for UD. For every message size, 250 000 samples were sent.

Varying both the sample size and generation rate Figure 5.11 aims to give a complete view on the influence of the several possible generation rate and message size combinations by combining the previously presented tests. Since the reliable connection shows—although only slightly—the lowest median latencies, this figure only depicts the measurements for RC. In this test, the generation rate was varied between 100 Hz and 100 kHz and the number of values in a sample between 1 and 64. All tests were performed until 250 000 samples were transmitted.

Figure 5.11 shows that, in accordance with Figure 5.10, the median latency increased with the message size. Additionally, as can be seen along the rate-axis, a higher message generation rate corresponded to a lower median latency. This could also be seen in Figure 5.9.

When the *signal* node missed more than 10 % of the steps for a particular sample rate/sample size combination, this is indicated with a red colored percentage in Figure 5.11. From these numbers, it becomes evident that the file-node was not

able to process large amounts of data. With tests that missed a substantial amount of samples, a threshold T can be approximated to:

$$T = \left(1 - \frac{P_{missed}}{100\,\%}\right) \cdot S_{sample} \cdot f_{signal} \qquad [\mathrm{B}] \cdot [\mathrm{Hz}] = [\mathrm{B/s}], \tag{5.12}$$

where P_{missed} is the percentage of missed samples, S_{sample} is the sample size, and f_{signal} the set signal generation rate. In case of the VILLASnode node-type benchmark, this value was approximately 20 MiB/s. This is, nevertheless, only a rough estimation; the signal generation rate probably has a higher impact on the threshold than the sample size.

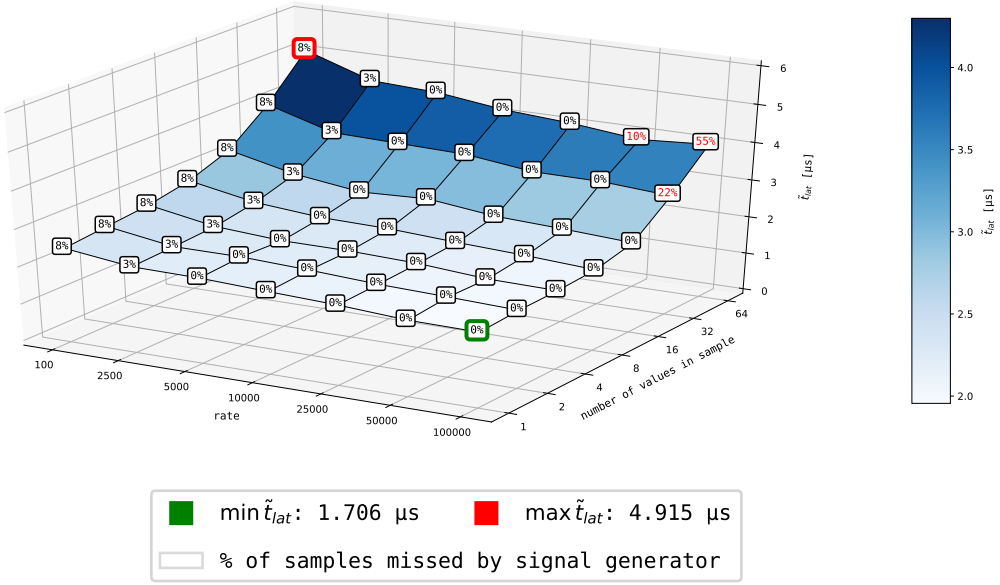


Figure 5.11: The influence of the message size and generation rate on the median latency between two *InfiniBand* nodes that communicate over an Reliable Connection (RC).

The fact that up to 8 % of the steps were missed at low rates with the TSC was already mentioned at the beginning of this section. Since these rates are non-critical for the node-types that were analyzed, it is improbable that a difference of 8 Hz in case of a set rate of 100 Hz, and 75 Hz in case of 2500 Hz, will noticeably affect the median latency. Using an alternative timer, however, would have considerably skewed the latencies in that range.

Appendix F.5 shows the same graphs for UC and UD, respectively. Both modes show a very similar behavior to the RC service type. As observed before, UD shows slightly higher median latencies than RC. UC, on the other hand, shows slightly lower median latencies. This backs the suspicion that was raised earlier, on why UD was slightly slower than RC. Regarding latency, UC does not have three major disadvantages of both types: it does not need to guarantee delivery of a message,

but it does also not require an AH with every WR and does not need to add the 40 B GRH to every message.

Thus, the smallest median latencies among the service types that are officially supported by the RDMA CM were observed for the reliable connection. When varying both the message size and generation rate, the minimum latency of about 1.7 µs was observed for high rates and low message sizes. The maximum latency was observed for high rates and low message sizes and was approximately 4.9 µs.

5.3.2 Comparison to the zero-latency reference

The first comparison to be done is between the *InfiniBand* node-type and the *shmem* node-type. The latter uses the POSIX shared memory API to enable communication between nodes over shared memory regions [Ker10]. Because the latency between two *shmem* nodes will approximately be the time it takes to access memory, its $\tilde{t}_{lat}$ can be approximated to the time $\tilde{t}_{villas}$. $\tilde{t}_{villas}$ is the amount of time that is spent by the super-node, apart from the nodes that are being tested. It thus corresponds to the time that is spent in all blocks of Figure 4.4, minus the time that is spent in the nodes that are being tested.

In the tests that were performed, the sample generation rate was varied between 100 Hz and 100 kHz, every sample contained 8 64-bit floating-point numbers, and for every rate, 250 000 samples were sent. The results of these tests can be seen in Figure 5.12. Compared to previous graphs, this graph additionally contains an indication of the missed steps of the *signal* node for generation rate.

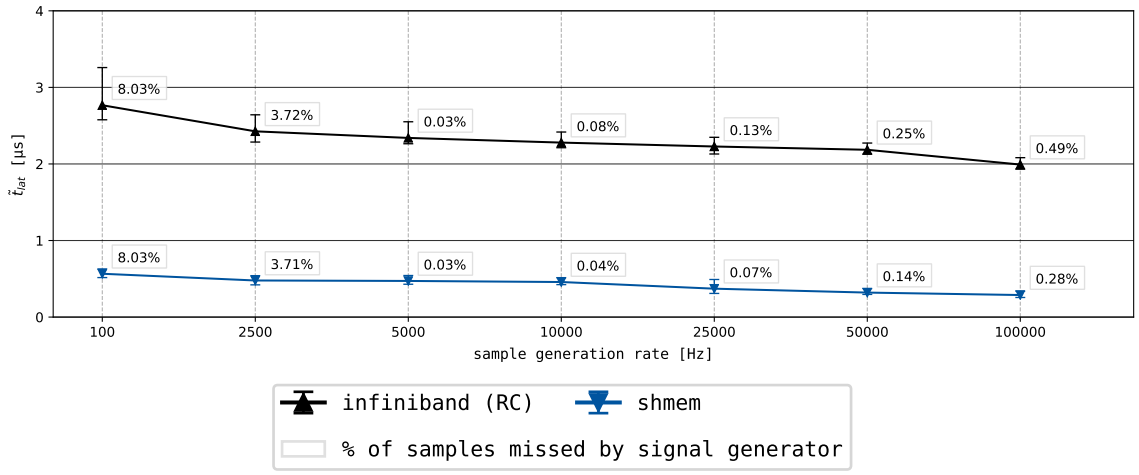


Figure 5.12: Results the benchmark yields for the *shmem* and *InfiniBand* node-type with 8 64-bit floating-point number per sample. The sample generation rate was varied between 100 Hz and 100 kHz and for every rate, 250 000 samples were sent.

Difference in latency The difference between the latencies of these node-types can be seen as the additional latency that communication over InfiniBand adds. The time penalty that the implementation of the read- and write-function add can be approximated to:

$$t^{IB}_{\text{r/w-function}} \approx \tilde{t}^{IB}_{lat} - \tilde{t}^{shmem}_{lat} - \tilde{t}^{HCA}_{lat}, \tag{5.13}$$

with $\tilde{t}^{IB}_{lat}$ the median latency that is measured when transmitting data between two InfiniBand VILLASnode nodes, $\tilde{t}^{shmem}_{lat}$ the median latency of communication between two *shmem* nodes, and $\tilde{t}^{HCA}_{lat}$ the latency that was seen for inline communication in section 5.1.

With $\tilde{t}^{IB}_{lat} \approx 2\,\mu s$, $\tilde{t}^{shmem}_{lat} \approx 0.3\,\mu s$, and $\tilde{t}^{HCA}_{lat} \approx 0.8\,\mu s$ this number adds up to approximately 0.9 µs. Since, as could be seen in subsection 5.1.2, up to 0.3 µs latency was added when the send rate decreased, the values for the highest frequency from Figure 5.12 were used. In that way, the added time should should mainly be caused by the implementations from subsection 4.2.3 and subsection 4.2.4.

Missed steps The graph shows that, in most cases, the *signal* node only missed slightly more steps when testing the *InfiniBand* node, than when testing the *shmem* node. This indicates that the *InfiniBand* node-type did not give much back pressure and that its write-function returned fast enough and did therefore not influence the signal generation at these rates. Since median latencies around 2500 ns were achieved, transmission rates up to

$$\frac{1}{2500\,\text{ns}} \approx 400\,\text{kHz} \tag{5.14}$$

should be possible. This number is probably more pessimistic than the reality since it does not take into account that the latency is not entirely caused by the sending node.

The same similarities could be seen for other sample sizes and sample generation rates. Appendix F.6 shows the results the benchmark yielded when the sample generation rate and the message size were varied for the *shmem* node-type. Regarding missed steps, this graph shows similarities to Figure 5.11 in this chapter and Figure F.5 and Figure F.6 in appendix F.5. Since the common denominator of these tests is the *file* node-type, these results again indicate that the component that caused the most complications in the VILLASnode node-type benchmark's datapath was the *file* node-type.

Thus, since the *file* node-type is currently the bottleneck in the benchmark from section 4.3, this node-type should be optimized in order to bring down the number of steps the benchmark misses.

Decline in latency Analogous to previous observations, the median latency of the *InfiniBand* node-type increased for lower frequencies. Remarkable, however, is that the median latency of the *shmem* node-type also increased—although only slightly—for lower frequencies. Even though this decline is not unambiguously visible in Figure 5.12, it is more evident in Figure F.7 in appendix F.

In a previous subsection the suspicion was raised that techniques such as ASPM caused this effect. But, since the same effect also occurred with node-types that are independent from the PCI-e bus, the cause of this problem cannot solely lie within I/O optimization techniques. Hence, the (scheduler of the) OS is probably also partially responsible for the increasing latency at lower rates.

5.3.3 Comparison to other node-types

The objective of the present work that was raised in subsection 1.1.3 was to implement hard real-time communication between different host systems that run VILLASnode. It showed that none of the server-server node-types that were available at the time of writing the present work were able to realize this (Table 1.1).

This subsection examines whether the addition of the *InfiniBand* node-type to the pool of available VILLASnode node-types has an added value. It does so by comparing the results of two commonly used node-types for server-server communication—*zeromq* and *nanomsg*—with the *InfiniBand* node-type and the *shmem* node-type.

In the tests that were performed, the sample size was fixed to 8 values. The rate was varied between 100 Hz and 100 kHz and every test was conducted until 250 000 messages were transmitted.

Loopback and physical link First, the tests were performed in loopback mode, in which the source and target node of the *zeromq* and *nanomsg* node-type were both bound to `127.0.0.1`. However, to make a fair comparison to the *InfiniBand* node-type tests, which were performed on an actual physical link, these tests had to be performed on a physical link as well.

To exclude that using different hardware with inferior or superior specifications would skew the results, the back-to-back connected InfiniBand HCAs were also used to perform the tests with the Ethernet based node-types. This was done using the Internet Protocol over InfiniBand (IPoIB) driver (subsection 2.3.3), which enables processes to send data over the InfiniBand network using the TCP/IP stack (Figure 2.20).

In order to compel processes to actually use the physical link although both network devices were part of the same system, the Linux network namespace was used. With namespaces,[6] it is possible to wrap system resources in an abstraction, so that they are only visible to processes in that namespace. In case of the network namespace, processes in such a namespace make use of a copy of the network stack. It can be seen as a separate subsystem, with its own routes, firewall rules, and network device(s). The network namespace was managed with `ip-netns`.[7]

Results Figure 5.13 shows the results of these runs. For rates below 25 kHz, the latencies of the loopback tests were almost identical to the latencies of the tests on

[6] http://man7.org/linux/man-pages/man7/namespaces.7.html
[7] http://man7.org/linux/man-pages/man8/ip-netns.8.html

the physical link. Above 25 kHz the latencies of the latter start to increase. Although especially the *zeromq* node showed a humongous latency increase, the performance of both node-types started to become unsuited for real-time simulations.

The percentage of missed steps for 100 Hz and 2500 Hz was exactly the same for the *nanomsg* and *zeromq* node-type as for the *InfiniBand* and *shmem* node-type. This again indicates that this effect was caused by the TSC. It is, however, unlikely that the relatively high median latencies around these rates were caused by the TSC. After all, in all previously presented tests in which the TSC was used for these rates, such a large difference was not seen.

Although a considerable number of samples were never transmitted, especially for high rates, no samples were dropped after the first sequence number appeared in the out file. The percentages of missed steps of the *nanomsg* and *zeromq* node-type are displayed in appendix F.7.

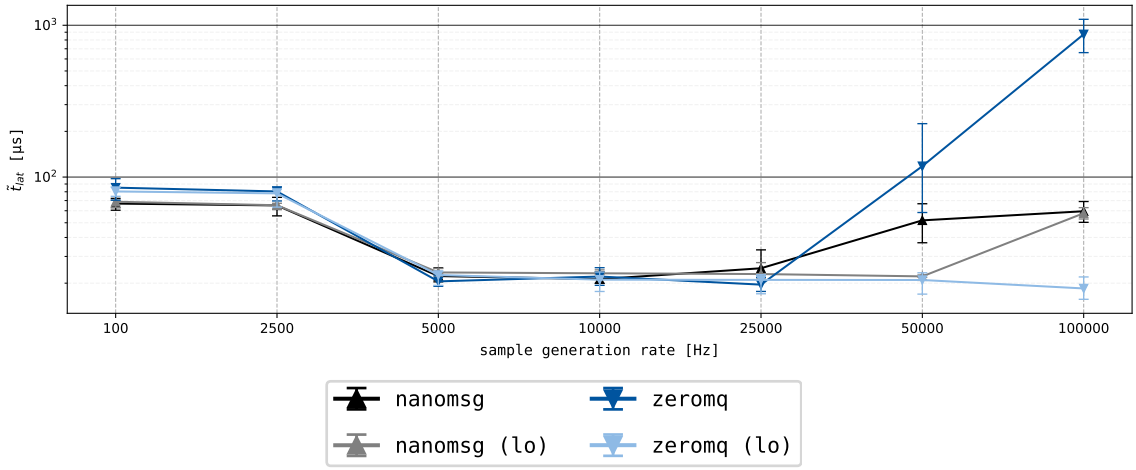


Figure 5.13: Results the benchmark yielded for the *zeromq* and *nanomsg* node-types. Both node-types were once tested in loopback mode and once over an actual physical link. Every sample contained 8 64-bit floating-point numbers and the sample generation rate was varied between 100 Hz and 100 kHz. For every rate, 250 000 samples were sent.

Figure 5.14 compares the results of the *nanomsg* and *zeromq* node-type on the physical link with the results of the *InfiniBand* and *shmem* node-type. It is apparent from this graph that the *InfiniBand* node-type had a latency that was one order of magnitude smaller than the soft real-time node-types. Furthermore, the variability of the latency of the samples that were sent over InfiniBand was lower than the variability of the latency of the same samples over Ethernet. Finally, both the *nanomsg* and *zeromq* node unmistakably started to show performance losses when exceeding a sample generation rate of 25 kHz.

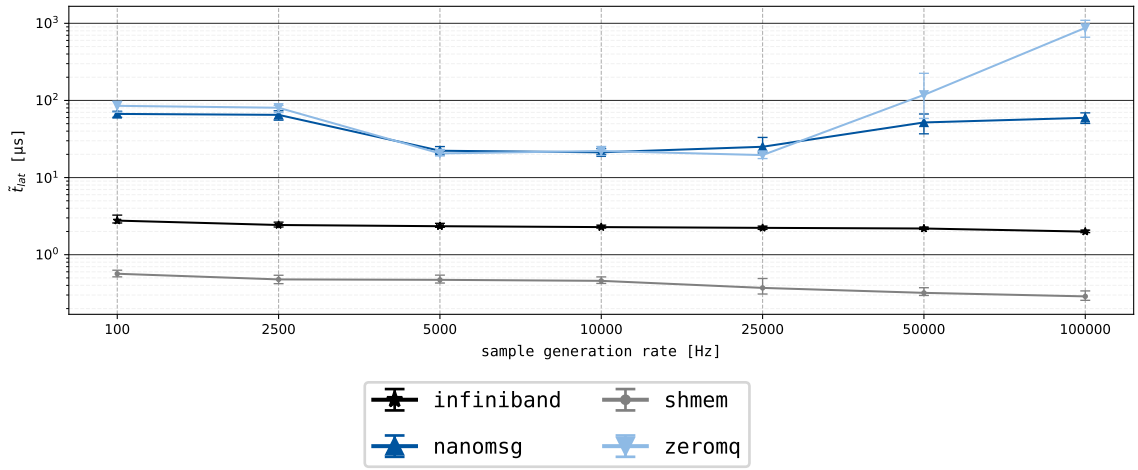


Figure 5.14: Results the benchmark yielded for the server-server node-types *zeromq*, *nanomsg*, and *InfiniBand* and for the internal node-type *shmem*. Every sample contained 8 64-bit floating-point numbers and the sample generation rate was varied between 100 Hz and 100 kHz. For every rate, 250 000 samples were sent.

6 Conclusion

The present work shows that the InfiniBand Architecture (IBA) enables the transmission of small messages at high rates with sub-microsecond latencies. Together with the presented performance optimizations, the IBA is eminently suitable as communication technology for applications with hard real-time requirements.

With only a few adaptions to the existing node-type interface and the buffer management of VILLASnode, it was possible to create an *InfiniBand* node-type that makes full use of the IBA's zero-copy capabilities and can initiate data transfers without needing system calls. The adaptions that have been made were necessary because complications, which were non-existent with prior node-types, emerged. Eventually, since the InfiniBand Architecture is rooted in the Virtual Interface Architecture (VIA), these alterations facilitate VILLASnode with an interface that is—with minimal adaptions—compatible to other VIAs as well.

In the custom benchmark that was used, the InfiniBand node-type showed median latency between approximately 1.7 µs (for high rates and small message sizes) and 4.9 µs (for low rates and large message sizes). Compared to the *shmem* node-type, which can be seen as zero-latency reference, these median latencies were only roughly 1.5–2.5 µs higher. This is an excellent achievement since the latter solely allows communication between nodes on the same host system. The former, on the other hand, also allows communication between nodes on different host systems. For comparison: prior node-types that allow communication between different host systems and rely on Ethernet as communication technique, showed median latencies which were one order of magnitude larger than the median latencies the *InfiniBand* node-type showed. Furthermore, with the new node-type, much higher transmission rates could be achieved and the latency's predictability substantially improved.

It can thus be concluded that the *InfiniBand* node-type is a valuable extension to the pool of existing VILLASnode node-types. Although the node-type is not suitable for inter-laboratory communication, it enables hard real-time scalability of simulation power within laboratories.

7 Future Work

7.1 Real-time optimizations

In his master's thesis [Vog16], Vogel wrote that "careful optimizations and tuning [of the Linux OS] are indispensable. The most important change is a `PREEMPT_RT`-patched kernel."

Real-time Linux was first presented by Barabanov and Yodaiken [BY96], and the `PREEMPT_RT` patch is currently maintained by Ingo Molnar and Thomas Gleixner. Its main purpose is not to increase the throughput of a Linux system or to decrease its latency, but rather to make it more predictable. It does so by:

- making parts of the kernel, which were originally not preemptible, preemptible;
- adding priority inheritance to the kernel;
- running interrupts as threads;
- replacing timers, which leads to high-resolution, user-space-accessible timers.

The internals of the RT patch are described by Rostedt and Hart [RH07] and can also be found on the Real-Time Linux Wiki.[1]

The Mellanox modified OFED™ stack that was used together with the Mellanox' HCAs (Table 5.1) did not support `PREEMPT_RT`-patched Linux kernels. Therefore, none of the benchmarks that were evaluated in chapter 5 could be run on a real-time operating system. Consequently, the predictability of the benchmark was not always ideal. Examples are:

- Figure 5.7: $\max t_{lat} = 17.4\,\mu s$ and $0.0125\,\%$ of $t_{lat} > 10\,\mu s$ with a median latency of only 786 ns;
- Figure F.4d: $\max t_{lat} = 262.0\,\mu s$ and $0.02\,\%$ of $t_{lat} > 50\,\mu s$ with a median latency of only $2.1\,\mu s$.

Although there is a chance that the median latencies will become a little higher with a `PREEMPT_RT`-patched kernel, these (sometimes excessive) latency spikes should diminish and the variability should decrease. Furthermore, the increasing latencies for lower transmission rates should diminish with an RT-patched kernel.

It would certainly be interesting for future research to examine the behavior of InfiniBand hardware in real-time optimized operating systems. Although InfiniBand is already an attractive communication solution for for real-time applications, this could make it even more attractive.

[1] https://rt.wiki.kernel.org

7.2 Optimization & profiling

Benchmark optimizations During the coarse of the present work, it turned out that the bottleneck of the benchmark from Figure 4.4 is the *file* node-type. Although several optimizations, e.g., suppressing as much system calls as possible, were applied to this node-type, it remained the bottleneck for high frequencies.

Reducing the effect the *file* node-type has on the benchmark would yield less distorted, more realistic indications of the latencies that can be achieved. More important, however, is that this would facilitate a method to examine the limitations of low-latency node-types such as *InfiniBand* and *shmem*.

Furthermore, it would be beneficial to evaluate whether the TSC can be optimized in a way that it works with low rates as well.

InfiniBand node-type optimizations Currently, the read- and write-function of the *InfiniBand* node-type add a latency penalty of roughly 0.9 µs to the transmission latency of a message. Since this is the lion's share of the total latency, it would be interesting to analyze how many time is spent in the several functions and what the hot spots are. Profiling tools like *gprof* [L+83] can be used for this kind of analysis.

Moreover, all settings were optimized for maximum rates. However, the optimal settings for lower rates probably differ from the optimal settings for high rates.

7.3 RDMA over Converged Ethernet support

In their publication [MR12], MacArthur and Russel observed that RoCE, which allows RDMA over conventional Ethernet networks, was just slightly outperformed by InfiniBand for small messages. Although RoCE's performance would be marginally worse than InfiniBand's and although it does not have as much support for QoS as InfiniBand, it would be a great addition to VILLASnode for cases in which existing infrastructure must be used.

With the alterations that have been made to VILLASnode in order to support InfiniBand, support for RoCE would not require too many changes to the existing source code.

Appendices

A OpenFabrics Verbs

Experimental functions are not included in this appendix. Furthermore, the RDMA verbs API is omitted because it is not used in the present work. A comprehensive documentation on all verbs can be found in the RDMA Aware Networks Programming User Manual [15].

A.1 IB verbs API

This section presents the default InfiniBand verbs API.

Table A.1: IB verbs

```
void ibv_ack_async_event(struct ibv_async_event *event)
```

Acknowledges events from `ibv_get_async_event()`. Events must be acknowledged before associated objects can be destroyed. This is done in order to avoid races.

```
void ibv_ack_cq_events(struct ibv_cq *cq, unsigned int nevents)
```

Acknowledges events from `ibv_get_cq_event()`. Events must be acknowledged before associated objects can be destroyed. This is done in order to avoid races. Calling this function is relatively expensive and it is possible to ackknowledge multiple events in one call.

```
struct ibv_pd * ibv_alloc_pd(struct ibv_context *context)
```

Allocates a Protection Domain (PD) for the given context.

```
int ibv_attach_mcast(struct ibv_qp *qp, const union ibv_gid *gid, uint16_t lid)
```

Attaches a UD QP to a multicast group with a given GID and LID.

```
int ibv_close_device(struct ibv_context *context)
```

Closes the context that was opened by `ibv_open_device()`.

```
int ibv_close_xrc_domain(struct ibv_xrc_domain *d)
```

Closes an eXtended Reliable Connection (XRC) domain. This will only succeed if no more QP or SRQ is associated with the XRC.

```
struct ibv_ah *ibv_create_ah(struct ibv_pd *pd, struct ibv_ah_attr *attr)
```

Creates an Address Handle (AH) out of AH attributes. These can be manually created or acquired via `rdma_get_cm_event()`.

```
struct ibv_ah * ibv_create_ah_from_wc(struct ibv_pd *pd, struct ibv_wc *wc,
struct ibv_grh *grh, uint8_t port_num)
```

Combines `ibv_init_ah_from_wc()` and `ibv_create_ah()`.

```
struct ibv_comp_channel *ibv_create_comp_channel(struct ibv_context *context)
```

Creates a new, unbound Completion Channel (CC).

```
struct ibv_cq * ibv_create_cq(struct ibv_context *context, int cqe, void
*cq_context, struct ibv_comp_channel *channel, int comp_vector)
```

Creates a Completion Queue (CQ). The variable `cq_context` is user defined and is returned as parameter in `ibv_get_cq_event()` if a Completion Channel (CC) is used.

```
struct ibv_qp * ibv_create_qp(struct ibv_pd *pd, struct ibv_qp_init_attr
*qp_init_attr)
```

Creates a Queue Pair (QP) in the *reset* state. All desired initial attributes must be placed in `*qp_init_attr`.

```
struct ibv_srq * ibv_create_srq(struct ibv_pd *pd, struct ibv_srq_init_attr
*srq_init_attr)
```

Creates a Shared Receive Queue (SRQ). All desired initial attributes must be placed in `*srq_init_attr`. An SRQ serves as RQ for several QPs and must be passed to `ibv_create_qp()` with `*qp_init_attr`. It is used in order to save resources.

```
int ibv_create_xrc_rcv_qp(struct ibv_qp_init_attr *init_attr, uint32_t
*xrc_rcv_qpn)
```

Creates an eXtended Reliable Connection (XRC) Queue Pair (QP).

```
struct ibv_srq * ibv_create_xrc_srq(struct ibv_pd *pd, struct ibv_xrc_domain
*xrc_domain, struct ibv_cq *xrc_cq, struct ibv_srq_init_attr *srq_init_attr)
```

Creates an eXtended Reliable Connection (XRC) Shared Receive Queue (SRQ).

```
int ibv_dealloc_pd(struct ibv_pd *pd)
```

Deallocates a Protection Domain (PD). Fails if objects are still associated with the given PD.

```
int ibv_dereg_mr(struct ibv_mr *mr)
```

Destroys a Memory Region (MR). This will only succeed if no more MWs are associated to the MR.

```
int ibv_destroy_ah(struct ibv_ah *ah)
```

Destroys an Address Handle (AH).

```
int ibv_destroy_comp_channel(struct ibv_comp_channel *channel)
```

Destroys an Completion Channel (CC). This will only succeed if no more CQ is associated with the CC.

```
int ibv_destroy_cq(struct ibv_cq *cq)
```

Destroys an Completion Queue (CQ). This will only succeed if no more QP is associated with the CQ.

```
int ibv_destroy_qp(struct ibv_qp *qp)
```

Destroys a Queue Pair (QP).

```
int ibv_destroy_srq(struct ibv_srq *srq)
```

Destroys a Shared Receive Queue (SRQ). This will only succeed if no more QP is associated with this Shared Receive Queue (SRQ).

```
int ibv_detach_mcast(struct ibv_qp *qp, const union ibv_gid *gid, uint16_t lid)
```

Detaches a UD QP from the multicast group with a given GID and LID.

```
const char * ibv_event_type_str(enum ibv_event_type event_type)
```

Translates an enumeration that is included in an event returned by `ibv_get_async_event()` into a string.

```
int ibv_fork_init(void)
```

Initializes the data structure to handle `fork()` safely.

```
void ibv_free_device_list(struct ibv_device **list)
```

Frees the list that was previously returned by `ibv_get_device_list()`.

```
int ibv_get_async_event(struct ibv_context *context, struct ibv_async_event
*event)
```

Retrieves asynchronous events from the devices. This function is a blocking function.

```
int ibv_get_cq_event(struct ibv_comp_channel *channel, struct ibv_cq **cq, void
**cq_context)
```

Retrieves notifications from a Completion Channel (CC) that is bound to one or more CQs. This function is a blocking function.

```
uint64_t ibv_get_device_guid(struct ibv_device *device)
```

Returns the 64-bit GUID of a device returned by `ibv_get_device_list()`.

```
struct ibv_device ** ibv_get_device_list(int *num_devices)
```

Returns a list, including name and other properties, of devices in the system that support the IB verbs.

```
const char * ibv_get_device_name(struct ibv_device *device)
```

Returns a pointer to the name of a device returned by `ibv_get_device_list()`.

```
int ibv_init_ah_from_wc(struct ibv_context *context, uint8_t port_num, struct
ibv_wc *wc, struct ibv_grh *grh, struct ibv_ah_attr *ah_attr)
```

Initializes an Address Handle (AH) in `*ah_attr`, based on a CQE of a received message.

```
int ibv_modify_qp(struct ibv_qp *qp, struct ibv_qp_attr *attr, enum
ibv_qp_attr_mask attr_mask)
```

Modifies the attributes and the state of a Queue Pair (QP). Attribute changes and transitions are subject to strict rules that can be found in [15] and [07].

```
int ibv_modify_srq (struct ibv_srq *srq, struct ibv_srq_attr *srq_attr, int
srq_attr_mask)
```

Modifies the attributes which are specified in the bitmask `srq_attr_mask` with the values in `*srq_attr`.

```
int ibv_modify_xrc_rcv_qp(struct ibv_xrc_domain *xrc_domain, uint32_t xrc_qp_num,
struct ibv_qp_attr *attr, int attr_mask)
```

Modifies the attributes which are specified in the bitmask `attr_mask` with the values in `*attr`. The QP is then transitioned through *reset* → *init* → *started*.

```
const char * ibv_node_type_str (enum ibv_node_type node_type)
```

Returns the type of the device: HCA, switch, router, RDMA enabled NIC, or unknown.

```
struct ibv_context * ibv_open_device(struct ibv_device *device)
```

Opens a device returned by `ibv_get_device_list()` and returns a context that can be used with all verbs that directly modify the device.

```
struct ibv_xrc_domain * ibv_open_xrc_domain(struct ibv_context *context, int fd,
int oflag)
```

Opens and eXtended Reliable Connection (XRC) domain for the device.

```
int ibv_poll_cq(struct ibv_cq *cq, int num_entries, struct ibv_wc *wc)
```

Retrieves Completion Queue Entries (CQEs) from the Completion Queue (CQ).

```
const char * ibv_port_state_str (enum ibv_port_state port_state)
```

Returns a string that describes the enumeration `port_state`.

```
int ibv_post_recv(struct ibv_qp *qp, struct ibv_recv_wr *wr, struct ibv_recv_wr
**bad_wr)
```

Submits a linked list of Work Requests (WRs) to the Receive Queue (RQ). Processing will stop on the first error and the erroneous WR will be returned via `**bad_wr`. At least one WR must be placed in the RQ to transition to the state *ready to receive*.

```
int ibv_post_send(struct ibv_qp *qp, struct ibv_send_wr *wr, struct ibv_send_wr
**bad_wr)
```

Submits a linked list of Work Requests (WRs) to the Send Queue (SQ). All communication is initialized through this function. Processing will stop on the first error and the erroneous WR will be returned via `**bad_wr`.

```
int ibv_post_srq_recv(struct ibv_srq *srq, struct ibv_recv_wr *recv_wr, struct
ibv_recv_wr **bad_recv_wr)
```

Submits a linked list of Work Requests (WRs) to a given Shared Receive Queue (SRQ). This function is similar to `ibv_post_recv()`.

```
int ibv_query_device(struct ibv_context *context, struct ibv_device_attr
*device_attr)
```

Retrieves an extensive list of attributes of a device, e.g., maximum MR size, maximum number of QPs, maximum number of CQs, vendor ID, node and system image GUID, and hardware version.

```
int ibv_query_gid(struct ibv_context *context, uint8_t port_num, int index, union
ibv_gid *gid)
```

Retrieves an entry from the port's GID table.

```
int ibv_query_pkey(struct ibv_context *context, uint8_t port_num, int index,
uint16_t *pkey)
```

Retrieves an entry from the port's partition key table.

```
int ibv_query_port(struct ibv_context *context, uint8_t port_num, struct
ibv_port_attr *port_attr)
```

Retrieves and extensive list of attributes of a port, e.g., maximum MTU and message size.

```
int ibv_query_qp(struct ibv_qp *qp, struct ibv_qp_attr *attr, enum
ibv_qp_attr_mask attr_mask, struct ibv_qp_init_attr *init_attr)
```

Retrieves the attributes which are specified in the bitmask `attr_mask` from the Queue Pair (QP).

```
int ibv_query_srq(struct ibv_srq *srq, struct ibv_srq_attr *srq_attr)
```

Similar to `ibv_query_qp()`, but for a Shared Receive Queue (SRQ).

```
int ibv_query_xrc_rcv_qp(struct ibv_xrc_domain *xrc_domain, uint32_t xrc_qp_num,
struct ibv_qp_attr *attr, int attr_mask, struct ibv_qp_init_attr *init_attr)
```

Similar to `ibv_query_qp()`, but for an eXtended Reliable Connection (XRC).

```
struct ibv_mr * ibv_reg_mr(struct ibv_pd *pd, void *addr, size_t length, enum
ibv_access_flags access)
```

Registers a Memory Region (MR) associated with a PD. The returned struct `ibv_mr` contains the local and remote key.

```
int ibv_reg_xrc_rcv_qp(struct ibv_xrc_domain *xrc_domain, uint32_t xrc_qp_num)
```

Registers a user process with the defined eXtended Reliable Connection (XRC) receive Queue Pair (QP).

```
int ibv_req_notify_cq(struct ibv_cq *cq, int solicited_only)
```

Informs the Completion Queue (CQ) about the fact that it must send completion events to a Completion Channel (CC). This works only for Completion Queue Entries (CQEs) that were not yet present in the CQ when this function was called.

```
int ibv_resize_cq(struct ibv_cq *cq, int cqe)
```

Resizes the Completion Queue (CQ). The new size must be bigger than the number of CQEs present in the CQ.

```
int ibv_unreg_xrc_rcv_qp(struct ibv_xrc_domain *xrc_domain, uint32_t xrc_qp_num)
```

Unregisters a user process from the defined eXtended Reliable Connection (XRC) receive Queue Pair (QP).

A.2 RDMA CM API

This section presents the RDMA communication manager API, as presented in subsection 2.3.3.

Table A.2: RDMA CM verbs

```
int rdma_accept(struct rdma_cm_id *id, struct rdma_conn_param *conn_param)
```

Accepts a connection request or datagram service lookup. This function is called on the passive, listening side.

```
int rdma_ack_cm_event(struct rdma_cm_event *event)
```

Acknowledges events from `ibv_get_cm_event()`. Events must be acknowledged before associated objects can be destroyed. This is done in order to avoid races.

```
int rdma_bind_addr(struct rdma_cm_id *id, struct sockaddr *addr)
```

Binds the communication identifier to a local RDMA device that is associated with the source IP address.

```
int rdma_connect(struct rdma_cm_id *id, struct rdma_conn_param *conn_param)
```

Initiates a connection request. When relying on a connected service type, this requests a connection to a remote location. In case of an unconnected service type, this requests all information from the remote QP to send datagrams. Before calling `rdma_connect()`, `rdma_resolve_route()` must have been called.

```
int rdma_create_ep(struct rdma_cm_id **id, struct rdma_addrinfo *res, struct
ibv_pd *pd, struct ibv_qp_init_attr *qp_init_attr)
```

Creates a communication endpoint. If `*qp_init_attr` is provided, a QP to track communication information will be created. If no PD is defined, it will be associated with the default PD. Furthermore, it creates an communication identifier that operates synchronously. To let it operate asynchronously, it must be bound to a Completion Channel (CC) using `rdma_migrate_id()`.

```
struct rdma_event_channel * rdma_create_event_channel(void)
```

Creates a new, unbound communication event channel.

```
int rdma_create_id(struct rdma_event_channel *channel, struct rdma_cm_id **id,
void *context, enum rdma_port_space ps)
```

Creates an RDMA CM communication identifier. Although similar to sockets, the `rdma_cm_id` must be bound to a device before communication can occur.

```
int rdma_create_qp(struct rdma_cm_id *id, struct ibv_pd *pd, struct
ibv_qp_init_attr *qp_init_attr)
```

Creates a Queue Pair (QP) that is associated to an communication identifier. The QP's state transitions are managed by the identifier.

```
int rdma_destroy_ep(struct rdma_cm_id *id)
```

Destroys the communication endpoint and all associated resources.

```
void rdma_destroy_event_channel(struct rdma_event channel *channel)
```

Destroys a communication event channel. Prior to destroying the channel, all associated communication identifiers must have been destroyed and all events must have been acknowledged.

```
int rdma_destroy_id(struct rdma_cm_id *id)
```

Destroys an RDMA CM communication identifier. Prior to destroying the identifier, all associated QPs must have been destroyed and all events must have been acknowledged.

```
void rdma_destroy_qp(struct rdma_cm_id *id)
```

Destroys a Queue Pair (QP) associated with the communication identifier.

```
int rdma_disconnect(struct rdma_cm_id *id)
```

Disconnects any QP and transitions them tot the *error* state.

```
char * rdma_event_str(enum rdma_cm_event_type event)
```

Translates an enumeration that is included in an event returned by `ibv_get_cm_event()` into a string.

```
void rdma_free_devices(struct ibv_context **list)
```

Destroys the array that was retrieved by `rdma_get_devices()`.

```
void rdma_freeaddrinfo(struct rdma_addrinfo *res)
```

Frees the address that was retrieved with `rdma_getaddrinfo()`. This function is very similar to `freeaddrinfo()` [Ker10].

```
int rdma_get_cm_event(struct rdma_event_channel *channel, struct rdma_cm_event
**event)
```

Retrieves asynchronous events from the communication event channel. This function is a blocking function.

```
struct ibv_context ** rdma_get_devices(int *num_devices)
```

Retrieves an array of available RDMA devices in the system.

```
uint16_t rdma_get_dst_port(struct rdma_cm_id *id)
```

Returns the port number of a communication identifier's peer endpoint. If the identifier is unconnected, the function shall return `'0'`.

```
struct sockaddr * rdma_get_local_addr(struct rdma_cm_id *id)
```

Retrieves the local `sockaddr` address of a communication identifier.

```
struct sockaddr * rdma_get_peer_addr(struct rdma_cm_id *id)
```

Retrieves the remote `sockaddr` address of a communication identifier. If the identifier is unconnected, the function shall fill the complete `sockaddr` C structure with zeros.

```
int rdma_get_request(struct rdma_cm_id *listen, struct rdma_cm_id **id)
```

Retrieves the next pending connection request event from a synchronously operating communication identifier. If the function returns successfully, it will create a new communication identifier that represents the connection.

```
uint16_t rdma_get_src_port(struct rdma_cm_id *id)
```

Returns the local port number of a communication identifier.

```
int rdma_getaddrinfo(char *node, char *service, struct rdma_addrinfo *hints,
struct rdma_addrinfo **res)
```

Resolves the destination node and service address and returns all information to communicate with a remote node in `**res`. This function is very similar to `getaddrinfo()` [Ker10].

```
int rdma_join_multicast(struct rdma_cm_id *id, struct sockaddr *addr, void
*context)
```

Joins a multicast group and attaches a Queue Pair (QP) that is associated to the communication identifier.

```
int rdma_leave_multicast(struct rdma_cm_id *id, struct sockaddr *addr)
```

Leaves a multicast group and detaches a Queue Pair (QP) that is associated to the communication identifier.

```
int rdma_listen(struct rdma_cm_id *id, int backlog)
```

Sets the communication identifier to listening mode for incoming connection requests. Prior to calling this function, the device must have been bound to a local device with `rdma_bind_addr()`.

```
int rdma_migrate_id(struct rdma_cm_id *id, struct rdma_event_channel *channel)
```

Migrates a communication identifier and all its pending events to a new communication event channel.

```
int rdma_notify(struct rdma_cm_id *id, enum ibv_event_type event)
```

Notifies the communication identifier about events that have occured on a QP that is associated to it. Usually, this is not necessary. However, it can be necessary if the QP was created out of band and the communication identifier does not know its status yet.

```
int rdma_reject(struct rdma_cm_id *id, const void *private_data, uint8_t
private_data_len)
```

Rejects a connection request or datagram service lookup. This function is called on the passive, listening side.

```
int rdma_resolve_addr(struct rdma_cm_id *id, struct sockaddr *src_addr, struct
sockaddr *dst_addr, int timeout_ms)
```

Resolves a destination IP address to a valid RDMA address. If, additionally, a source IP address is provided, the RDMA CM communication identifier will be bound to that device. The latter is only necessary if `rdma_bind_addr()` has not been called yet.

```
int rdma_resolve_route(struct rdma_cm_id *id, int timeout_ms)
```

Resolves the route to a given destination address. Prior to calling this function, `rdma_resolve_addr()` must have been called on a destination address.

```
int rdma_set_option(struct rdma_cm_id *id, int level, int optname, void *optval,
size_t optlen)
```

Sets options for a communication identifier. The options can be found in the `rdma_cma.h` header file.[1]

[1] https://github.com/linux-rdma/rdma-core/blob/master/librdmacm/rdma_cma.h

B Tuned daemon profile

This appendix shows the *latency-performance* `tuned` profile that was used during the benchmarks that were run on the HCAs and VILLASnode.

```
[main]
summary=Optimize for deterministic performance at the cost of
increased power consumption

[cpu]
force_latency=1
governor=performance
energy_perf_bias=performance
min_perf_pct=100

[sysctl]
kernel.sched_min_granularity_ns=10000000
vm.dirty_ratio=10
vm.dirty_background_ratio=3
vm.swappiness=10
kernel.sched_migration_cost_ns=5000000
```

Listing B.1: The `tuned` default profile *latency-performance*. Comments are omitted for the sake of brevity.

C VILLASnode node-type interface

Table C.1: Function pointers from the `node_type` C structure (Listing D.3), which are used to register node-type functions with the super-node.

```
int (*check)(struct node *n);
```

This function is executed once for every instance of a node-type and shall check if all previously parsed settings are valid.

```
int (*destroy)(struct node *n);
```

This function is the counterpart of `int (*start)(struct super_node *sn);` and can be used as global de-initialization of certain node-type.

```
int (*fd)(struct node *n);
```

This function must return a file descriptor, which enables the node to send notifications to the super-node. This is necessary when the super-node multiplexes incoming data from several nodes, and only wants to call a node's read-function when data is actually available.

```
int (*parse)(struct node *n, json_t *cfg);
```

The number of instances to be created of a certain node-type and their settings are specified in a JSON file and decoded using the Jansson C library.[1] This function is executed once for every instance of a node-type and shall interpret all settings which are passed to the instance via the parameter `json_t *cfg`.

```
char * (*print)(struct node *n);
```

When invoked, this function shall return a string with a textual representation of this instance of the node-type.

[1] http://www.digip.org/jansson

```
int (*read)(struct node *n, struct sample *smps[], unsigned cnt, unsigned
*release);
```

This function forms—together with the write-function—the core of every node-type instance. By calling this function, the super-node passes an empty structure which can hold `cnt` samples. The node-type instance than fills `*smps[]` with $0 <$ `ret` $<$ `cnt` values, where `ret` is also the value that is returned by the function.
Every sample contains a reference counter which is incremented before the sample is passed to a node. Usually, after the read (or write) function returns, these samples are released for re-use by decrementing the reference counter. There are certain situations in which it is undesirable to release the samples after read (or write) is called. The `*release` value lets the node define how many samples from `*smps[]` may be released on return. `*release` is initialized to `cnt`; this indicates the default case in which all samples are released after the struct returns.
An example of this situation for the InfiniBand node can be found in subsection 4.2.3.

```
int (*reverse)(struct node *n);
```

This function may be called by the super-node between the parse- and the start-node-function. It will interchange the source and target address information and can be used to initialize a source and destination node-type instance with the same configuration file. This function is mainly used for debugging environments.

```
int (*start)(struct node *n);
```

This function is executed once for every instance of a node-type and is used to initialize resources which are unique for every node.

```
int (*start)(struct super_node *sn);
```

This function is executed once when at least one instance of a certain node-type is present in the super-node. It can be used as global initialization to create shared resources for a certain node-type.

```
int (*stop)(struct node *n);
```

This function is the counterpart of `int (*start)(struct node *n);` and shall be used to stop operation of the node-type instance.

```
int (*stop)();
```

This function shall free the memory of an instance of a node-type.

```
int (*write)(struct node *n, struct sample *smps[], unsigned cnt, unsigned
*release);
```

The parameters of this function are similar to those of the read-function. The super-node provides `cnt` samples, which it wants the node to send to its target. The return value of this function indicates the number of samples which were successfully sent. By changing `*release`, the node-type instance can define that only the first `*release` elements from `*smps[]` should be released for re-use on return of the function.
An example of this situation for the InfiniBand node can be found in subsection 4.2.4.

D VILLASnode structs

This appendix presents a few structures which help to understand the VILLASnode architecture from chapter 3. A full overview of all header files can be found on the VILLASnode Git repository.[1]

D.1 `struct sample`

```
1  struct sample {
2      uint64_t sequence;
3      int length;
4      int capacity;
5      int flags;
6
7      struct list *signals;
8
9      atomic_int refcnt;
10     ptrdiff_t pool_off;
11
12     struct {
13         struct timespec origin;
14         struct timespec received;
15     } ts;
16
17     union signal_data data[];
18 };
```

Listing D.1: The C structure of a VILLASnode sample.

[1] https://git.rwth-aachen.de/acs/public/villas/VILLASnode

D.2 struct node

```
1  struct node_direction {
2          int enabled;
3          int builtin;
4          int vectorize;
5
6          struct list hooks;
7
8          json_t *cfg;
9  };
10
11 struct node
12 {
13         char *name;
14         char *_name;
15         char *_name_long;
16
17         int affinity;
18
19         uint64_t sequence;
20
21         struct stats *stats;
22
23         struct node_direction in, out;
24
25         struct list signals;
26
27         enum state state;
28
29         struct node_type *_vt;
30         void *_vd;
31
32         json_t *cfg;
33 };
```

Listing D.2: The C structure of a VILLASnode node.

D.3 struct node_type

```
1  struct node_type {
2          int vectorize;
3          int flags;
4
5          enum state state;
6
7          struct list instance;
8
9          size_t size;
10         size_t pool_size;
11
12         struct {
13                 // Global, per node-type
14                 int (*start)(struct super_node *sn);
15                 int (*stop)();
16         } type;
17
18         // Function pointers
19         void * (*create)();
20         int (*init)();
21         int (*destroy)(struct node *n);
22         int (*parse)(struct node *n, json_t *cfg);
23         int (*check)(struct node *n);
24         char * (*print)(struct node *n);
25         int (*start)(struct node *n);
26         int (*stop)(struct node *n);
27
28         int (*read)(struct node *n, struct sample *smps[],
29                     unsigned cnt, unsigned *release);
30
31         int (*write)(struct node *n, struct sample *smps[],
32                      unsigned cnt, unsigned *release);
33
34         int (*reverse)(struct node *n);
35
36         int (*fd)(struct node *n);
37
38         // Memory Type
39         struct memory_type * (*memory_type)(struct node *n,
40                                 struct memory_type *parent);
41 };
```

Listing D.3: The C structure of a VILLASnode node-type.

E InfiniBand node configuration

```
source_node = {
    type = "infiniband",
    rdma_transport_mode = "${IB_MODE}",

    in = {
        address = "10.0.0.2:1337",

        max_wrs = 4,
        cq_size = 4,
        buffer_subtraction = 2
    },
    out = {
        address = "10.0.0.1:1337",
        resolution_timeout = 1000,
        send_inline = true,
        max_inline_data = 128,
        use_fallback = true,

        max_wrs = 4096,
        cq_size = 4096,
        periodic_signaling = 2048
    }
},

target_node = {
    type = "infiniband",
    rdma_transport_mode = "${IB_MODE}",

    in = {
        address = "10.0.0.1:1337",

        max_wrs = 512,
        cq_size = 512,
        buffer_subtraction = 64,

        signals = {
            count = ${NUM_VALUE},
            type = "float"
        }
    }
}
```

Listing E.1: The configuration that was used to examine the InfiniBand node-type with the benchmark from Figure 4.4. The bash variables were replaced by a script that controlled the benchmark.

F Results benchmarks

F.1 Influence of CQEs on latency of RDMA write

Table F.1: The benchmark's settings which were used to analyze the influence of Completion Queue Entry (CQE) creation on latency for *RDMA write* operations.

	service type	polling (send)	polling (recv)	inline mode	unsignaled	operation	burst size	repetitions	loop pauses	timestamp	message size
Figure F.1	UC	busy	busy	✓	✗	rdma	8000	20	0 ns	t_{subm}	32 B
Figure F.1	UC	busy	busy	✓	✓	rdma	8000	20	0 ns	t_{subm}	32 B

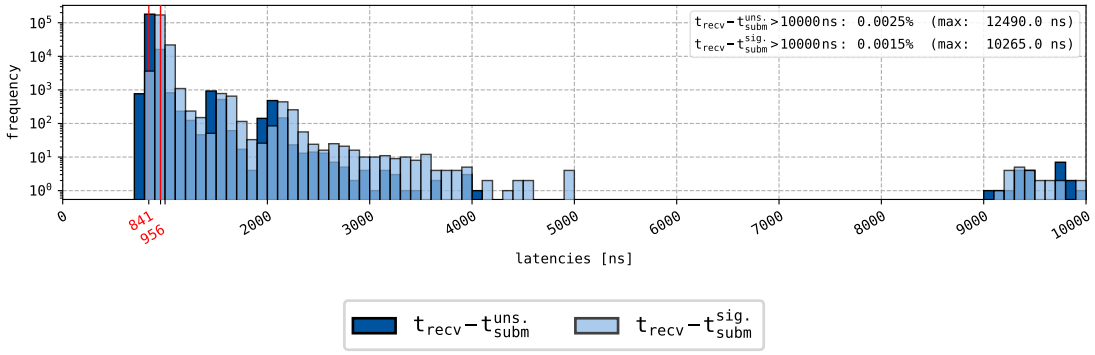
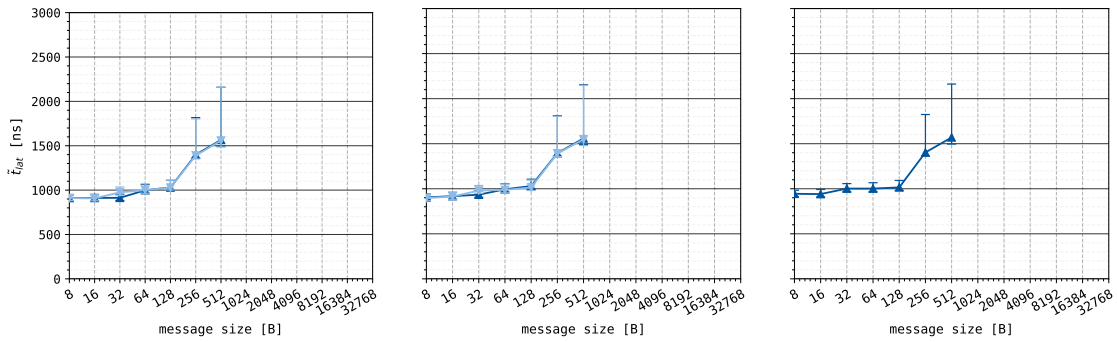


Figure F.1: Results of the one-way benchmark with the settings from Table F.1. These were used to analyze the difference in latency between messages that did and did not cause a Completion Queue Entry (CQE). The *RDMA write* operation mode was used in this test.

F.2 Influence of constant burst size on latency

Table F.2: The benchmark's settings which were used to analyze whether the steep slope between 128 B and 256 B in Figure 5.8 was caused by the non-constant burst sizes, with $j \in [0, 7]$.

	service type	polling (send)	polling (recv)	inline mode	unsignaled	operation	burst size	repetitions	loop pauses	timestamp	message size
Figure F.2a ▲	RC	busy	busy	✓	✗	send	2730	10	0 ns	t_{subm}	$8 \cdot 2^j$ B
Figure F.2a ▼	RC	busy	busy	✓	✗	rdma	2730	10	0 ns	t_{subm}	$8 \cdot 2^j$ B
Figure F.2b ▲	UC	busy	busy	✓	✗	send	2730	10	0 ns	t_{subm}	$8 \cdot 2^j$ B
Figure F.2b ▼	UC	busy	busy	✓	✗	rdma	2730	10	0 ns	t_{subm}	$8 \cdot 2^j$ B
Figure F.2c ▲	UD	busy	busy	✓	✗	send	2730	10	0 ns	t_{subm}	$8 \cdot 2^j$ B



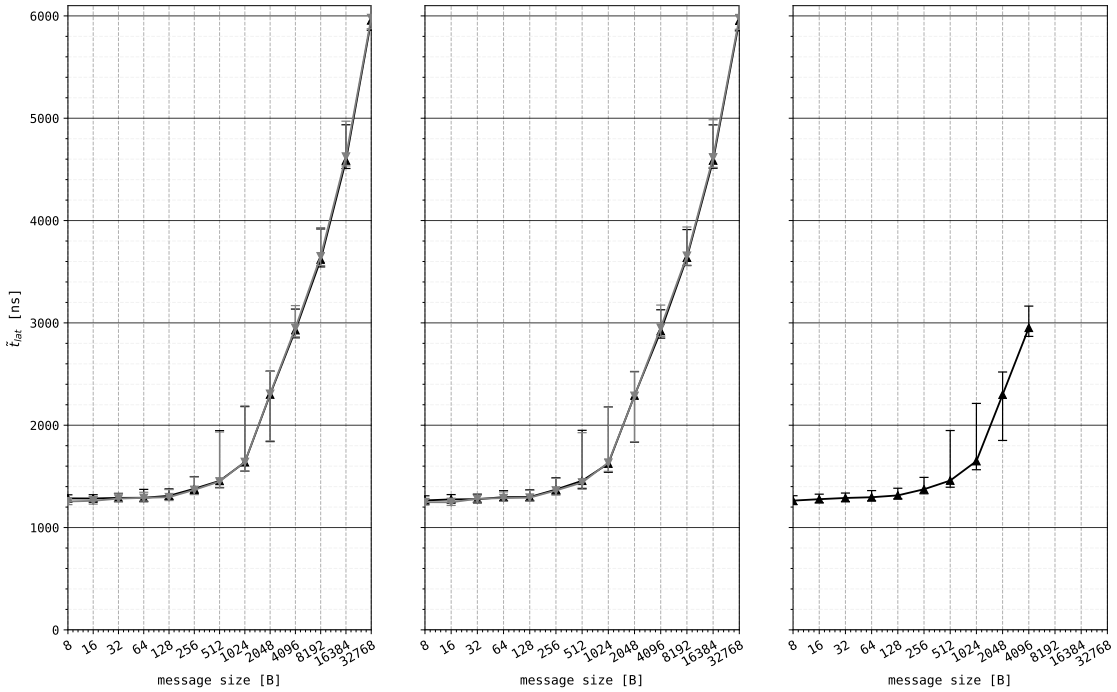
(a) RC (b) UC (c) UD

Figure F.2: Results of the one-way benchmark with the settings from Table F.2. While a triangle indicates $\tilde{t}_{lat}$ for a certain message size, the error bars indicate the upper and lower 10% of t_{lat} for that message size.

F.3 Influence of intermediate pauses on latency

Table F.3: The benchmark's settings which were used to analyze whether the increasing latency in Figure 5.8 was caused by congestion control, with $i \in [0, 12]$.

	service type	polling (send)	polling (recv)	inline mode	unsignaled	operation	burst size	repetitions	loop pauses	timestamp	message size
Figure F.3a ▲	RC	busy	busy	✗	✗	send	8000	5	5500 ns	t_{subm}	$8 \cdot 2^i$ B
Figure F.3a ▼	RC	busy	busy	✗	✗	rdma	8000	5	5500 ns	t_{subm}	$8 \cdot 2^i$ B
Figure F.3b ▲	UC	busy	busy	✗	✗	send	8000	5	5500 ns	t_{subm}	$8 \cdot 2^i$ B
Figure F.3b ▼	UC	busy	busy	✗	✗	rdma	8000	5	5500 ns	t_{subm}	$8 \cdot 2^i$ B
Figure F.3c ▲	UD	busy	busy	✗	✗	send	8000	5	5500 ns	t_{subm}	$8 \cdot 2^i$ B



(a) RC (b) UC (c) UD

Figure F.3: Results of the one-way benchmark with the settings from Table F.3. While a triangle indicates $\tilde{t}_{lat}$ for a certain message size, the error bars indicate the upper and lower 10% of t_{lat} for that message size.

F.4 Comparison of timer functions

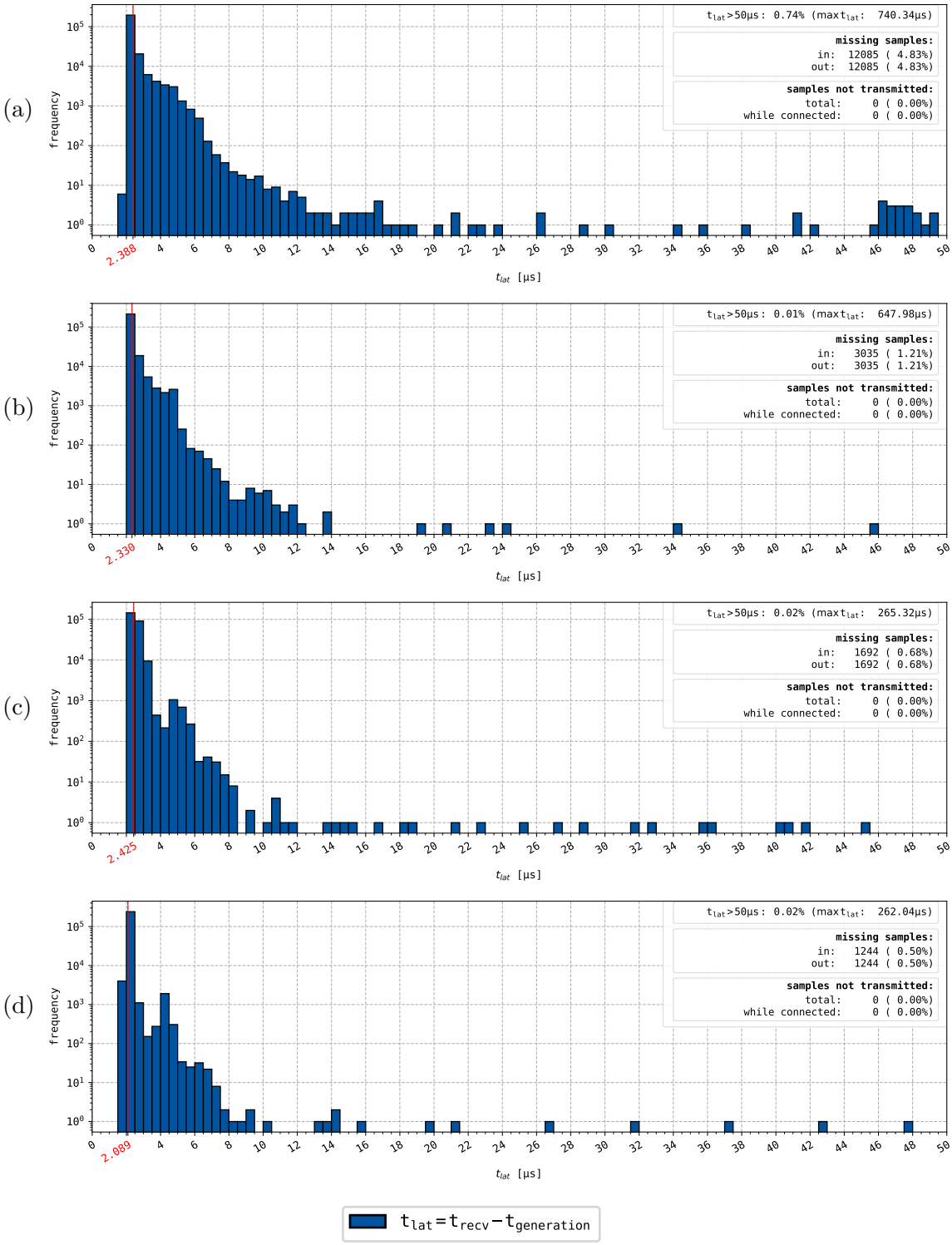


Figure F.4: Comprehensive plots of the results from Table 5.12. Subfigure (a) and (b) show the results in the unoptimized environment with `timerfd` and TSC, respectively. Subfigure (c) and (d) show the results for the same settings, but in the optimized environment.

F.5 3D plots InfiniBand nodes (UC & UD)

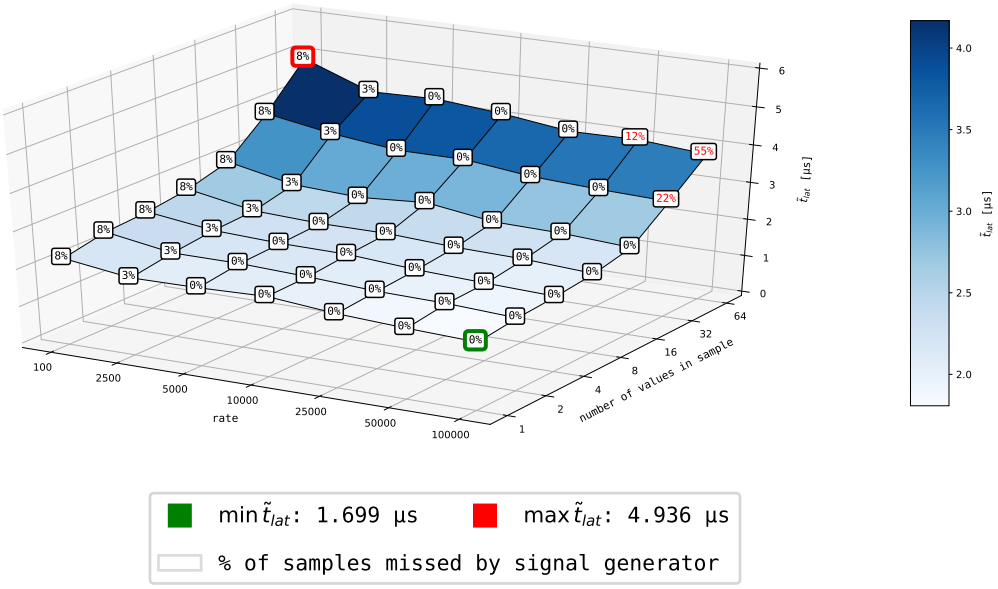


Figure F.5: The influence of the message size and generation rate on $\tilde{t}_{lat}$ between two InfiniBand nodes that communicate over an Unreliable Connection (UC).

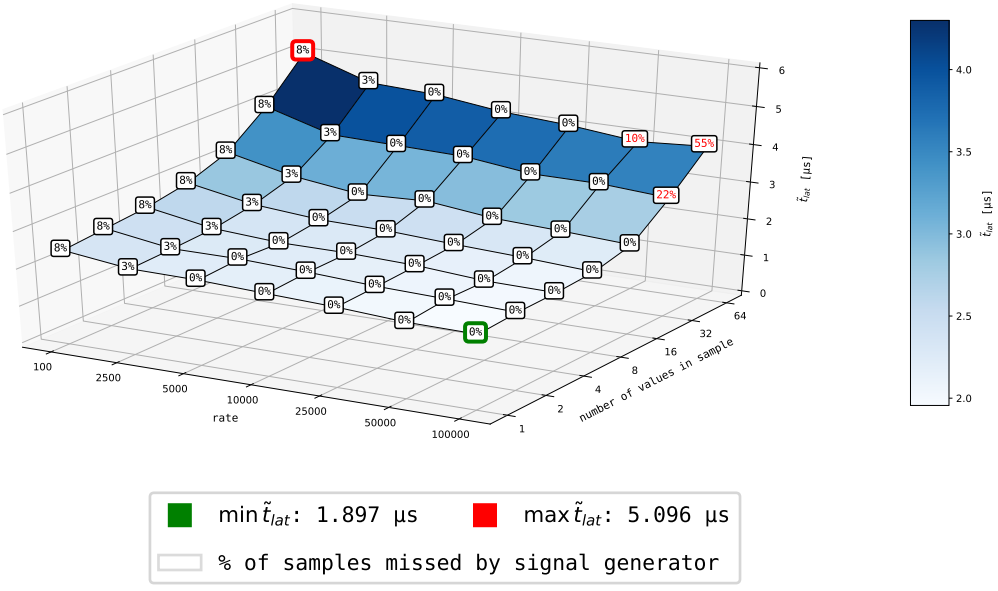


Figure F.6: The influence of the message size and generation rate on $\tilde{t}_{lat}$ between two InfiniBand nodes that communicate over Unreliable Datagram (UD).

F.6 3D plot shmem node

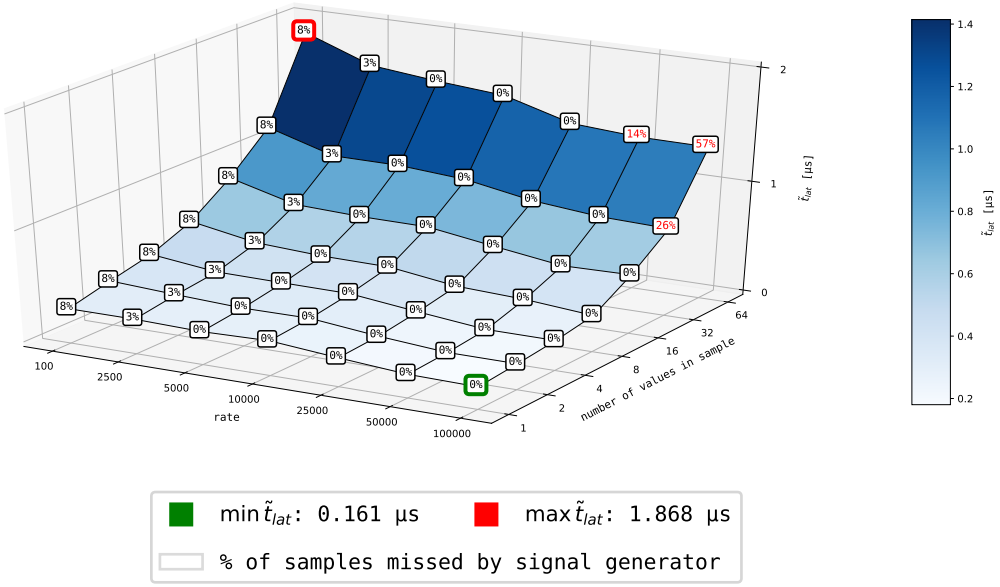


Figure F.7: The influence of signal generation rate and the message size on the median latency between two *shmem*.

F.7 Missed steps nanomsg and zeromq nodes

Table F.4: The percentage of missed steps in the in and out files that were generated by the VILLASnode node-type benchmark for the *nanomsg* and *zeromq* node-type. Although a considerable number of samples never got transmitted, especially for high rates, no samples were dropped after the first sequence number appeared in the out files.

rate [Hz]	**file**	**Missed steps [%]**			
		nanomsg	*nanomsg (lo)*	*zeromq*	*zeromq (lo)*
100	in	8.03	8.03	8.03	8.03
	out	8.04	8.03	8.04	8.04
2500	in	3.72	3.71	3.72	3.71
	out	3.80	3.71	3.78	3.78
5000	in	0.03	0.03	0.03	0.04
	out	0.20	0.03	0.15	0.18
10 000	in	0.04	0.05	0.04	0.07
	out	0.36	0.05	0.27	0.34
25 000	in	0.08	0.08	0.11	0.11
	out	0.90	0.08	0.70	0.76
50 000	in	0.17	0.17	0.24	0.22
	out	1.75	0.17	1.42	1.58
100 000	in	0.54	0.99	0.45	0.61
	out	3.91	1.00	2.68	3.33

List of Figures

List of Tables

List of Listings

Bibliography

[07] S.a. *InfiniBand™ Architecture Specification, Volume 1*. Release 1.2.1. InfiniBand Trade Association et al. Nov. 2007.

[10] S.a. *PCI Express® Base Specification*. Revision 3.0. PCI-SIG. Nov. 2010.

[15] S.a. *RDMA Aware Networks Programming User Manual*. Rev 1.7. Mellanox Technologies. May 2015.

[16] S.a. *InfiniBand™ Architecture Specification Volume 2*. Release 1.3.1. InfiniBand Trade Association et al. Nov. 2016.

[17] S.a. *Guidelines for Use of Extended Unique Identifier (EUI), Organizationally Unique Identifier (OUI), and Company ID (CID)*. Institute of Electrical and Electronics Engineers. Aug. 2017.

[18a] S.a. *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX®)*. Base Specifications, Issue 7. Institute of Electrical and Electronics Engineers. Jan. 2018. ISBN: 978-1-5044-4542-9. DOI: 10.1109/IEEESTD.2018.8277153.

[18b] S.a. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Volume 3A: System Programming Guide, Part 1. Intel. May 2018.

[18c] S.a. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Volume 3B: System Programming Guide, Part 2. Intel. May 2018.

[18d] S.a. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Volume 2B: Instruction Set Reference, M-U. Intel. May 2018.

[18e] S.a. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Volume 2A: Instruction Set Reference, A-L. Intel. May 2018.

[18f] S.a. *Mellanox OFED for Linux User Manual*. 2877. Rev 4.3. Mellanox Technologies. Mar. 2018.

[18g] S.a. *OFA Overview*. OpenFabric Alliance, 2018. URL: https://www.openfabrics.org/ofa-overview/ (visited on 08/22/2018).

[97] S.a. *Virtual Interface Architecture Specification*. Version 1.0. Compaq, Intel, Microsoft. Dec. 1997.

[Bow+09] Terrehon Bowden et al. *The /proc Filesystem*. Version 1.3. June 2009. URL: https://www.kernel.org/doc/Documentation/filesystems/proc.txt (visited on 09/19/2018).

[BY96] Michael Barabanov and Victor Yodaiken. “Real-Time Linux”. In: *Linux journal* 23.4.2 (1996), p. 1.

[CDZ05] Diego Crupnicoff, Sujal Das, and Eitan Zahavi. *Deploying Quality of Service and Congestion Control in InfiniBand-based Data Center Networks*. Tech. rep. 2379. 2005.

[Der+04] Simon Derr et al. *Cpusets*. 2004. URL: https://www.kernel.org/doc/Documentation/cgroup-v1/cpusets.txt (visited on 09/16/2018).

[Dre07] Ulrich Drepper. *What Every Programmer Should Know About Memory*. Version 1.0. Red Hat, Inc., Nov. 2007.

[Dun+98] Dave Dunning et al. “The Virtual Interface Architecture”. In: *IEEE micro* 18.2 (Mar. 1998), pp. 66–76. ISSN: 0272-1732. DOI: 10.1109/40.671404.

[Far+15] MD Omar Faruque et al. “Real-Time Simulation Technologies for Power Systems Design, Testing, and Analysis”. In: *IEEE Power and Energy Technology Systems Journal* 2.2 (June 2015), pp. 63–73. ISSN: 2332-7707. DOI: 10.1109/JPETS.2015.2427370.

[Gan+16] Jayneel Gandhi et al. “Range Translations for Fast Virtual Memory.” In: *IEEE Micro* 36.3 (May 2016), pp. 118–126. ISSN: 0272-1732. DOI: 10.1109/MM.2016.10.

[Kas15] Kashyap, V. *IP over InfiniBand (IPoIB) Architecture*. Internet Engineering Task Force. May 2015.

[Ker10] Michael Kerrisk. *The Linux Programming Interface: a Linux and UNIX System Programming Handbook*. No Starch Press, 2010. ISBN: 978-1-59327-220-3.

[Koz05] Charles M Kozierok. *The TCP/IP-Guide: A Comprehensive, Illustrated Internet Protocols Reference*. No Starch Press, 2005. ISBN: 978-1593270476.

[KR78] Brian W Kernighan and Dennis M Ritchie. *The C Programming Language*. 1st ed. Feb. 1978. ISBN: 0-13-110163-3.

[Kro03] Greg Kroah-Hartman. “udev–A Userspace Implementation of devfs”. In: *Proceedings of the Linux symposium*. July 2003, pp. 263–271.

[L+83] Susan L. et al. “gprof: A Call Graph Execution Profiler”. In: *Proceedings: USENIX Association [and] Software Tools Users Group Summer Conference*. Toronto, Ontario, Canada, 1983, pp. 81–88.

[Lam13] Christoph Lameter. “NUMA (Non-Uniform Memory Access): An Overview”. In: *Queue* 11.7 (July 2013), pp. 40–51. ISSN: 1542-7730. DOI: 10.1145/2508834.2513149.

[Lar+09] Steen Larsen et al. “Architectural breakdown of end-to-end latency in a TCP/IP network”. In: *International Journal of Parallel Programming* 37.6 (Dec. 2009), pp. 556–571. ISSN: 1573-7640. DOI: 10.1007/s10766-009-0109-6.

[Lov10] Robert Love. *Linux Kernel Development*. Pearson Education, Inc., June 2010. ISBN: 978-0-672-32946-3.

[LR14] Qian Liu and Robert D. Russell. "A Performance Study of InfiniBand Fourteen Data Rate (FDR)". In: *Proceedings of the High Performance Computing Symposium*. HPC '14. Tampa, Florida: Society for Computer Simulation International, 2014, pp. 1–10.

[Mir+18] Markus Mirz et al. "Distributed Real-Time Co-Simulation as a Service". In: *Industrial Electronics for Sustainable Energy Systems (IESES), 2018 IEEE International Conference on*. Hamilton, New Zealand: IEEE, Feb. 2018, pp. 534–539. DOI: 10.1109/IESES.2018.8349934.

[MJL08] Paul Menage, Paul Jackson, and Christoph Lameter. *Cgroups*. 2008. URL: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt (visited on 09/16/2018).

[MR12] Patrick MacArthur and Robert D Russell. "A Performance Study to Guide RDMA Programming Decisions". In: *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*. IEEE, 2012, pp. 778–785. DOI: 10.1109/HPCC.2012.110.

[Pao10] Gabriele Paoloni. *How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures*. Tech. rep. Sept. 2010.

[Pfi01] Gregory F Pfister. "An Introduction to the Infiniband™ Architecture". In: *High Performance Mass Storage and Parallel I/O* 42 (2001), pp. 617–632.

[PG07] Fernando Pérez and Brian E. Granger. "IPython: a System for Interactive Scientific Computing". In: *Computing in Science and Engineering* 9.3 (May 2007), pp. 21–29. ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.53. URL: https://ipython.org.

[Rei+06] S. Reinemo et al. "An Overview of QoS Capabilities in InfiniBand, Advanced Switching Interconnect, and Ethernet". In: *IEEE Communications Magazine* 44.7 (Sept. 2006), pp. 32–38. ISSN: 0163-6804. DOI: 10.1109/MCOM.2006.1668378.

[RH07] Steven Rostedt and Darren V Hart. "Internals of the RT Patch". In: *Proceedings of the Linux symposium*. Vol. 2. June 2007, pp. 161–172.

[Ste+17] Marija Stevic et al. "Multi-site European framework for real-time co-simulation of power systems". In: *IET Generation, Transmission & Distribution* 11.17 (2017), pp. 4126–4135. ISSN: 1751-8687. DOI: 10.1049/iet-gtd.2016.1576.

[Str+15] Thomas Strasser et al. "A Review of Architectures and Concepts for Intelligence in Future Electric Energy Systems". In: *IEEE Transactions on Industrial Electronics* 62.4 (Apr. 2015), pp. 2424–2438. ISSN: 0278-0046. DOI: 10.1109/TIE.2014.2361486.

[TB14] Andrew S Tanenbaum and Herbert Bos. *Modern Operating System*. 4th ed. Pearson Education, Inc, 2014. ISBN: 978-0-13-359162-0.

[Vog+17] Steffen Vogel et al. "An Open Solution for Next-generation Real-time Power System Simulation". In: *Energy Internet and Energy System Integration (EI2), 2017 IEEE Conference on*. IEEE, Nov. 2017, pp. 1–6. DOI: 10.1109/EI2.2017.8245739.

[Vog16] Steffen Vogel. "Development of a modular and fully-digital PCIe-based interface to Real-Time Digital Simulator". MA thesis. Institute for Automation of Complex Power Systems, Aug. 2016.