Designing an InfiniBand Metric Collector and Exploring InfiniBand Management Overhead and Scalability

Sjur Tveito Fredriksen

Thesis submitted for the degree of Master in Programming and Networks

60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2017
Designing an InfiniBand Metric Collector and Exploring InfiniBand Management Overhead and Scalability

Sjur Tveito Fredriksen
Abstract

The InfiniBand (IB) communication standard has over the last decade emerged to be one of the most popular interconnects used in high-performance computing. Deployment of IB networks have grown in terms of size and scale, and it is becoming a challenging task to keep track of the behavior of these networks. Monitoring of the performance of an IB network and overseeing errors and problems that might occur in the network is critical for the network administrators. There are already some tools that can help the network administrators, but many of these tools are old, and the user interface has failed to evolve with the network administrators expectations. Furthermore, little research has been done on how monitoring of an IB network influences application traffic or on how well switches handle repeated in-depth querying of their performance counters.

The first goal of this thesis was to build a robust, efficient, and scalable IB performance monitor plugin for the Fabriscale Fabric Manager and monitoring software suite. The developed monitoring plugin swipes the network continuously and updates the Fabriscale Fabric Manager with metrics almost instantly so that network characteristics can be presented in real-time to the network administrators using a modern web-based graphical user interface. The developed plugin is of great value to Fabriscale by offloading the Fabriscale Subnet Manager, and it helps to reduce the time needed to reconfigure devices in a subnet when a fault is occurring. Thus the plugin helps network administrators to minimize downtime and to get improved utilization of the cluster.

The second goal of this thesis was to inspect how fabric monitoring impacts the network, and how the network is affected by the metric collection. That is, this thesis looked on how multiple switches from multiple vendors handle in-depth querying of performance counters. It investigated how much extra bandwidth is generated by the monitoring software developed in this thesis, and how this can scale with the size of the network. Information about how the monitoring impacts the fabric is useful for Fabriscale and in turn network administrators when configuring the monitoring. We found that using in-band collection of metrics produces a minuscule amount of overhead and that the switches are handling being queried for their performance counters as often as every few milliseconds well.
# Contents

1 Introduction 1

1.1 Background and Motivation . . . . . . . . . . . . . . . . . . 1
1.2 Problem Statement . . . . . . . . . . . . . . . . . . . . . . . 4
1.3 Research Methods . . . . . . . . . . . . . . . . . . . . . . . 5
1.4 Thesis Outline . . . . . . . . . . . . . . . . . . . . . . . . . 6
1.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2 Technical Background 9

2.1 InfiniBand Overview . . . . . . . . . . . . . . . . . . . . . . 9
  2.1.1 Physical Layer . . . . . . . . . . . . . . . . . . . . . . 10
  2.1.2 Link Layer . . . . . . . . . . . . . . . . . . . . . . . . 13
  2.1.3 Network Layer . . . . . . . . . . . . . . . . . . . . . . 16
  2.1.4 Transport Layer . . . . . . . . . . . . . . . . . . . . . 16
  2.1.5 Management Model . . . . . . . . . . . . . . . . . . . 20

2.2 Monitoring the Fabric . . . . . . . . . . . . . . . . . . . . . . 26
  2.2.1 Current Monitoring Tools . . . . . . . . . . . . . . . . 26

2.3 Fabriscale Technologies . . . . . . . . . . . . . . . . . . . . . . 28
  2.3.1 Existing Fabriscale Software . . . . . . . . . . . . . . . 28
  2.3.2 Enhancing the Metric Collection . . . . . . . . . . . . 29

2.4 Libraries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
  2.4.1 Libibmad and Libibumad . . . . . . . . . . . . . . . . 31
  2.4.2 Google Protobuf . . . . . . . . . . . . . . . . . . . . . 31
  2.4.3 Zero Message Queue . . . . . . . . . . . . . . . . . . . 32

2.5 IBSim . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

2.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

3 Design and Implementation 37

3.1 Area of Use . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
3.2 Programming Language . . . . . . . . . . . . . . . . . . . . . 38
3.3 Requirements . . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.4 Specifications . . . . . . . . . . . . . . . . . . . . . . . . . . 39
3.5 Architectural drawing . . . . . . . . . . . . . . . . . . . . . . 40
3.6 Metric Representation . . . . . . . . . . . . . . . . . . . . . 40
3.7 Finding the Switches to Monitor . . . . . . . . . . . . . . . . 43
  3.7.1 Changes in Fsmonitoring . . . . . . . . . . . . . . . . 44
  3.7.2 Designing the Request and Response Protocol . . . . . 44
6 Conclusion and Future Work
6.1 Conclusion ........................................... 109
6.2 Future Work .......................................... 110

Glossary .................................................... 113
List of Figures

2.1 Logical view of an IB subnet with connected components[12]. 11
2.2 The OSI reference model[15]. 11
2.3 The IBA layers[12]. 12
2.4 A IBA PHY frame[12]. 12
2.5 IBA data packet seen at the link layer[12]. 14
2.6 IBA LRH[12]. 14
2.7 Overview of the IBA layers[13]. 19
2.8 Base MAD[12]. 20
2.9 PerfMgt GMP[12]. 24
2.10 The FFM software architecture. 29
2.11 Error and performance metrics displayed in the FFM web-based GUI. 30

3.1 Architectural view of ZeroMQ pipes when perfswiper is fully integrated. 41
3.2 Call flow in perfswiper when issuing a RPC request. 52

4.1 Observed max bidirectional bandwidth in a PCIe 1.1 environment. 69
4.2 Measured response time from the PMA on the different switches. 76
4.3 Experiment A1: Subnet topology with switches and a compute node. 78
4.4 Experiment A2: Subnet topology with switches and compute nodes. 78
4.5 Experiment A3: Subnet topology with switches and compute nodes. 79
4.6 Experiment D1: Subnet topology with switches and a compute node. 82

5.1 Experiment A1: Response times from switches in a chained network topology with no load. 87
5.2 Experiment A1: The 50 individual queries that compose the average of the first switch in the chain. 88
5.3 Experiment A2: Measured CPU time. 90
5.4 Experiment A2: Measured WALL response time. 90
5.5 Experiment A2: Response times based on the degree of network load. .................. 91
5.6 Experiment A3: Measured CPU time. .................. 93
5.7 Experiment A3: Measured WALL response times. ....... 93
5.8 Experiment A3: Response times based on the degree of network load. .................. 94
5.9 Experiment B: WALL time results. .................. 96
5.10 Experiment B: CPU time results. .................. 97
5.11 Experiment C1: Results from hardware. .................. 99
5.12 Experiment C1: Results from simulation. .................. 99
5.13 Experiment C2: Results from hardware. .................. 101
5.14 Experiment D1: Real-life test on a fat-tree topology. .................. 102
# List of Tables

2.1 IB Link Characteristics[12]. . . . . . . . . . . . . . . . . . . . 13
2.2 IBA Service Types[12]. . . . . . . . . . . . . . . . . . . . . . . 18
2.3 IBA performance counters[12]. . . . . . . . . . . . . . . . . . 25

4.1 Switch specifications[46][47][48]. . . . . . . . . . . . . . . . . 66
4.2 Compute node specifications. . . . . . . . . . . . . . . . . . . . . 67
4.3 Network load with different message sizes used. . . . . . . . . 70
4.4 Specifications of the machine running simulations. . . . . . . 70

5.1 Experiment D1: Average response times . . . . . . . . . . . . 103
List of Listings

2.1 An example .proto file[33]. ................................. 33
3.1 struct perf_metric in perfswiper.h. .................. 42
3.2 Protobuf message PSNodeInfo. .......................... 46
3.3 Default perfswiper configuration file. ................. 50
3.4 String utils used to get key and value from the configuration file. ................................. 51
3.5 pma_query_via() in mad.h. .............................. 53
3.6 IB_PM_EXT_WIDTH_SUPPORTED definition in ib_types.h. .......................... 54
3.7 Protobuf format used when transmitting metrics to fsmonitoring. ......................... 58
3.8 SIGTERM handler in perfswiper ......................... 60
3.9 perfswiper.service file used by systemd ............. 61
4.1 PCIe information about our HCAs. ........................ 68
This page is dedicated to the memory of my grandparents who passed away during my work with this thesis.

Dagfinn Tveito
Else Tveito
Astrid Fredriksen
Acknowledgements

This master thesis was written at the Department of Informatics at the Faculty of Mathematics and Natural Sciences, at the University of Oslo (UiO) in 2016/2017. This thesis was a collaboration between UiO, Fabriscale Technologies AS, and Simula Research Laboratory. Fabriscale Technologies provided the thesis topic, supervision, and equipment. Simula Research Laboratory provided supervision, additional equipment, and a professional working environment.

I would like to thank my supervisors Sven-Arne Reinemo, Ernst Gunnar Gran and Tor Skeie. All the guidance and discussions provided by them helped me through this thesis. The time and pieces of advice I got from you all is greatly appreciated.

I would also like to thank my sister Stine Tveito, Svenn-Andre Smestad and Hans Petter Taugbøl Kragset for reading my thesis and provide helpful feedback.

Finally, I would like to thank my good friends at Assa. We managed to make Assa a special place at the university for all of us and all the fun that unfolded in this room will be remembered forever.
Chapter 1

Introduction

This chapter contains a short introduction to computer networks and high-performance computing. It will provide a short background for, and the definition of, the research questions of this thesis. Some terminology is introduced and will be explained in more detail in the next chapter. The limitations of the thesis will be discussed and presented. A presentation of research methodologies is given, and our choices are explained. This chapter ends with a brief outline for the rest of the thesis.

1.1 Background and Motivation

The development of modern central processing units (CPUs) has since 1970 followed Moore’s law\(^1\), a law that dictates that the number of transistors in a microprocessor should double approximately every two years. Since 2007 the amd64 (x86_64) architecture has been the dominant CPU architecture in the list of the top 500 supercomputers in the world\(^2\). The x86_64 architecture has over the years become very affordable compared to older architectures. Because of this evolution in processing capacity and the lower cost of developing and building CPUs\(^2\), the number of supercomputers in the world has grown tremendously. Many of these supercomputers are clusters with a vast number of computers interconnected. We normally use the term *compute node* about a single machine inside a cluster. The largest supercomputer today is named Sunway TaihuLight and is located at the National Supercomputing Center in Wuxi, China. The performance of supercomputers is measured in the number of floating-point operations per second (FLOPS) they are able to compute in a Linpack benchmark\(^1\). Sunway TaihuLight can do 93.014 peta\(^2\) flop/s\(^4\). We often describe this

---

\(^1\)Linpack benchmark: Software that tests the computers ability to solve linear equations using a dense random matrix.

\(^2\)Peta: Quadrillion (thousand trillion).
area of computing as high-performance computing (HPC). If we look back ten years on the list of the top 500 supercomputers, we see that the leading system could only do 280.6 teraflop/s. From this, we can conclude that the growth in computing power of HPC systems (on this list) has over the last decade increased more than 100 times\[^2\].

The amount of computing power needed by researchers and enterprises in the world is expected to grow in the future. Already today, scientific problems that need to be solved by this group are varied and complex. For instance, in medicine, protein folding is prominent. In astrophysics, simulation of The Big Bang is a major subject, and simulation of car crashes by car producers. Another area that relies on large amounts of computing power is weather forecasting. The scientific problems in these fields of study are expected to become more complex and will require more computing time to be solved. According to [5], supercomputing is now recognized as the "third pillar" of scientific inquiry. Since more and more researchers are utilizing HPC systems to solve their problems, and existing problems are becoming even more complicated, the demand for HPC resources is growing.

Even though modern CPUs are getting new instruction sets\[^4\], the number of cores, speed, and memory capacity are increasing every year; there are still physical limitations on how powerful one single compute node can be. It is difficult to improve computer performance enough using a single processor. The amount of power such a CPU would use is not feasible\[^6\]. The best solution to gain the needed computing power is to interconnect multiple machines and CPUs. The growth in the number of compute nodes and cores in modern supercomputers puts high pressure on the network infrastructure used to interconnect these nodes. The first time a supercomputer with the InfiniBand (IB) interconnect appeared on the Top500 list was in June 2003. This supercomputer was a system with 128 nodes with single core processors\[^2\]. Today, the fastest computer using the IB interconnect has 241,108 cores.

**Interconnection Networks**

When building a computer cluster, a interconnect is required so that the individual compute nodes can exchange data between processors, or e.g. a shared or distributed memory. The interconnect needs to provide efficient data movement and integrate computational resources as one single system\[^7\]. Interconnecting the compute nodes have become a central part of supercomputers as the number of CPU cores are still increasing. Multiple factors of an interconnection network affect the performance of it.

\[^3\]Tera: Trillion.
\[^4\]Instruction set: A set of instructions/operators that microprocessors support.
• **Network links**: built using copper wires, optic fiber or wireless using radio spectrum.

• **Switches and routers**: connects and aggregates network links together. Switches also connect to end-nodes.

• **Network topology**: how the switches, links, and end-nodes are connected.

• **Routing algorithm**: how routes and/or paths that network packets follow in the network, are calculated.

• **Network protocols**: a set of system rules that are used during communication.

In these distributed environments, factors such as high bandwidth and low latency are the central requirements in the communication equipment. In the most known communication network, the Internet, the TCP/IP stack is used on top of varying underlying network standards, but Ethernet has long been the default standard for local area networks (LANs). Ethernet was in 2005 the dominating standard used for interconnecting supercomputers listed in the top500 list, and it was used extensively in the years before. Due to the high latency of a transmission signal between two nodes in Ethernet, and the failing of keeping pace with the increased needs for higher bandwidth, a new interconnect standard with lower latency, and improved bandwidth was needed. Work on a new standard started already in the late 90s, and in 2000 the first version of the IB standard was released. The InfiniBand Architecture (IBA) has emerged to become one of the most used interconnect standard in HPC. Of the top 500 supercomputers in the world, IB is today used in 35.4% of the systems. Until June 2017, IB was the most-used interconnect family with 37.4% market share of the supercomputers on this list. In this last period, both OmniPath which is an Intel owned interconnect, and 10 Gbit Ethernet has increased with some percentages which caused IB market share to decrease.

We also see that IB recently has started to take marked shares in enterprise computing and storage systems as well. A key highlight of the IB standard is that it has an end-to-end latency that is around six times lower than Ethernet and the IB standard supports remote direct memory access (RDMA). RDMA permits memory access in a remote computer over the network without interrupting the remote host operating system (OS). By directing the signal outside of the OS buffers and processing, and by not using shared buses, it ensures low latency and high bandwidth which is a requirement in these environments.

---

5LAN: A computer network that links devices and adjacent buildings together. Normally with a radius less than 1km.
To support the growth in the number of compute nodes in cluster environments, the IB topologies are becoming bigger and more complex. Building and operating these networks are thus more complicated, and fast detection of problems within the network is becoming more important to ensure that the cluster utilization is optimal. Some of the challenges are to monitor the performance of the network without wasting CPU cycles, filling the network with overhead traffic or disturbing the application data in the network. In this thesis, we will take a deeper look at how to do efficient performance monitoring of an IB cluster. Furthermore, we will provide an evaluation of how well the IB switches handles in-depth monitoring of performance counters using official low-level libraries for querying. We will also look at how querying of these counters is affecting the network and specifically the switches.

1.2 Problem Statement

This thesis will have two goals. The first goal is to design and implement an IB performance metric collection software plugin to the Fabriscale Fabric Management stack, and the second is to explore multiple factors on how metric collection affects the network, and how the network affects the metric collection.

As part of the first goal, we have to look into how to build an efficient metric collection software, and we need to evaluate multiple design solutions in the context of the development process. The questions we need to answer are listed below:

- How to integrate the software with the existing Fabriscale software.
- How to communicate with the existing Fabriscale software efficiently.

The investigation of the metric collection in an IB subnet will answer these research questions:

- It is a likely scenario that an IB network is heavily loaded with traffic. How does this network load affect querying?
- Metric collection involves asking a switch for its counters. How well does an InfiniBand switch handle these queries and how often is it possible to query a switch for its counters? Does the querying pattern affect the result?
• When querying the network for performance counters, we introduce more network load into a network used for running large processes on multiple machines. We need to know how much additional data traffic is injected into the network when collecting metrics, and how this does scale with the number of nodes and ports in the network.

1.3 Research Methods

In 1989, the Association for Computing Machinery (ACM) Education Board published a paper where they defined that the discipline of computer science is divided into three major paradigms[11]. In this paper, the ACM Education Board finds it irrational to say that any of these paradigms are fundamental for computer science, but that they are intricately intertwined. Computer science, as well as applied mathematics and engineering, has a unique blend of theory, abstractions, and design. In the following paragraph we explain these three paradigms in more detail.

• The theory paradigm: is rooted in mathematics. In this paradigm objects to study is defined. Initially, a set of hypotheses and theorems are built together with their possible outcomes. These theorems are processed to be proved or disproved. The last step is to interpret the result.

• The abstractions paradigm: is rooted in experimental research and consists of four stages. The researcher forms a hypothesis, builds a model, and makes some predictions on what to expect before the experiment is designed. The last stage is data collection.

• The design paradigm: is rooted in engineering. In this model, a system or device is built to solve a problem. This process starts with stating requirements and specifications. Then the system or device is designed and implemented. Tests are run, and if failing the above steps can be repeated.

As previously mentioned, in this thesis we have two goals; we will design and implement a robust metric collection software to integrate with the software stack of Fabriscale Technologies AS and we will investigate how metric collection affects network equipment that is queried. It would have been very difficult to use the theory paradigm in this case, as there is no mathematical way of calculating the performance of IB switches. Too many factors are unknown. Of these three paradigms, we found that using a combination of both the abstraction paradigm and the design paradigm was the most fitting way of tearing into the problems of this thesis. The design paradigm is perfect for the first group of research questions in this thesis, and the abstractions paradigm can be to the investigation needed to answer the
second group of research questions. First, we will define a set of requirements and specifications for the metric collection software and implement it. We will design multiple sets of experiments and then collect data on how well switches handles querying and how the software implemented as the first goal affects the network.

1.4 Thesis Outline

The rest of the thesis is organized as follows:

Chapter 2 provides a detailed overview of the IBA and its basic building blocks such as switching and quality of service. The chapter presents the management model and some key elements used for administrating an IB network. The Fabriscale Fabric Manager is introduced, and some of its specifications revealed. A presentation of relevant software libraries is given.

Chapter 3 presents the design and implementation of a performance metric collector plugin for the Fabriscale Fabric Manager (FFM) built in this thesis. It discusses the specific implementation details and explains why it was done this way.

Chapter 4 focuses on the experiments performed in this thesis. It explains the experiments we did and gives a reasoning for the chosen experiments. Some limitations and issues are presented.

Chapter 5 presents a discussion of the results from the previous chapter.

Finally, Chapter 6 gives the conclusion that we drew from the research in this thesis.

1.5 Summary

This chapter first gave a brief introduction to HPC and supercomputers. Here, it presented some historical aspects of the top 500 most powerful supercomputers. One of the reasons for the growth in these systems was discussed, and it touched into how to make these high-performance computers more powerful using interconnect networks.

Following the first section was an section on interconnect networks and specifically the standard IB. This section also mentioned requirements for a modern interconnect, as well as addressed multiple factors that affect the
performance of the interconnect.

The problem statement of this thesis was given, where it was explained that this thesis is aiming towards building an IB performance metric collector. It also explained that we are interested in seeing how performance metric collection is affecting the network, and how well switches handles in-depth querying of its counters.

Next up we gave the research methods used in this thesis. For the first goal, designing and implementing the plugin, we will use the design paradigm by the ACM Education Board; this paradigm focuses on the implementation and testing of the research problem. For the next goal, running experiments, we will use the abstractions paradigm also by the ACM Education Board; this paradigm focuses on building an hypothesis and make some predictions before data is collected and analyzed.

At the end of this chapter, we gave the outline for the rest of the thesis.
Chapter 2

Technical Background

In this chapter, we present a deeper introduction to the InfiniBand Architecture. This chapter also describes the layered network stack, and it will focus on the management model. It explains some of the introduced concepts and terminologies from the last chapter in more detail and many new concepts are introduced. Following, it presents the software libraries and tools that we made use of in this thesis, and the Fabriscale software stack is given.

2.1 InfiniBand Overview

The IBA\[12\] is defined by the InfiniBand Trade Association (IBTA), a group that today consists of more than 220 companies, founded in 1999[13]. IBTA has the responsibility of maintaining and furthering the IB specification. The sole purpose behind the design of the IBA was to build a new open industry standard interconnect technology that reduced the overhead found in the existing industry-standard I/O\(^1\) systems which used shared buses. Overhead issues in those systems were related to copying, buffering, checksumming, and interrupts. The existing interconnects had failed to keep pace with the computer evolution and the increased needs for higher capacity and lower latency.

An IB network is divided into smaller sub-networks which are interconnected by routers. Within subnets, computing nodes are connected using switches. Switches also connect to each other to scale the subnet. This topology is often referred to as a switched fabric or just fabric. The computing nodes use adapters called host channel adapters (HCAs) to connect to switches. These adapters are used to connect processors and the I/O devices of the computing nodes to the network. The HCA is designed to allow direct

\(^1\)I/O: Input/Output: Operation that transfer data from and to a computer.
application level communication and avoiding kernel\textsuperscript{2} operations to achieve lower latency operation. Kernel operation is expensive in terms of computing and requires the CPU to raise its privilege level. Data must be copied multiple times which leads to an increase in the computing time. Figure 2.1 presents a logical view of an IB subnet. In this figure, we can see multiple switches connected to each other, and we see storage systems as well as computing nodes. The illustration refers to computing node as processor node. The figure also shows an entity called TCA. This stands for target channel adapter, and these adapters are used on nodes not operating as computing nodes, like for instance, storage clusters. These adapters differ little from HCAs. Some interfaces are required on HCAs, but not on target channel adapters (TCAs).

The architecture is independent of host operating system and processor platform. A required entity called subnet manager (SM) is responsible for configuration and operation of the devices in the subnet. We will give a more extensive presentation of the SM in Section 2.1.5.

The IBA is like many other communication systems an abstract model which can be seen as a layered stack inspired by the Open Systems Interconnect (OSI) model\cite{14}. The OSI model was published in 1984, and was a result of a collaboration between the International Organization for Standardization (ISO) and the Telecommunications Standardization Sector International Telecommunication Union, or in short ITU-T. The OSI model is a basic abstract layered model used for networking. In Figure 2.2 we present a view of the OSI model and its layers, together with a descriptive text for each layer. On the right side of this figure, the IBA layered stack is presented in conjunction with some of its responsibilities (Figure 2.3). Note that on this figure we have removed the network layer.

In the next section, we will give the core layers of the IBA and go into detail about their individual responsibilities. This section will explain why there is no network layer shown in Figure 2.3

2.1.1 Physical Layer

At the physical layer (PHY), it is specified how bits are placed on wires and fiber optic cables. The PHY provides an interface between the link layer packets and serial bits sent over a physical medium. When data is handed over from the link layer to the PHY, the PHY adds start and end delimiters to the packet. These delimiters are used for clarification of where a packet starts and ends. If there is no immediate packet from the link layer that needs to be sent, the PHY adds an idle signal to fill the channel. By adding this idle signal, one makes sure that no random noise that appears on the

\textsuperscript{2}Kernel: A core computer program inside the operating system with a complete control over all hardware and software.
channel will be interpreted as data on the receiving end. An illustration of a complete IBA PHY frame is shown in Figure 2.4. This illustration shows how the data from the link layer is encapsulated inside start and end delimiters, and it shows the idle signal blocks at the end of the packet.

To help reduce transmission errors due to overlap and distortion of the transmission signal, the data packets from the link layer has to be modified. We call this modification *encoding*. Encoding the signal helps the receiver interpret the signal without making mistakes that would cause a transmission error. After the data packets have been encoded, we call the new bit stream line-code.

Today, two different encoding schemes are found in the IBA. The oldest PHYs standards use 8b/10b encoding, and in the more recent PHYs, the 64b/66b encoding scheme is applied. In 8b/10b encoding, it takes 10 bits to send 8 bits of data from the link layer, and with 64b/66b it takes 66 bits to send 64 bits of data. The introduction of 64b/66b encoding to the IB PHY lowered the overhead from 20% (using 8b/10b) to 3.125%. We present the different encoding schemes that are found in the IBA in Table 2.1. In the calculation above, overhead is defined as the extra bits that are needed to send the data bits from layer two.

At the IBA PHY, one copper port requires about 0.25 watts to transmit a signal, in contrast, the Gigabit Ethernet (GbE) PHY requires around 2
watts per copper port. GbE over copper is designed for LANs and must reach 100 meters. To achieve this, the ports have to send a high powered signal. The IBA PHY is designed for data center usage and does not need to reach that far and can operate with less power.

### Link Width and Link Speed

In addition to the encoding we discussed above, the PHY is also responsible for link training, maintaining links, and receive error detection logic. The IBA has support for three different link widths, 1x, 4x, and 12x. Each link width needs one pair of wires for each direction to support full-duplex\(^3\) communication. Thus a 4x link needs four different wire pairs in each direction, eight in total. If optics are used on a link, only one fiber is necessary for each direction for full-duplex communication.

Together with the different link width, there are multiple options in link

\(^3\)Full-duplex: Transmission of data in two directions simultaneously
speeds that can be used to establish a link. As of today, the IBA supports five different link speeds, all of which can be used together with the mentioned link widths. Single data rate (SDR), double data rate (DDR), quad data rate (QDR), fourteen data rate (FDR-10/FDR-14), and enhanced data rate (EDR). The link speed with the highest available data rate today is EDR. EDR has a signaling rate of 25 Gbit/s and gives a theoretical throughput of 24.24 Gbit/s on 1x link width. A 12x EDR link would offer a signaling rate of 300 Gbit/s. As the demand for real-time data analysis and offloading of tasks from the CPUs to other parts of the network is increasing, the demand for faster interconnects is growing[16]. The IBTA is currently working on the next link speed called high data rate (HDR), which they plan to release later this year (2017). This new specification aims for a signaling rate of 50 Gbit/s. We present some of the link characteristics for the different links supported by the IBA in Table 2.1.

<table>
<thead>
<tr>
<th></th>
<th>SDR</th>
<th>DDR</th>
<th>QDR</th>
<th>FDR(14)</th>
<th>EDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signaling Rate (Gbps)</td>
<td>2.5</td>
<td>5</td>
<td>10</td>
<td>14.0625</td>
<td>25</td>
</tr>
<tr>
<td>Theoretical Throughput (Gbps)</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>13.64</td>
<td>24.24</td>
</tr>
<tr>
<td>Speeds for 4x links (Gbps)</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>54.54</td>
<td>96.97</td>
</tr>
<tr>
<td>PHY encoding</td>
<td>8b/10b$^4$</td>
<td>8b/10b</td>
<td>8b/10b</td>
<td>64b/66b$^5$</td>
<td>64b/66b</td>
</tr>
</tbody>
</table>

Table 2.1: IB Link Characteristics[12].

2.1.2 Link Layer

Many of the IBA core functions are located at the link layer. The link layer handles all point-to-point link operations. Services provided by the link layer are; addressing, buffering, flow control, error detection, quality of service, and switching of packets inside a subnet. At the link layer, we find two kinds of packet types. Link layer management packets and regular data packets. The link layer management packets are used to establish and maintain link operations. These packets are not subject to flow control but are part of the operation of it. For all other purposes, a data packet is used. The data packet begins with a local route header (LRH), which can be seen in Figure 2.6. The maximum allowed transmission unit (MTU) at this layer is 4096 bytes. Including all the upper layer headers, a packet at the link layer can be maximum 4222 bytes. The link layer header (LRH) takes up 8 bytes. A view of a complete packet sent from the link layer down to the PHY can be seen in Figure 2.5. This illustration also includes the additional headers added by the PHY and the upper layers of the IBA.

---

$^4$8b/10: 8-bit words are encoded to 10-bit symbols before transmission.

$^5$64/66: 64-bit data blocks is encoded to 66-bit blocks before transmission.
Addressing and Switching

Within a subnet, switching is handled at the link layer. All devices within the subnet have a 16-bit address called local identifier (LID) assigned to it by the subnet manager. Inside a subnet, the LID is used for addressing between IB devices. When a host has some data destined for another host, the source LID and destination LID fields in the LRH have to be filled accordingly. When packets arrive on an IB switch, the switch is determining where to forward the packet based on the destination LID header field. The LID is not a persistent address and can change, for instance after a power cycle of the device. Therefore, initially, before paths have been calculated, and the SM has fully configured devices with forwarding tables, addressing is done using a 64-bit long address called the global unique identifier (GUID). This address is assigned to the device when fabricated by the manufacturer.

<table>
<thead>
<tr>
<th>bits/bytes</th>
<th>31-24</th>
<th>23-16</th>
<th>15-8</th>
<th>7-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>VL</td>
<td>LVer</td>
<td>SL</td>
<td>Rev2</td>
</tr>
<tr>
<td>4-7</td>
<td>Reserve 5</td>
<td>Packet Length (11 bits)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.5: IBA data packet seen at the link layer[12].

Figure 2.6: IBA LRH[12].
Flow Control

When the IBTA designed the IBA, they had one critical requirement for the link layer. The network had to be lossless. Packet loss is causing retransmissions which are a source of increased latency and increased bandwidth usage. To prevent packet loss, IBA describes flow control at the link level using an absolute credit based[17] scheme for each link and direction. In this scheme, the transmitter has to allocate credits from the receiver before sending. IBA offers a credit limit which describes the total amount of data that the transmitter is authorized to send over a link. This limit is never higher than what the receiver has in free buffers. The transmitter is informed about this value when the link is initialized, and updates are sent periodically to make sure the value is synchronized between the transmitter and the receiver. Having this absolute limit which ensures that the receiver always has enough space in its buffers, is making sure that no packets are dropped due to the receiver getting more data than it can handle. All information exchanged by this protocol is transmitted using link layer management packets.

Quality of Service

Another important feature of the link layer is the quality of service (QoS) system. To achieve QoS in IB networks, each physical link in the network has multiple virtual lanes (VLs), where each VL potentially have a differentiated transmit priority. VLs are logically separated channels on a link, adopting their own set of dedicated transmit and receive buffers (in short: tx/rx) on each port. VLs are also implemented with individual flow control mechanisms. The IB specification allows for a total of 16 different VLs. VL0-14 are used to carry application traffic, and VL15 are used exclusively for subnet management traffic and has no flow control. The management VL has the highest priority of all the lanes, but packet loss can occur since VL15 does not implement flow control.

The IBA provides a four-bit header field in the LRH for marking packets with their QoS level. In the IBA specification, this is frequently described as the service level (SL), and this is also the name of the field in the LRH. An illustration presenting the LRH is shown in Figure 2.6. The SL header field may be arbitrarily used to indicate a class of service. The IBA does not define the mapping between the SL and forwarding behavior. It is up to the network administrator to construct policies for this.

In addition to the SL field, the LRH also has a VL field that indicates which VL the packet was transmitted on. When a switch receives a packet, the VL field is checked, and the packet is placed on the corresponding VLs receive buffer. All switches in the fabric have an SLtoVL mapping table managed
by the SM. By looking up in this table, a switch will know which VL to forward the packet on. The switch uses the SL field, the port it was received on, and the port it should be forwarded on to determine the VL to transmit the packet on.

The process a switch uses on an output port to find which VL to transmit from is called **VL arbitration**. IBA has specified a dual priority weighted round robin scheme for this. As mentioned above, each VL has a different transmitting priority. Packets from the high priority VLs are always transmitted before packets on lower priority ones. The VL arbitration is specified using a VL arbitration table on each IB port where each list entry contains a VL number and a *weighting value*. The weighting value specifies the number of 64-byte units that can be sent from that VL before moving to the next VL.

### 2.1.3 Network Layer

The IB network layer handles routing of packets between different IB subnets. It uses IPv6\(^6\) as the addressing scheme. Addresses are 128 bits long and are stored in the global route header (GRH) of network packets. It should be noted that the network layer is not required to operate within one subnet, which is the likely scenario for an IB network. When the network layer is not in use, the 40-byte long network layer header can be dropped. This is one feature of the IBA that ensures that overhead traffic is kept as small as possible. If multiple IB subnets are to be connected, a router that connects to all of the subnets is needed. The router is routing packets between subnets based on source and destination addressing in the GRH.

### 2.1.4 Transport Layer

In IB networks the transport layer is responsible for in-order packet delivery, partitioning, channel multiplexing and transport services. The IBA uses a transport header on all packets which contains the information required by the end node to handle the incoming packets and deliver it in-order to the correct application. All IB transport packets have a 12-byte base transport header (BTH). It contains multiple fields used by the IBA transport layer, field such as sequence number and partition. The destination application is also addressed through the BTH.

Queue Pair

Applications running on a computing node are communicating with the transport layer using work queues for receiving and transmitting operations. These queues are referred to as the queue pair (QP), and they can be seen as the IB consumer and producer interface to the fabric. In general, the transmit queue holds instructions that cause the hardware to transfer data between the requester’s memory, and the memory of another node. The receive queue is containing information about where in memory to store received data.

To locate the correct application a packet is destined for, a destination QP header field in the BTH is used. This header field is inspected when a packet is received at the transport layer. The transport layer moves the packet into the corresponding receive QP. When creating QPs, they must be associated with one given transport service defined by the IBA. The size, layout and stored information of the QP vary depending on what service it is associated with.

Transport services

The IBA supports these five different transport services: unreliable datagram (UD), reliable datagram (RD), unreliable connected (UC), reliable connected (RC), and Raw Datagram. These different types of transport protocols provide various services for data reliability. In Table 2.2 we present the differences in these transport services using three attributes. An explanation of these follows.

- **Connection Oriented** versus **Datagram** - Connection Oriented service works with a specific set of QPs. Datagram service allows a single QP to be used to send to any other QP on any node.

- **Reliable** versus **Unreliable** - Reliable service provides a guaranteed delivery of each packet, data in-order and without errors by using acknowledgements\(^7\). Unreliable service does not guarantee that all data is delivered. In some cases, it may deliver packets out-of-order if network configuration changes.

- **IBA transport** versus **Other transport** - The IBA transport service is specifying channel based and memory based operation. The IBA also supports using channel adapters in RAW mode which allows raw packets to be sent. RAW mode is useful for supporting legacy networks and protocol stacks or running custom protocols.

\(^7\)Acknowledgment: Signal passed to inform that some packet was received without error.
<table>
<thead>
<tr>
<th>Service Type</th>
<th>Connection Oriented</th>
<th>Acknowledged</th>
<th>Transport</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliable Connection</td>
<td>yes</td>
<td>yes</td>
<td>IBA</td>
</tr>
<tr>
<td>Unreliable Connection</td>
<td>yes</td>
<td>no</td>
<td>IBA</td>
</tr>
<tr>
<td>Reliable Datagram</td>
<td>no</td>
<td>yes</td>
<td>IBA</td>
</tr>
<tr>
<td>Unreliable Datagram</td>
<td>no</td>
<td>no</td>
<td>IBA</td>
</tr>
<tr>
<td>RAW Datagram</td>
<td>no</td>
<td>no</td>
<td>Raw</td>
</tr>
</tbody>
</table>

Table 2.2: IBA Service Types[12].

Transport Service Scalability

When using the RC transport service, one QP is associated with one specific remote QP. Since RC is a connected service, each consumer has to establish a communication channel between all remote consumers it wishes to communicate with. When using RC, the QP is keeping track of the reliability context for each communication channel. The reliability context is the various state information needed to provide reliable service, such as sequence numbers. In a fully connected fabric with $N$ multi-processor nodes where each node has $P$ processor cores, $(N - 1) \cdot P^2$ QPs are needed to keep the context for all nodes. In larger HPC systems, such as the "Ranger" at Texas Advanced Computing Center (TACC) in the US, with 60,000 cores and 4,000 IB ports in the fabric, the memory usage of RC can reach hundreds of megabytes per process[18]. This level of memory usage illustrates how bad RC scales with the increase in both the number of CPUs and cores in modern HPC systems. To deal with this, the developers behind the IBA moved the reliability context out of the QP and established a separate entity called end-to-end context (EE context) - this is where RD differentiates from RC. In RD, the separate EE context solution is using $P$ QPs plus $N$ EE contexts per node to hold state information. When using RD and the external EE context as discussed above, an additional header is added after the BTH - the extended transport header (ETH). An additional header was needed to identify the EE context that the QP uses to detect missing packets.

As a result of growth in the number of cores in HPC systems, the IBTA released an annex in 2008 to reduce memory usage in larger systems. Annex A14[19] to the IBA describes the extended reliable connected (XRC) transport service. XRC was a new approach that reduced the required QPs needed for full connectivity. As discussed above, RC requires a connection to each process in the cluster for full connectivity. By using XRC, this requirement is lowered as only one connection per destination node is required. With this new annex, the memory usage is reduced by a factor of $P$. This is a significant reduction in memory used for QPs in larger systems.
Remote Direct Memory Access

As part of both the transmitting and receiving instructions supported we find remote direct memory access (RDMA). IBA supports both reading and writing operations to another application’s memory over the fabric. IBA RDMA is zero-copy, which means that the reads and writes can be done without copying the data multiple times at either host before it can be transmitted or handled when received. The QP associated with the communication pair holds information about which virtual addresses that the remote host can read from and which it can write to at the local host. The RDMA service supports reads and writes of up to $2^{31}$ bytes. When IBA RDMA is used, the CPU of the remote host is not interrupted during the data transfer. RDMA thus ensures low latency data transfer over the network and helps to minimize the CPU resources used on computing nodes for data copying.

IBA Layering Summary

In this section, we have presented the four core layers of the IBA and their responsibilities and discussed the services they offer. In Figure 2.7, all of these layers and its headers are shown. The terminologies in this figure have been introduced and explained. This figure is presenting a good summary of the different layers and their tasks in the IBA.

Figure 2.7: Overview of the IBA layers[13].
2.1.5 Management Model

In the management protocols of the IBA, management control messages are transmitted using data packets called management datagrams (MADs). These packets are transmitted using the UD transport service as we discussed in Section 2.1.4. MADs are the basic elements of the management messaging in the IBA. The IBA management is classified into multiple classes, and for each class there is a specialized use and behavior. Common for all classes are the MAD base header. Sub-class headers are placed in the data field of the base MAD header. In Figure 2.8 the base MAD packet format is shown. When a MAD is constructed, it must be exactly 256 bytes long. MADs must be padded\(^8\) with zeros if a message is not taking up all 256 bytes. Each management class defines extra header fields that are added to the data part of the base MAD. In this thesis, we plan to use MADs to collect performance metrics from the fabric.

The IBA has organized the management model using abstract functional entities referred to as managers, agents, and interfaces. These entities have the following responsibilities and functions.

<table>
<thead>
<tr>
<th>bytes</th>
<th>bits 31-24</th>
<th>bits 23-16</th>
<th>bits 15-8</th>
<th>bits 7-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>BaseVersion</td>
<td>MgmtClass</td>
<td>ClassVersion</td>
<td>R</td>
</tr>
<tr>
<td>12</td>
<td>Status</td>
<td>ClassSpecific</td>
<td>TransactionID</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>AttributeID</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>AttributeModifier</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>252</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.8: Base MAD[12].

- **Managers** are entities that have control over the fabric elements, or they provide methods of gathering information from fabric elements. They may reside anywhere in the fabric as long as they have access to the fabric.

- **Agents** are entities that are present in HCAs, switches, and routers. Agents are responsible for the processing of management messages arriving at ports of its host.

- **Interfaces** represent an abstract target of which messages may be sent and specifies where these messages will be forwarded for processing. The IBA defines two such interfaces, the subnet manager interface (SMI) and general service interface (GSI).

---

\(^8\)Padding: Bits or characters that fill up unused portions of a data structure.
So far in this thesis, we have referred to an entity named subnet manager (SM) multiple times. Finally, we have come to the point where we will present more details about the SM, and the work it is responsible for inside an IB network.

In Section 2.1.5 we gave an introduction to the abstract entities that are defined in the IBA management model. One of these was the manager. The SM is the concrete object and is an essential part of the IBA. The SM is responsible for initializing, configuring, and managing all routers, switches, and channel adapters on the network. In a subnet, there must be at least one SM present. The SM continuously probes the fabric to get information about the connected IB devices and to get a view of the fabric topology. It is the job of the SM to configure channel adapters and switches in the fabric with a LID and subnet prefix. The SM also maintains a LID to GUID mapping table.

The IBA supports the notion of multiple SMs present on a subnet, as long as just one is active at a time. Standby SMs that are present in the fabric keeps a copy of the state of the master SM, and should be ready to become the master SM at any time. Standby SMs are verifying that they have a valid copy of the master state and that the master SM still is active in intervals. The interval is not specified by the IBA specification, but must be configured by the administrator. A short interval is recommended by the IBA. The network administrator must also set a priority variable on the different SMs. If a master SM fails, the standby SM with the highest priority will take over the responsibilities to ensure that the subnet does not go down together with the SM. The SM communicates with all IB devices using subnet management packets (SMPs) which are a specialized class of MAD and is transmitted using QP0 exclusively over VL15.

One of the core responsibilities of the SM is to calculate all paths between all pairs of source and destination nodes. The SM should distribute the calculated forwarding tables and load them in switches. Multiple paths between end nodes may be calculated and loaded onto switches. These can be used for redundancy or load sharing. The IB specification does not force any particular algorithm. The choice is left to the network administrator. If an error or fault occurs in the fabric, it is the SMs job to recalculate paths and reconfigure forwarding tables on all devices so that normal operation of the subnet can continue.
**Subnet Manager Agent**

Another entity inside the conceptual management model is the subnet manager agent (SMA). The SMA is the entity inside an IB device that is responsible for the communication with the SM.

The SMA can be described as a daemon\(^9\) and must be present on channel adapters, switches, and routers. The SM communicates with the SMA using a defined interface called SMI. The SMI can be addressed using both LID routed packets or direct routed packets. Direct routed packets are SMPs that include a path vector which specifies the ports the packet should be forwarded on throughout the network. These packets are for instance used by the SM before switches and routers are configured with forwarding tables. The SMA is an essential part of a node during the setup process. It will receive configuration parameters from the SM and should apply these to the node. The SMA is also a vital part of the fabric discovery process. It is responsible for transmitting necessary information about the device, such as addresses and capabilities. More extensive information about the device can be gathered by querying the baseboard management agent. We will present this agent shortly.

Another responsibility of the SMA is to transmit traps\(^10\) when certain events occur on the device. Traps are sent to the master SM using SMP over UD. The IBA defines five types of traps; *Fatal, urgent, security, subnet management,* and *informational.*

One example of when a trap is sent is when a switch sees a link down event, or when new nodes are connected and discovered. A link down event is classified as an urgent trap, and newly discovered nodes are classified as informal. Another scenario that would cause a trap is when a path is no longer valid. Since transport of traps is unreliable, the SM can not solely depend on information it receives from traps, but getting traps will, for instance, speed up processing of a topology change.

**General Services Management**

In addition to the SMA, IB channel adapters, switches, and routers contain entities called general service agents (GSAs) which can be communicated with using general management packets (GMPs) using the GSI. These packets are a specialized class of MADs just like SMPs, but GMP uses QP1 and can not be transmitted using VL15. Since these packets must be transmitted over any of the data VLs, they are subject to flow control as

---

\(^9\) Daemon: Computer program that runs as a background process, rather than being under the direct control of an interactive user.

\(^10\) Trap: Upstream messages from nodes in the network to the SM.
described in section 2.1.2.

The IBA does specify some GSAs that must be present on devices, and it is also possible for manufacturers to define own agents using this interface. Some of the mandatory agents are performance management agent (PMA) and baseboard management agent (BMA).

**Baseboard Management Agent**

In addition to the GSA, another agent found on IB devices is the BMA. When requested, it provides an answer with extensive IB specification information about the device it resides in. The BMA provides an in-band\textsuperscript{11} low-level management of the chassis, it extends the SMA and provides low-level information about the device. For instance, it can provide status about the light-emitting diodes (LEDs) present on a switch. Commands received on the BMA is handed over to the module management entity (MME) and is processed there. The MME provides a response back to the BMA.

**Performance Management Agent**

The PMA is mandatory for all IB devices. It provides mechanisms to retrieve performance and error statistics, and capability information from the IB device it resides. It also provides functions to set or reset some of the performance counters found on the device. To communicate with the PMA over the IB fabric, one must use performance management (PerfMgt) packets. These packets are a sub-class of GMP. An illustration of the PerfMgt packet is given in Figure 2.9. In the data field of this packet, attribute data is mapped bit for bit from the format described in the IBA specification. These MAD packets are a significant factor in the IB metric collection software we describe in chapter 3. We will use these packets to ask for switch capabilities and to retrieve metrics from switches connected in the fabric where our software is deployed.

**Error and Performance Counters**

The IBA specifies a set of counters that are required on all IB ports. Table 2.3 lists all of the mandatory error and performance counters together with an explanatory text for each counter.

These counters provide basic performance and exception statistics for IB ports. When an SM initializes the fabric, these counters are set to zero

\textsuperscript{11}In-band: Communication over the same link as regular application data.
on all devices. When they reach their maximum value, they are defined to stop and not overflow. Writing zero to the counter will reset it, writing any other value has undefined behavior. Some of the available counters are LinkDownedCounter, PortRcvErrors, PortXmitDiscards, PortRcvData, PortXmitPkts, PortRcvPkts, and PortXmitWait. Many of these counters have descriptive names, for example; PortRcvErrors which contains the number of received packets that had an error. Usually, this is caused by cyclic redundancy check (CRC)\textsuperscript{12} fail due to a bit error inside the packet. On modern hardware, most of these counters are 64-bit long, but on older hardware, they were limited to 32-bits. Using a 32-bit counter with the most recent high bandwidth link-speeds such as EDR, makes the counter hit max almost instantly. With a 4x EDR link that operates at 100 Gbit/s and a 32-bit PortXmitData counter which can hold a maximum value of 4,294,967,295 bytes, the counter would hit max and stop after 1.36 seconds. By using a 64-bit counter at the same data rate, it would hit max after approximately 188 years.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PortXmitData</td>
<td>Number of data octets divided by four that has been transmitted on this port.</td>
<td>64</td>
</tr>
<tr>
<td>PortRevData</td>
<td>Number of data octets divided by four that has been received on this port.</td>
<td>64</td>
</tr>
<tr>
<td>PortXmitPkts</td>
<td>Total number of packets transmitted on this port.</td>
<td>64</td>
</tr>
<tr>
<td>PortRecvPkts</td>
<td>Total number of packets received on this port.</td>
<td>64</td>
</tr>
<tr>
<td>PortUnicastXmitPkts</td>
<td>Total number of unicast packets transmitted from the port.</td>
<td>64</td>
</tr>
<tr>
<td>PortUnicastRecvPkts</td>
<td>Total number of unicast packets received on the port.</td>
<td>64</td>
</tr>
</tbody>
</table>

\textsuperscript{12}CRC: Code used in communication networks to detect transmission errors.
<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PortMultiCastXmitPkts</td>
<td>Total number of multicast packets transmitted from the port.</td>
<td>64</td>
</tr>
<tr>
<td>PortMultiCastRcvPkts</td>
<td>Total number of multicast packets received on the port.</td>
<td>64</td>
</tr>
<tr>
<td>PortXmitWait</td>
<td>The number of ticks during which the port had data to transmit but no data was sent during the entire tick either because of insufficient credits or because of lack of arbitration.</td>
<td>16</td>
</tr>
<tr>
<td>SymbolErrorCounter</td>
<td>Total number of minor link errors detected on one or more physical lanes.</td>
<td>8</td>
</tr>
<tr>
<td>LinkErrorRecovery</td>
<td>Total number of times the Port Training state machine has successfully completed the link error recovery process.</td>
<td>8</td>
</tr>
<tr>
<td>LinkDownedCounter</td>
<td>Total number of times the Port Training state machine has failed the link error recovery process and downed the link.</td>
<td>8</td>
</tr>
<tr>
<td>PortRcvErrors</td>
<td>Total number of packets containing an error that were received on the port. (CRC, bad length, bad VL, and many other incidents).</td>
<td>16</td>
</tr>
<tr>
<td>PortRcvErrorsPortRcvRemotePhysical</td>
<td>Total number of packets marked with the EBP delimiter received on the port.</td>
<td>16</td>
</tr>
<tr>
<td>PortRcvSwitchRelayErrors</td>
<td>Total number of packets received on the port that were discarded because they could not be forwarded by the switch relay.</td>
<td>16</td>
</tr>
<tr>
<td>PortXmitDiscards</td>
<td>Total number of outbound packets discarded by the port because the port is down or congested.</td>
<td>16</td>
</tr>
<tr>
<td>PortXmitConstraintError</td>
<td>Total number of outbound packets not transmitted from the switch due to a constraint error.</td>
<td>8</td>
</tr>
<tr>
<td>PortRcvConstraintError</td>
<td>Total number of inbound packets that was discarded due to a constraint error.</td>
<td>8</td>
</tr>
<tr>
<td>LocalLinkIntegrityErrors</td>
<td>The number of times that the count of local physical errors exceeded the threshold specified by LocalPhyErrors.</td>
<td>4</td>
</tr>
<tr>
<td>ExcessiveBufferOverrunErrors</td>
<td>The number of times that OverrunErrors consecutive flow control update periods occurred, each having at least one overrun error.</td>
<td>4</td>
</tr>
<tr>
<td>VL15Dropped</td>
<td>Number of incoming VL15 packets dropped due to resource limitations. (e.g., lack of buffers) in the port.</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 2.3: IBA performance counters[12].

25
2.2 Monitoring the Fabric

Since IB first was introduced in the HPC world, we have seen that many players have adopted this interconnect. The size of subnets has expanded, and bandwidth has increased together with the performance of the supercomputers. The need for an easy to use low-overhead monitoring tool for fabric management is crucial to be able to understand the behavior and predicting possible problems. Monitoring of network equipment must not disrupt application traffic in the subnet, nor steal computing time from applications. For the monitoring tools, it is important that they in a user-friendly way can show the network administrator how well the network is performing and which faults are occurring with little or no delay. It is essential to know when congestion emerges in the fabric. These tools should also help the network administrator plan for future expansion and upgrading of switches and links.

2.2.1 Current Monitoring Tools

OpenFabrics Enterprise Distribution (OFED) is an open-source stack of software drivers, kernel code, middleware\textsuperscript{13}, and user-level interfaces that support the IBA. OFED includes various diagnostics and performance tools that can be used in a terminal by the network administrator. These tools are quite old, and their user interface has failed to evolve with the network administrators expectations. For performance monitoring, the OFED has multiple specialized tools, each with a narrow administrative view of the fabric. Little context is given to the administrator thus making it hard to get a complete and satisfying view of the subnet and its performance without spending a lot of time.

Other tools, both open and closed source are already proposed, developed, and currently in use. Popular tools like Ganglia\textsuperscript{20} and Nagios\textsuperscript{21} are both open-source and have support for IB using plugins. These tools offer much of the same information in total, and for the IB level, they are based on the same binaries to collect the performance counters. Ganglia and Nagios, with their respective plugins, are using binaries from OFED to gather data from IB devices. OFED offers various utilities to read information from IB devices. One of the binaries that these plugins use is perfquery. When run, perfquery will report back the PortXmtData, PortRecvData, and PortXmtWait among other things. The tools that OFED offers and that Ganglia and Nagios use are restricted to these counters, and thus these tools have no knowledge of links or the topology of the fabric.

To get further information, these tools are daemon based which means that each monitored device in the IB cluster must run a software daemon on every

\textsuperscript{13}Middleware: Software that provides services beyond native OS support.
monitored node in the fabric to collect data about the device. The data gathered by the daemon is transmitted to a central daemon and stored in a database. Separate daemons executing on the compute nodes are stealing valuable computing time from research applications.

When the daemons transmit data, they also generate traffic on the network which is transmitted together with the application traffic. The developers of these tools have tried to limit the overhead caused by not having the daemon sample data constantly, but rather in intervals. Longer time intervals between each sampling reduce overhead, but it also reduces the liveness of the monitoring tool. Such monitoring solutions also have another problem: the data collecting daemons cannot run on switches and routers where the network administrator are not allowed to launch processes.

Another available tool is FabricIT[22], an IB management solution developed by Mellanox. A performance monitoring software, among many other features, is built into their SM called FabricIT. This software is not based on a host agent such as Ganglia or Nagios, and the overhead issues from those tools are not present in this solution. FabricIT scales up to 648 nodes. One issue with this package is that it does not offer long-term data storage of performance counters. Another drawback is the fact that FabricIT is beginning to age, as it was released in 2010. Its web-based graphical user interface (GUI) has not evolved over the years. For instance, the web pages are not dynamically loaded. The biggest issue with FabricIT is that it is Mellanox proprietary, and it will only work with switches made by Mellanox.

In 2011, Nishanth Dandapantuhula wrote a thesis at the Ohio State University (OSU) called InfiniBand Network Analysis and Monitoring using OpenSM[23]. OpenSM[24] is an open source implementation of an SM developed by OFED. He looked into how to build a low overhead monitoring tool for IB clusters that is capable of depicting the communication matrix of target applications and the link usage of various links in the IB network. He proposed a system with two distinct modules, the InfiniBand Network Querying Service (INQS) and the Web-based Visualization Interface (WVI). The INQS uses MADs to collect data from the performance counters of IB devices in the fabric and stores them in a MySQL database. The WVI presents the data to users using HighCharts JS[25]. The monitoring software proposed by Nishanth Dandapantuhula has a static old-style web-based GUI and has no alert functions. To be able to depict the communication matrix of applications, the proposed software is integrated with message passing interface (MPI) over InfiniBand (MVAPICH), which is an implementation of MPI for multiple high-performance network stacks developed at the OSU[26].

Little research has been done on how low-level monitoring of an InfiniBand network using MADs influence application traffic or on how well switches handle repeated in-depth querying of its performance counters. In this
thesis, we will investigate these questions along building a metric collection software, which will be part of a modern monitoring software suite developed by Fabriscale Technologies.

2.3 Fabriscale Technologies

As part of this thesis, we aim to build an IB metric collection plugin to the Fabriscale Fabric Manager (FFM). The FFM is developed by Fabriscale Technologies, a company with focus on developing a new generation of fabric management tools that ensure more efficient and reliable operation of data centers and high-performance clusters based on IB and Converged Ethernet\textsuperscript{14}. Fabriscale offers unique services such as a modern web-based GUI that is ensuring easy administration of the subnet. The web-based GUI also has a monitoring view where faults, congestion, and other fabric health-related variables are shown. The FFM has a representational state transfer (REST)[27] application programming interface (API) exposed that the network administrator can access using Hypertext Transfer Protocol (HTTP)[28] requests. This API facilitate existing applications to be integrated into the FFM or so that the network administrators can develop new applications around the fabric manager if they desire.

As part of the FFM, Fabriscale is developing the Fabriscale subnet manager (FSM) which is a fork of OpenSM with a key focus area of fault tolerance and routing. In [29] the researchers behind the FFM, Jesus Camacho Villanueva, Tor Skeie, and Sven-Arne Reinemo, show that the FFM offers excellent performance and fault tolerance compared to OpenSM. FSM comes with the Fabriscale routing engine (FRE) as default.

We aim to build an efficient, scalable and reliable performance counter collector that will integrate into the FFM. The application will collect performance metrics from all switches in the fabric where the FFM is deployed.

2.3.1 Existing Fabriscale Software

Today, metrics found in the FFM are originating from an event plugin inside the FSM. The event plugin is collecting metrics and pushing them to a central module called fsmonitoring where the performance metrics are processed and stored in an Elastic Search\textsuperscript{15} database. The event plugin is also pushing the fabric topology to fsmonitoring. Fsmonitoring stores

\textsuperscript{14}Converged Ethernet: An enhanced version of Ethernet with support for features that makes Ethernet more reliable and lossless, like InfiniBand.

\textsuperscript{15}Elastic Search: An open-source highly scalable search engine.
the topology in an Arango[31] database\textsuperscript{16}. Fsmonitoring is a core module in FFM and is responsible for communication between the FSM, Elastic, ArangoDB and the web server. This event plugin inside FSM is also a key element for alerting the network administrator when faults occur. For instance, if a switch stops responding, the event plugin will notify the FFM and the web-based GUI will be updated immediately and trigger an alarm. An architectural drawing of the existing Fabriscale software is presented in Figure 2.10.

Figure 2.10: The FFM software architecture

### 2.3.2 Enhancing the Metric Collection

As part of this thesis, we will build a new module to take over the responsibility of performance metric collection from the FFM. A standalone solution of the metric collection has many benefits over an integrated solution. As we previously mentioned Fabriscale is also focusing on Converged Ethernet. Developing a standalone application makes it easy to further evolve the collector to also support Converged Ethernet. Separating the metric collection also means that it can evolve independently of the FSM, and deployment of a new version of the collector in the IB fabric would not generate downtime for the computing cluster. The most significant benefit of taking the metric collector out of the FSM is that it reduces the number of non-critical tasks the FSM has to execute. It is very crucial that the FSM is as efficient as possible when a fault occurs in the network, and consequently has to recalculate paths and reconfigure devices in the subnet.

To give the reader one graphical example of how the collected metrics from the new module built in this thesis will be used, we have included a screenshot of the web-based GUI in FFM. The screenshot, presented in Figure 2.11, displays a real-time graph of the network load on a given switch

\textsuperscript{16}Arango DB: A universal NoSQL database[32].
port together with the load the last 15 minutes. Some of the port error metrics are also presented in this view.

Figure 2.11: Error and performance metrics displayed in the FFM web-based GUI.

2.4 Libraries

The existing code in FFM, and specifically the fsmonitoring module, makes use of many open-source libraries for communication with the FSM. The communication protocol between the FSM and fsmonitoring is well designed, and interfaces in both ends are clean. For serializing of messages with metrics or topology information, the FFM takes use of Google Protobuf[33]\(^\text{17}\). The Protobuf messages are published from the FSM to fsmonitoring using a pair of Zero Message Queue (ZeroMQ)[34] sockets in PUB/SUB mode. The natural way of designing the new metric collection module is to utilize this existing protocol and communication interfaces by modifying them and build support for the new module.

We plan to collect performance metrics in a standalone module, independently from the SM. In order to communicate with the IB switches in the subnet, we will need to use a middleware. As referred to in Section 2.2.1, OFED provides open-source software libraries for IB. Libibmad and libibumad together provides a great interface towards the IBA and has all the functions we need to build a separate metric collector.

\(^{17}\)Protobuf: Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data.
2.4.1 Libibmad and Libibumad

The OFED software stack includes two libraries for management communication in an IB fabric. These two libraries are known as Libibmad and Libibumad.

Libibmad

The Libibmad library is designed for easy implementation of a MAD client and server tools. It is a rich library with a proper implementation of the management protocol. This library declares common structures, fields, and enumerations that are used for different types of MAD communication. Libibmad handles network data and native data from the host and the translation between these. Since it defines field structures for every separate MAD field with individual conversion functions, the marshaling/de-marshaling problems are transparent to most of the application code resulting in a clean, machine independent code. Marshaling and de-marshaling of data are easy since the library itself knows what functions to use for the different fields.

Libibmad supports reliable remote procedure calls (RPC) and pure sending and receiving functions for MADs. It has functions to handle IB address resolution, and it can dump MAD fields to a human readable format.

Libibumad

The Libibumad library is a user-level library with less functionality than Libibmad. It can provide basic information about the IB devices discovered which includes IB names, a list of ports, and specific port attributes. It has support for basic MAD functions such as open and closing of a port.

2.4.2 Google Protobuf

Google Protobuf is Google’s software solution for serializing messages containing data sent between different applications. Protobuf is platform- and programming language neutral, and is practically used by everyone inside Google[35]. Development of Protobuf started at Google in early 2001 but was back then never released to the public. Google released version 2.0, a total rewrite of the software to the public under the Berkeley Software Distribution (BSD) licence in 2008. Before that, Google had multiple projects that they wanted to release to the public, many of which used Protobuf for internal messaging. Open-sourcing these projects meant that
Protobuf also needed to be open-sourced. Google also wanted its users to be able to communicate directly with Google's API using Protobuf, which is way more efficient than text encoded messages. Also, behind the scenes, Google was already converting all incoming API requests to Protobuf. When these decisions were made, Google started a project to rewrite the code since the state of version 1 was in complete disarray[35].

Google Protobuf is designed to be extremely lightweight and fast to serialize and deserialize messages. Messages are encoded in binary, and it makes use of positional binding so that variable names does not have to be sent, ensuring low overhead for messages. Computers are much faster at processing binary than text-based encoding formats like Extensible Markup Language (XML)[36] or Javascript Object Notation (JSON)[37]. Messages are defined in .proto files, which define name-value pairs. An example .proto file is shown in Listing 2.1. The presented file is from Google’s Protobuf documentation, and it contains information about a person. Each message type has one or more uniquely numbered fields, and each field has a name and a value type. Protobuf supports a variety of types, and the example message presents some of them. Fields can be specified to be optional or required, and they can be repeated.

The defined .proto file is compiled to native programming language classes by the Protobuf compiler. These classes implement automatic encoding and parsing of the data specified in .proto files, to the binary format transmitted.

In FSM, Protobuf is used for serializing of all messages sent from the event plugin inside FSM. The FFM Protobuf files define a set of different messages for sending device, topology, performance metrics, and port information to fsmonitoring. Individual messages can be combined and be sent as one single message.

2.4.3 Zero Message Queue

Zero Message Queue (ZeroMQ) is a message-oriented middleware library which resembles Berkely sockets. It is a simple message library that provides a more advanced socket interface than Berkeley sockets offers. ZeroMQ is lightweight and fast by using its brokerless\footnote{Broker: A central message server in the middle of all communication. A Classical star architecture.} architecture. Avoiding the usage of a broker, ZeroMQ reduces the network overhead quite a bit, and the single point of failure is removed from the messaging service. When a broker architecture is used, all messages have to be passed through the broker who can lead to the broker itself to be the bottleneck of the system.

ZeroMQ supports many different communication patterns and various
transport services. ZeroMQ provides a lot of freedom, and the user can easily optimize message passing. The four supported types of transports are INPROC\(^{19}\), IPC\(^{20}\), MULTICAST\(^{21}\), and TCP\(^{22}\).

ZeroMQ defines the following four types of messaging patterns that can be used with ZeroMQ sockets.

- **REQUEST/REPLY** - This is bidirectional, load balanced and state based messaging. This messaging pattern is very widely used and can be found in web protocols such as HTTP. One end uses a `send()` function, the other end is calling `receive()`.

- **PUBLISH/SUBSCRIBE** - The publisher is broadcasting messages to subscribers, as a radio station and its listeners. This messaging pattern is scaling with many subscribers. Unfortunately, subscribers can miss information if they connect after the broadcast of a message.

- **UPSTREAM/DOWNSTREAM** - This is similar to request/reply. The difference is that it is not required to send the reply back to the requester. The reply can be pushed down a pipe to another receiver. Typically a collector of some sort.

---

\(^{19}\)INPROC: An In-Process communication model.
\(^{20}\)IPC: An Inter-Process communication model.
\(^{21}\)Multicast: UDP Encapsulated multicast messages.
\(^{22}\)TCP: Transport Control Protocol, a reliable transport protocol used in the Internet.
• **PAIR** - This is also very similar to regular request/reply sockets. This mode is bidirectional, no specific states at the endpoints, and it can only be one connected peer. This is a pattern for connecting two threads in a process, not to be confused with "normal" pairs of sockets.

### 2.5 IBSim

For supporting development testing and verifying IB management tools, OFED has developed a simulator for the IB management model. This tool is named IBSim. IBSim emulates a real IB subnet and fabric behavior. The simulator supports any regular IB management tool, like OFEDs IB diagnostics tools and OpenSM. It works out of the box with third party IB tools and runs in user-space. The simulator has built-in-support for agents like the SMA and PMA. Usually, management traffic from a Linux host is sent to the subnet using the `/dev/umadX` file descriptor and trapped down to `ib_umad.ko`. When running IBSim, this interface is replaced using a preloaded shared library called `libumad2sim.so`. This library is a wrapping library distributed together with IBSim, and it conveys MADs from and to the IB management application to IBSim.

IBSim has a simple console command interface and can also be controlled using UNIX or inet sockets. Some simple simulations can be done inside IBSim, like adding random packet drops and taking up and down ports. IBSim lets the operator set values to performance counters, i.e. PortXmitPackets, but it has no concept of links or bandwidth, and no function to simulating network load.

The usage of IBSim in this thesis has two purposes. In chapter 4 we need to use IBSim for running large-scale simulations to verify that the implemented software in this thesis is supporting it. We will also use IBSim to simulate some of the same experiments that we will run on hardware.

### 2.6 Summary

This chapter presented the theory and background for building an efficient monitoring software for the IB interconnect. It introduced the core services of the IBA such as addressing, switching and quality of service. It presented the layered stack, and we talked about appropriate headers. It gave a detail view of the management model and its entities. It showed how communication with IB devices for management is done. Key performance and error metrics from the IBA was given.

---

23ib_umad.ko: InfiniBand kernel module for user-level access to MAD services
This chapter discussed monitoring of an IB fabric and gave more motivation on why to build a metric collector plugin for the FFM. Some of the existing tools that have been developed and are in use were examined. The key aspects of efficient monitoring were given.

It introduced an overview of the existing software in FFM and some of its capabilities. Furthermore, this chapter described how we in this thesis could enhance the FFM by building a separate metric collection module. It established the purpose of the metric collection, and some of the requirements for the new software we aim to build was given.

The different software libraries and middleware we need to understand to build the metric collection software was presented, and the reason for choosing them was given.

At the end of this chapter, we presented IBSim, an IB management model simulator. We explained how MADs are conveyed into the simulator and mentioned some of its simulation possibilities. It was mentioned what we are planning on using IBSim for later in this thesis.
Chapter 3

Design and Implementation

This chapter dives into the design and implementation choices that we had to make to build the performance metric collector in this thesis. It presents the area of use of the software and some background for the choice of programming language. This chapter is dedicated to the first process of the design paradigm as detailed in section 1.3, particularly the requirements and specifications stages. All the requirements is presented, and each of the specifications and how they build up under the requirements. Combined, the presentation of these specifications answers the first group of research questions that were defined in section 1.2.

3.1 Area of Use

The purpose of the metric collection software built in this thesis is to facilitate IB subnet administration. IB subnets have grown in size and adoption over the last decade, and subnet management is becoming a more challenging task. Metric collection from IB devices gathers health and performance related attributes that are vital for a network administrator when administrating the subnet. The metric collector described in this chapter will be a vital part of the FFM by inputting metrics collected from switches in a subnet to the core FFM module fsmonitoring. The software plugin developed in this thesis will be named perfswiper. The name follows the name style used in the OFED diagnostics tools package. In this package, there is a tool called perfquery which is a command line tool for printing out performance metrics of an addressed switch. We took the first part of the name and combined it with an informal signature (swiper), which we added at the end.
3.2 Programming Language

There were two programming languages considered for the development of perfswiper. Golang and C. Perfswiper will operate as a component between the FFM module fsmonitoring and switches in an IB fabric using the low-level OFED libraries for the IBA management model: libibmad and libibumad. Fsmonitoring is written in Golang, and the two OFED libraries are written in C. One of the main requirements of perfswiper is that it can communicate with both IB switches and fsmonitoring. Golang does support calling native C functions using a wrapper named cgo so it would be possible to write perfswiper entirely using Golang. Perfswiper could also be directly implemented inside fsmonitoring as a Go package if it was to be developed in Golang. One weakness with a Go package solution is that the metric collection could not be run on a separate host from fsmonitoring. Some of Fabriscale’s customers might want to be able to do this.

It was decided that since the author is experienced in C and unfamiliar with Golang, C was the best choice. It would speed up the development process, and the room for error was considered lower when writing in C, even though C is a less modern language with fewer features. Since the communication between the existing FFM module and perfswiper should be done using Google Protobuf (which is programming language neutral), it should not be an issue that the two applications are written in different languages. We want to keep perfswiper as simple as possible to avoid bugs and make sure the code is robust. Therefore, we have decided that perfswiper should not be multi-threaded. Additionally, since a significant part of perfswiper is to wait for I/O, spawning multiple threads would probably not result in any extreme speedup\textsuperscript{1}[38].

3.3 Requirements

The first aim of this thesis was to build a metric collection software for IB networks. To archive this first goal, we decided to use the design paradigm which was formalized by the ACM board of education. This paradigm was described in section 1.3 Research Method. The first step of this method of research is to state the requirements of the program. To build the metric collection software we have established the following requirements:

- **Collect error and performance metrics from all IB devices**
  Perfswiper should be able to assemble metrics from all IB devices regardless of the manufacturer. It must follow the IBA management specifications.

\textsuperscript{1}Speedup: Improvement of speed when running the same task with more amounts of resources.
• **Use low-level in-band communication** Perfswiper should utilize efficient low-level communication over the IB network.

• **Configurable** Perfswiper must be able to obtain a configuration when launched. It must be able to configure a set of metrics to collect, and an interval between a fabric swipe.

• **Communicate with existing Fabriscale software** Perfswiper must be able to communicate with fsmonitoring to transmit the collected metrics to it for further processing and long-term storage.

### 3.4 Specifications

To fulfill the requirements, perfswiper has the following specifications:

• **A structure for metric representation** Perfswiper must implement a way of representing all the available performance metrics supported by the IBA. It must be possible to store metrics temporarily in a structure located in memory before they are transmitted to fsmonitoring.

• **Awareness of switches** Perfswiper must have knowledge about all switches it can reach in the connected fabric. Perfswiper must know the switches by their LID. When a new switch is inserted into the subnet, or if a switch is removed, perfswiper must be noticed and update its list of switches to monitor.

• **A configuration file** Perfswiper must define a configuration file format and be able to read and interpret it. The file should allow the user to configure multiple settings and variables needed for the communication. Some settings include the swipe interval and timeout value if a switch does not respond. It should also be possible to specify which counters perfswiper should collect from the fabric. It should be easy to implement new configuration keywords to support more settings in a future version of perfswiper.

• **A function for collecting metrics** The metric collecting function of perfswiper must utilize the libibmad and libibumad libraries to query switches in-band using MADs. It must only collect and store the metrics that are configured.

• **A function to push metrics to fsmonitoring** Perfswiper should be able to communicate with fsmonitoring efficiently. The software should coalesce multiple port metrics from the subnet and transmit them in one message to fsmonitoring to minimize transmission overhead.
• **Be run as a daemon** Perfswiper must be able to run as an independent process in the background, with no terminal hosting it. Running as a daemon must be the default behavior, but it should also be able to run perfswiper in the foreground within a terminal.

• **Robustness** Perfswiper must be able to handle changes in the fabric topology and sudden loss of contact with switches or fsmonitoring without crashing. It must also properly deal with errors that occur and ensure that no garbage will be transmitted to fsmonitoring.

• **Debugging functions** Several debugging functions and assert functions must be included while being developed. Perfswiper should also safely destroy data structures and deallocate memory it has used to prevent memory leaks.

### 3.5 Architectural drawing

Before the reader proceeds into the individual sections presenting each of the specifications of perfswiper, we suggest that the illustration given in Figure 3.1 is examined. This illustration shows how the integration of perfswiper and the FFM components will look like when finished. The integration will consist of many communication pipes, and the illustration will help the reader grasping the concepts when we in the next sections present the design of these pipes.

### 3.6 Metric Representation

Perfswiper will need to save the collected metrics temporarily in a data structure before they are transmitted to fsmonitoring. We have defined a C struct named `perf_metric` that will be used to store a metric sample from a given port. Since we want to coalesce metric messages for efficiency, we will also need to implement an array to store multiple `perf_metric` structures.

Perfswiper will need to collect the LID and port number the counters are originating from together with all the port counters and a timestamp. All these variables together represent one set of port metrics. All this information will be stored in one single `perf_metric` struct to keep the context of the data combined. Perfswiper will need this context to be able to transmit data to fsmonitoring without being ambiguous. The timestamps are included since it is a necessity so events can be tracked. The web-based GUI included with the FFM also needs them. To correctly present network load in graphs, or display live bandwidth usage, the web-based GUI needs
to have a concept of time. Timestamps stored together with the metrics for long time storage also makes it possible to replay the bandwidth pattern in a given timespan. The FFM specification defines that this variable should follow the UNIX epoch time system\(^2\) and hold the number of milliseconds from January the 1st, 1970, up to the point in time when the port was sampled.

The variable names inside the `perf_metric` structure are originating from the existing code written by the OFED software stack developers. It is important for us to have clean code\(^3\) in `perfswiper`, and the use of established variable names for the same usage inside a specific field of programming just makes sense. The `perf_metric` struct contains entries for all the mandatory error and performance counters defined in the IBA. In the previous chapter we presented an overview of the mandatory metrics in Table 2.3, and now we introduce the struct used for temporary storage of metrics in Listing 3.1.

---

\(^2\)UNIX Epoch: The number of milliseconds that has passed since Coordinated Universal Time (UTC) 00:00:00, 1 January 1970.

\(^3\)Clean code: an expression used about nicely and unambiguously written source code.
struct perf_metric{
    uint32_t lidselect;
    uint32_t portselect;
    uint32_t symbolerrors;
    uint32_t linkrecovers;
    uint32_t linkdowned;
    uint32_t rcverrors;
    uint32_t rcvremotephyerrors;
    uint32_t rcvsrelayerrors;
    uint32_t xmtdiscards;
    uint32_t xmtconstrainterrors;
    uint32_t linkintegrityerrors;
    uint32_t excbufoverrunerrors;
    uint32_t vl15dropped;
    uint32_t xmtwait;
    uint64_t portxmitdata;
    uint64_t portrcvdata;
    uint64_t portxmitpkts;
    uint64_t portxmitpkts;
    uint64_t portunicastxmitpkts;
    uint64_t portunicastrcvpkts;
    uint64_t portunicastrcvpkts;
    uint64_t portunicastrcvpkts;
    uint64_t portmulticastxmitpkts;
    uint64_t portmulticastrcvpkts;
    uint64_t portmulticastrcvpkts;
    uint64_t timestamp_ms;
};

Listing 3.1: struct perf_metric in perfswiper.h.
3.7 Finding the Switches to Monitor

To be able to collect metrics from the IB fabric, perfswiper must have knowledge about available switches. A list of reachable switches that perfswiper should monitor must be stored in memory, and it must be regularly updated to reflect the subnet. There are several ways of gathering information about available devices on an IB network. One approach for obtaining topology awareness that was considered was to build a module inside perfswiper that continuously probed the network and looked for changes using libibmad. This method would introduce more MAD traffic into the IB network which is one of the crucial elements we want to avoid. As we stated in section 3.2, perfswiper will be built on one single thread, because of this, adding this additional swipe of the fabric would cause increased latency between each metric swipe, and potentially reduce the resolution of the collected metrics too much. Another possibility could be to rely on existing OFED binaries. We could run them in intervals and save the topology to a file that perfswiper could interpret. One tool that does this job is ibnetdiscover. This method is an easy way of finding available devices, and it would be quick to develop, but is probably the lousiest way of doing it. Relying on ibnetdiscover would also add additional MAD traffic in the subnet. Changes in the network are not likely to happen often, but to be able to notice the change quickly, ibnetdiscover would have to run in short intervals. This would lead to a lot of unnecessary MAD traffic. Perfswiper would also have needed to check if the file generated by ibnetdiscover have been changed with the same short interval, which leads to a huge waste of resources. Reducing the probe and file checking interval to save resources would result in a lower liveness in perfswiper. The time between a topology change is happening, and perfswiper is adjusted could potentially be long due to all the processing steps relying on disk I/O operations.

As presented in the technical background, the required SM already does a swiping process to learn about the switches and HCAs present on the network. Since the SM also receives traps when some event occurs, it is likely to be the first to be aware of changes in the network. The event plugin inside FSM is periodically sending the topology to fsmonitoring, and it also notifies fsmonitoring immediately if a switch suddenly is unreachable. These two features declare that fsmonitoring already has all the topology information needed by perfswiper. We can thus extend fsmonitoring to handle topology requests from perfswiper, and make sure that fsmonitoring is alerting perfswiper when it gets a notification that a switch in the subnet is unreachable or if new switches are discovered. This way of getting to know available switches involves some additional programming, but the resulting process of gathering this information would be superior. Since fsmonitoring will be the destination for the collected metrics, it is fair for us to rely on this component and have it as a requirement for getting knowledge of available switches.
3.7.1 Changes in Fsmonitoring

As presented in Section 2.3.1, fsmonitoring exposes a web API. This API could be used to request information about the switches in the network. Fsmonitoring also has a WebSocket[39] exposed that are being used to convey topology change alerts from FSM.

Perfswiper could be designed to include an HTTP client for requesting information, and it could listen to the WebSocket for alerts. We considered this solution to have too many weaknesses to be feasible. A major shortcoming is that the HTTP protocol is text based and very talky with a request and response schema for every message. Additional overhead would also have been introduced by the Transmission Control Protocol (TCP)/Internet Protocol (IP) stack needed to use HTTP. We typically categorize HTTP as a high-level communication protocol since it operates on top of multiple underlying communication protocols. Building an HTTP client inside perfswiper would increase the complexity and size of the application quite a lot by introducing more external libraries.

Fabriscale has already established ZeroMQ as the standard communication interface between the different components in FFM, and ZeroMQ already needs to be included in perfswiper for transmitting metrics to fsmonitoring. If perfswiper could use ZeroMQ to collect switch information from fsmonitoring, that would have been a superior solution compared to rely on the web-based API. There is currently no support in fsmonitoring to accept and process data requests over ZeroMQ sockets, so a request/response interface using ZeroMQ needs to be designed and added to fsmonitoring. The illustration that we presented earlier (Figure 3.1) had these two communication interfaces included. In the following sections, the design and reasoning behind these two are given in detail.

3.7.2 Designing the Request and Response Protocol

Between fsmonitoring and the event plugin in FSM, there are two communication pipes based on ZeroMQ sockets. FSM has one publisher socket that fsmonitoring is subscribing to for topology and metrics updates, and an additional socket in FSM used for receiving commands. A typical command could, for instance, be a request for the network topology. The command socket that listens for commands in FSM is created using the ZeroMQ router type. The socket in fsmonitoring that connects to FSM is a dealer type. The router and dealer sockets are non-blocking request and response sockets that add context to the message when received. This context ensures that an incoming message can be traced back to the sender by the receiving application. Since these sockets have a non-blocking property, they scale well. A low-level way of issuing commands and requests
to fsmonitoring could also benefit Fabriscale in the future if FFM, for instance, needs to be distributed to handle scaling. Fabriscale might also consider making this low-level API available to their customers as some customers might want an alternative to the HTTP API.

Request Protocol

To follow the already established practice in FFM for low-level requests, a ZeroMQ router socket must be implemented in fsmonitoring. Fsmonitoring must bind to the socket, and perfswiper will connect. This is the same order as between FSM and fsmonitoring. The router socket in fsmonitoring will work as a command interface, and perfswiper will utilize this to send requests. The command pipe between FSM and fsmonitoring are using strings agreed upon as keywords for commands. For instance, a request for topology sent to FSM has the keyword REQUEST_TOPOLOGY. To follow this practice we decided that the keyword perfswiper will send to fsmonitoring when requesting information about available switches will be REQUEST_SWITCHES. Fsmonitoring should, when a request for switches is received, build a response message with a list of all present switches and transmit it to perfswiper. This takes us to the next section, the response protocol.

Response Protocol

When designing the response protocol we had to solve two design problems: How to serialize the list of switches and the number of ports it has, and how to transmit it to perfswiper. Existing communication in FSM is using Google Protobuf for data serializing over a ZeroMQ pipe. To keep things in FFM clean, perfswiper will follow this already established data serialization method. We must define a Protobuf message format that is describing a list of available switches. We also have to create a new set of ZeroMQ sockets to transmit the response over. We do not want to use the router and dealer pipe for the response; Since fsmonitoring is going to inform perfswiper about changes after the initial command, a publish-subscribe architecture (as presented in Section 2.4.3) is more fitting, and it is cleaner to use only one pipe in perfswiper for these messages.

We have defined a Protobuf message called PSNodeInfo which will describe one switch found in the subnet. The letters P and S at the beginning of the message name stand for perfswiper and is added to not confuse the message with the already existing message format used by FSM to send extensive node information to fsmonitoring.

The message format is designed to support a future evolving of perfswiper
syntax = "proto2";
message PSNodeInfo {
  optional IBNodeInfo ibNodeInfo = 1;
  optional ENodeInfo eNodeInfo = 2;

  // message format describing an InfiniBand switch
  message IBNodeInfo {
    required uint32 lid = 1;
    required uint32 num_ports = 2;
  }

  // message format describing an Ethernet switch
  // (definition fullfilled when Ethernet support is added to perfswiper←
  )
  message ENodeInfo {
  }
}

message PSNodeInfo {
  repeated PSNodeInfo nodes = 1;
}

Listing 3.2: Protobuf message PSNodeInfo.

where support for monitoring of Ethernet switches is added. To support both IB and Ethernet switches, the PSNodeInfo message format describes two types of optional nested messages; an IBNodeInfo message, and an ENodeInfo message. In general, perfswiper needs to have a minimum of two attributes to be capable of handling metric collection, namely to identify a given switch and the ports of the switch to monitor. Perfswiper needs to know the LID of the device, and the number of ports it has. The IBNodeInfo message that is representing an IB switch must, therefore, contain the LID and the number of ports. The ENodeInfo message format will not be defined as part of this thesis. The specification of it is left to be fulfilled during the design process of adding Ethernet support in perfswiper.

Responses sent from fsmonitoring will be transmitted using a ZeroMQ publisher socket, which fsmonitoring should bind. By making the publisher end of the communication pipe bind, the system is able to handle multiple subscribers that can connect to the publisher socket. At the moment, perfswiper will be the only application to subscribe to this socket, but we want to design the communication pipes to handle multiple subscribers in the future.

The complete specification of a PSNodeInfo Protobuf message is presented in Listing 3.2.
3.7.3 Complete Protocol Specification

Perfswerper will during initialization request switch information from fsmonitoring by transmitting a REQUEST_SWITCHES message to a ZeroMQ router socket in fsmonitoring. Fsmonitoring must answer with a PSNodeInfo message using a ZeroMQ publisher socket which perfswerper is subscribing to. When topology changes happen, fsmonitoring must update perfswerper with a new list of available switches using the same publisher socket. The two communication pipes must be agreed upon and configured in both perfswerper and fsmonitoring.

3.7.4 Handling PSNodeInfo in Perfswerper

When perfswerper is launched, it will have to initialize multiple data structures and sockets before metric swiping of the subnet can begin. One of the key data structures is the list of switches to query. As part of the initialization, perfswerper must request the available switches in the network from fsmonitoring. The request function must block until a response containing a PSNodeInfo message is received from fsmonitoring. A timeout must be created so that perfswerper will not hang indefinitely if fsmonitoring is not running, or if the communication pipe has been misconfigured.

When a PSNodeInfo message is received from fsmonitoring, perfswerper must parse it and update its in-memory list of available switches. The function that updates the list of available switches will be used both in the initialization process, and when perfswerper are notified about a topology change. We want the algorithm that updates the list to be efficient and fast so that swiping of the subnet is not delayed unnecessarily. When a message is received a linked list will be built with the LID and port number information received. If it is not the first PSNodeInfo message that is received during initialization, but an update during operation, the previous list stored in memory will be destroyed, and a new list is built. We do not want perfswerper to use time and resources on checking what differs from the received message and what is stored in memory, so our solution is just to discard the old list and build a new.

3.8 Configuration

Fabriscale requested that it should be possible to configure the interval between each metric swipe and furthermore possible to configure a timeout reflecting for how long the system should wait for a switch to respond before giving up. They also wanted a way of configuring the metric counters to collect from switches. To adhere to the requirement of being
configurable, perfswiper would need a way of acquiring configuration from the subnet administrator. One common way of providing configuration to an application is to pass arguments to the process when it is launched. Perfswiper will need to have an argument parser to handle these during initialization. This argument parser will have a vital role for the use of perfswiper for performing experiments in the next chapter.

Perfswiper is designed to parse some simple arguments, but a configuration file is also needed for the more extensive configuration options requested by Fabriscale. We also found the configuration file useful for configuring the integration with fsmonitoring.

3.8.1 Argument Parser

The argument parser in perfswiper is used for acquiring basic configurations. For instance, if a non-standard location for the configuration file is desired, it can be passed to perfswiper with the \texttt{-c} or \texttt{--config} flag. Another argument that can be given is a \texttt{-m}. This flag toggles \textit{master mode} on/off which is a mode that we will use when carrying out the experiments that are designed in the next chapter. The argument parser is built using \texttt{getopt_long()} from the standard C library. It was decided to use the \texttt{getopt_long()} function since the author has used it before and since it does the job well. If the users pass an invalid argument, a synopsis is printed to the terminal. A user can also acquire this synopsis by passing \texttt{-h} as an argument.

3.8.2 Configuration File

For the more extensive configuration options that Fabriscale requested, such as specifying which error and performance counters to collect, we found that a configuration file is much more suited. The list of arguments would be very long, and the interface for configuring it would not be user-friendly to support a list of counters. Since we decided to have a configuration file, it would make sense to colocate all options here for user-friendliness, such as the swipe interval and timeout.

As part of the integration of perfswiper into fsmonitoring, communication sockets between the two modules must be agreed upon and passed to both modules. Fabriscale is already using configuration files in fsmonitoring and FSM for the similar task. To follow this established approach, we have added three configuration keywords to specify the address the ZeroMQ sockets should use for communication.

In Listing 3.3 we present the current configuration file that is included with perfswiper. Documentation for each of the supported configuration
parameters is also included in this file. We have designed it very basic with a key and value pair divided by the equals character, and it will be easy to extend this file in the future with additional settings. The default location where perfswiper will look for the configuration file is here: 

/etc/perfswiper/perfswiper.conf. As previously mentioned a custom path can be given with the -c argument.

3.8.3 Storing the Configuration

Inside perfswiper, the configuration will be stored in two global structs. One for the general configuration such as the swipe interval and communication sockets, and one for all the error and performance counters. It was decided that the default configuration should enable all supported counters and that the network administrator has to disable the desired counters explicitly in the configuration file. Separating these two areas facilitated setting the default behavior. By separating them, the port counter struct can be initialized to enable all counters like this ‘counter_conf_t

counter_config = { 1 };’ which is clean and makes initializing the counter one less thing to remember to modify when extending perfswiper for additional counters in the future.

3.8.4 Configuration File Parser

To interpret the configuration file, perfswiper will need a parser. The parser should be robust and handle errors and misspellings in the configuration file. The parser must ignore the rest of a line if a # is found. For optimal user-friendliness, it is important that the parser gives good error messages when something is wrong. It should not under any circumstance hang if there are input errors. Configuration settings are specified with a key and value pair separated by the equals character (=).

Since the configuration file is a place where a user inputs data to perfswiper, the design of the parser needs to be focused on error checking, and it must be robust. White spaces must be stripped, and verification of the found key and value pair must be done. If any of the values are empty, perfswiper should print a meaningful error message helping the user to correct the configuration file.

Two utility functions were designed to help parsing the key-value pairs and to avoid over-reads and invalid inputs: the functions cfkey() and cfval(). Both of these functions take a line from the config file as input. The c and f in the name stand for config and find. These functions are returning pointers to the key and value respectively in a line read from the config file. One should be aware that the cfkey() function is zero terminating
# /etc/perfswiper/perfswiper.conf - Configuration file for Perfswiper

# Version 1.0

# set the zmq (pub) url to use when sending metric updates
MetricServer = ipc:///tmp/fsm_metric_updates

# set the zmq (dealer) url to use for requesting topology updates
CmdServer = ipc:///tmp/fsmonitoring_cmd

# set the zmq (sub) url to use for getting topology updates
TopoServer = ipc:///tmp/fsmonitoring_topo_updates

# set the interval in seconds of which perfswiper should swipe the subnet for counters
# 0 = as fast as possible (no delay added)
Interval = 1

# set the timeout time in milliseconds to be used for metric query
# 0 = use the default libibumad timeout value (default)
# a negative number means no timeout is used
Timeout = 0

# set the number of metrics to combine before transmitting to fsmonitoring.
# default and maximum is 36
NMetrics = 36

# set the port counters to collect in a swipe
# Port counters must use their official IB spec names
# (all mandatory counters are enabled by default and must be explicitly disabled in this configuration file if desired)
# 0 = disabled
# 1 = enabled

# data counters
PortXmitData = 1
PortRcvData = 1
PortXmitPkts = 1
PortRcvPkts = 1
PortUnicastXmitPkts = 1
PortUnicastRcvPkts = 1
PortMultiCastXmitPkts = 1
PortMultiCastRcvPkts = 1

# error counters
PortXmitWait = 1
SymbolErrorCounter = 1
LinkErrorRecovery = 1
LinkDownedCounter = 1
PortRcvErrors = 1
PortRcvErrorsPortRcvRemotePhysical = 1
PortRcvSwitchRelayErrors = 1
PortXmitDiscards = 1
PortXmitConstraintError = 1
PortRcvConstraintError = 1
LocalLinkIntegrityErrors = 1
ExcessiveBufferOverrunErrors = 1
VL15Dropped = 1

Listing 3.3: Default perfswiper configuration file.
Listing 3.4: String utils used to get key and value from the configuration file.

the string right after the key is found. These two functions are presented in Listing 3.4.

The extracted key value is checked against the list of predefined configuration settings that are seen in Listing 3.3. If the specified key is not found in the list of available options, perfswiper shall print a helpful error message including the keyword and then terminate. A distinct message will help the user identifying the error and correct it quickly. The error message should sound something like this; Found invalid configuration option <insert found key>, please check the configuration file for errors. When a valid key is found, the program should copy the corresponding value into the struct holding the appropriate configuration option.

3.9 Metric Collection

A significant part of perfswiper is the functions used to collect error and performance metrics from IB switches. In chapter 2, we discussed metric collection and monitoring tools currently available, but to reiterate; many of the existing tools rely on OFED binaries such as perfquery to collect metrics, and they use software daemons on computing nodes for gathering more information than these binaries provide. These daemons together with OFED binaries introduce additional network traffic in networks already
heavily loaded. Other tools, such as FabricIT is vendor specific and will only work with switches produced by one particular manufacturer.

One of the requirements of perfswiper is that it can collect metrics using low-level in-band communication by utilizing libibumad and libibmad. Perfswiper must support all mandatory error and performance counters specified by the IBA, and it must be easy to implement support for custom or vendor specific counters in the future.

### 3.9.1 Metric Collection Call Flow in Perfswiper

The overall design of the metric collector inside perfswiper is presented in Figure 3.2. Later in this chapter, we will refer to functions in this graph, and we will present the functions listed in detail. The left topmost function in the call graph is responsible for iterating through the list of switches to monitor. It resolves and sets up some IB prerequisites before it calls `get_perf_counters()`, which handle the PMA query and the temporary storage of the collected metrics. The function `_do_mad_rpc()` is a function in libibmad that is used to send multiple types RPCs.

![Call flow in perfswiper when issuing a RPC request.](image)

### 3.9.2 Communicating with IB Switches

It was early decided that low-level libraries should be used to communicate with IB switches. In Section 2.4.1 we presented two OFED libraries written in the C programming language. These two libraries facilitate communication with IB equipment, and they have support for querying of performance counters.

The libibmad documentation states that client applications developed using the library should use the RPC mechanisms offered in the library. Since
the documentation is obscure and insufficient, the source code was explored to try to find useful functionality. The source code of perfquery was also inspected. The function named \texttt{pma\_query\_via()} defined in the libibmad library (\texttt{mad.h}) was considered appropriate for our use. \texttt{pma\_query\_via()} handles the low-level setup and issuing of an RPC call to the PMA. It returns a pointer to the buffer containing the reply. This function only supports one specific port on one particular switch. Perfswiper will use this function repeatedly to request all available information that can be collected via the PMA on a switch.

In Listing 3.5 the function signature of \texttt{pma\_query\_via()} is listed together with all its arguments. The \texttt{pma\_query\_via()} function needs a receive buffer, a destination, a timeout value, what attributes to query, and what source IB port to use. The function crafts an RPC MAD and sends it out on the port specified by the caller. It waits until the response is received before returning to the caller.

In section 2.1.5 we wrote about the PerfMgt packets and how they are a subclass of MADs. The function mentioned above handles the creation of such a packet. It crafts a general IB RPC packet and sets the management class to \texttt{IB\_PERFORMANCE\_CLASS} and the RPC method to \texttt{IB\_MAD\_METHOD\_GET}. Further, it sets the \texttt{IB\_PC\_PORT\_SELECT\_F} field in the packet to indicate what port perfswiper is interested in. What counters to collect is specified by the id argument when the function is called.

Before calling \texttt{pma\_query\_via()} perfswiper must set up a \texttt{ib\_portid\_t} struct by calling \texttt{resolve\_portid\_str()} with the LID as a string and an empty \texttt{ib\_portid\_t} struct. Some local identifiers such as the local IB adapter name and port number are also passed to the resolve function. This resolve function configures which QP and SL to use for the transmission. All of these settings are stored in the struct and the struct which the libibmad documentation refer to as the endpoint address structure.

When the \texttt{ib\_portid\_t} struct for the remote port is initialized, perfswiper can execute the RPC. Perfswiper must call \texttt{pma\_query\_via()} with \texttt{IB\_GSI\_PORT\_COUNTERS} as the id argument. This identifier specifies that perfswiper are interested in the mandatory GSI port counters which we commonly refers to as the error and performance counters.
3.9.3 Parsing the Response

The RPC response will be stored in the receive buffer by pma_query_via(). The response is a complete binary encoded MAD. Libibmad offer a function to decode MADs using predefined variables and macros to indicate offsets in the MAD. Perfswiper will use the function mad_decode_field() to decode each of the individual error and performance counters. The decoding function takes a pointer to an encoded MAD as the first argument, a single field name (specified in mad.h) as the second argument, and a variable to hold the value as the third argument. mad_decode_field() has no return value.

For each of the mandatory counters defined in the IBA, there is an identifier coded in mad.h. Perfswiper will go through all of them and extract the values and store them in the corresponding field in an instance of the perf_metric struct which was presented in section 3.6.

As discussed in section 2.1.5 modern IB switches has support for extended 64-bit long transmit and receive data counters. The legacy 32-bit counters are still the default counters retrieved when querying for IB_GSI_PORT_COUNTERS. We want perfswiper support both the legacy 32-bit counters as well as the new 64-bit versions. To control what counter we can collect, each switch will be queried for its capabilities. The query for capabilities uses the same libibmad function as introduced in Section 3.9.2. Instead of calling this RPC function with the request for counters, we use CLASS_PORT_INFO. The response will work as a capability mask which we later can check for extended (64-bit) counter support.

Perfswiper will issue the capability RPC message after collecting the error counters. It will check if IB_PM_EXT_WIDTH_SUPPORTED is inside the capability mask. This C macro is defined in iba_types.h and it is presented in Listing 3.6. The CL_HTON16 macro is converting a 16-bit value from host byte order to network byte order. The macro is creating a bitmask that is used to check if the 9th bit in the capability mask is set. If it is set, this indicates that support for extended counters is available[12]. If support for extended counters is available on the monitored switch, a new RPC call to the PMA with IB_GSI_PORT_COUNTERS_EXTENDED must be done to collect the extended counters.

---

4 Byte order refers to the way a computer interpret a sequence of bytes.
When the `pref_metric` struct is populated with all counters, it is placed on a waiting list with previously collected ports. The list is used to gather multiple port metrics before they are transmitted in a coalesced message to `fsmonitoring`. More on this in the next section.

### 3.9.4 Support for Vendor Specific Counters

In the future, Fabriscale wishes to extend what counters `perfswiper` will collect. We know that Mellanox has some error and performance counters in their equipment that their management solution `FabricIT` can collect. Support for additional counters can be added in `perfswiper` by issuing some additional capability requests to a switch. For `perfswiper` to be able to interpret the capability mask and sending vendor specific queries, documentation of the MAD format and what counters needs to be provided from Mellanox or other vendors. The in-memory storage of metrics and the configuration file needs to be updated for extended support for counters.

### 3.10 Pushing the Collected Metrics

After `perfswiper` has collected a certain amount of metrics, it will have to push them to `fsmonitoring`. The default behavior shall be to gather all port metrics from a given switch before transmitting them to `fsmonitoring`. If `perfswiper` is deployed in topologies with switches that have more than 36 ports, it will send metrics to `fsmonitoring` after collecting 36 ports. This value is chosen based on the fact that the larger switches that Mellanox currently offers have a scaling factor of 36 ports [40].

The limitation in the number of samples to coalesce is set to minimize jitter between port samples. Too many locally buffered metrics will increase the time `perfswiper` has to use on executing code related to transmitting messages to `fsmonitoring`, and it will increase the delay before regular swiping of metrics can begin again. The network administrator has the option to configure the number of metric messages to coalesce in the `perfswiper` configuration file to any lower number than 36 if higher resolution is wanted. The number of metrics to combine also impacts how often `perfswiper` can check if an incoming topology-change message has been received. After the configured number of metrics to send has been transmitted, `perfswiper` will return to the ZeroMQ poller and check for incoming topology update messages from `fsmonitoring` before returning to the metric swipe.

Today, `fsmonitoring` is receiving port metrics from the event plugin inside FSM. Metrics are transferred over a pair of publish-subscribe ZeroMQ
sockets and use Protobuf as the serializer. This communication pipe can be seen in the architectural illustration that was presented in section 3.5 (Figure 3.1). FSM is using the same publisher socket for pushing metrics as it uses to transmit topology messages. The ZeroMQ publish-subscribe channel support multiple publishers and one single subscriber so perfswiper can join in on the established communication pipe. Since the already established pipe and protocol has been working fine for Fabriscale, we do not see any reason to develop a new protocol or create a new channel. By joining in on the existing channel and protocol, we minimize the number of changes that need to be coded in fsmonitoring related to communication and processing of metrics. All the current code in fsmonitoring will come to use, and no new components for processing will be necessary to develop.

Some minor changes have to be done in both FSM and fsmonitoring so that perfswiper can utilize the existing communication pipe alongside the FSM event plugin. In Figure 3.1, two separate SUB sockets for receiving topology updates and metrics are illustrated in fsmonitoring. These two components will in the source code share the same ZeroMQ socket.

### 3.10.1 Changes in FSM and Fsmonitoring

If a ZeroMQ publish-subscribe channel is configured with the subscribing side as binding, it allows for multiple publishers to connect to a single subscriber. The source code of FSM and fsmonitoring was inspected, and it was discovered that currently, the channel between FSM and fsmonitoring is doing the opposite of what is needed for our enhancement. In fsmonitoring, the receiver code must be altered, so the socket is bound to instead of connected. In the event plugin in FSM the socket must be changed to connect. No other changes need to be done since we want to make use of the established communication protocol.

### 3.10.2 Protocol for Metric Updates

Perfswiper shall join in on the current communication pipe between FSM and fsmonitoring. This communication channel is used for multiple purposes, such as topology updates, node information, and metric updates. Fabriscale has defined a protocol to be used over this pipe so that fsmonitoring can distinguish between these different messages. The sender must craft a ZeroMQ packet consisting of two frames. The first frame acts as an identifying header and should be 64-bit long. This header must contain the type of message that is following. Fabriscale have defined an enum named \texttt{fep_msg_t} to keep all message types used over this pipe. The second entry in this enum is called \texttt{FEP MSG METRICS} and has the value of the number one. This entry identifies that the second frame in the ZeroMQ
packet contains the binary-encoded metrics.

The format of the second frame shall contain a binary encoded Protobuf message. Fabriscale has already defined a binary format that can be used in perfswiper without significant changes that would break the versioning. Since Google Protobuf supports fields to be set as optional, it is possible just to edit the fields that perfswiper does not have any information about, and leave them unset when the Protobuf message is encoded. In Listing 3.7 the edited version of the Fabriscale Protobuf format is presented. Fields that were edited from required to optional were: node_guid, node_type, node_name, and link_state. These are concepts that are out of the scope of perfswiper. Since we wanted to coalesce metrics, this Protobuf format has defined a message called PerfMetrics that can contain multiple PerfMetric messages. Since no other changes than editing fields from being required to optional are done in the Protobuf file, the input data parser in fsmonitoring does not need to be altered to support input from perfswiper.

To summarize, when a packet with metric updates destined for fsmonitoring is crafted, it needs to consist of two ZeroMQ frames. The first frame must be 64-bit long and contain a header. This header should state that the second frame is a metric update. Expressing this is done by setting the value of FEP_MSG_METRICS (1) in the header. The second frame must contain a binary encoded PerfMetrics message with one or more PerfMetric messages inside.

### 3.11 Running Perfswiper as a Daemon

It is a requirement that perfswiper can run as a background process. The traditional UNIX way of launching and managing a background daemon is by using UNIX system V (SysV) initialization scripts[41]. The authors of a program is responsible for implementing the daemonization process and handle security aspects that are part of this process inside the application. A more modern way of launching daemons is to use systemd[42]. Systemd is a modern open source init and service management system for Linux. It guarantees that execution of the daemon process happens in a clean process context and it ensures that the environment block is sanitized. These are security features that had to be implemented in all individual daemons when using SysV. Systemd also makes sure that signal handlers are reset, and it cleans up all left-over file descriptors. Systemd has been adopted as the default init engine by most of the major Linux distributions, such as Red Hat, CentOS, Ubuntu, Debian, Arch, and openSUSE[43].

We see no reason in implementing the SysV approach for daemonizing since all of the major Linux distributions support systemd which is safer security wise and has an easier approach. The old-style SysV daemons have become
syntax = "proto2";
import "nodeinfo.proto";

message PerfMetric {
  enum LinkState {
    LINK_NO_CHANGE = 0;
    LINK_DOWN = 1;
    LINK_INIT = 2;
    LINK_ARMED = 3;
    LINK_ACTIVE = 4;
    LINK_ACT_DEFER = 5;
  }
  optional uint64 node_guid = 1;
  required uint32 port_num = 2;
  required uint64 timestamp_ms = 3;
  optional uint64 time_diff_ms = 5;
  optional NodeType node_type = 6;
  optional string node_name = 7;
  optional LinkState link_state = 8;
  optional uint32 node_lid = 9;
  
  optional PortErrorMetricV1 portErrorMetricV1 = 100;
  optional DataCounterMetricV1 dataCounterMetricV1 = 101;
}

message PortErrorMetricV1 {
  required uint64 symbol_err_cnt = 1;
  required uint64 link_err_recover = 2;
  required uint64 link_downed = 3;
  required uint64 rcv_err = 4;
  required uint64 rcv_rem_phys_err = 5;
  required uint64 rcv_switch_relay_err = 6;
  required uint64 xmit_discards = 7;
  required uint64 xmit_constraint_err = 8;
  required uint64 rcv_constraint_err = 9;
  required uint64 link_integrity = 10;
  required uint64 buffer_overrun = 11;
  required uint64 vl15_dropped = 12;
  required uint64 xmit_wait = 13;
}

message DataCounterMetricV1 {
  required uint64 xmit_data = 1;
  required uint64 rcv_data = 2;
  required uint64 xmit_pkts = 3;
  required uint64 rcv_pkts = 4;
  required uint64 unicast_xmit_pkts = 5;
  required uint64 unicast_rcv_pkts = 6;
  required uint64 multicast_xmit_pkts = 7;
  required uint64 multicast_rcv_pkts = 8;
  optional uint64 active_rate_bits_per_sec = 9;
}

message PerfMetrics {
  repeated PerfMetric perfs = 1;
}

Listing 3.7: Protobuf format used when transmitting metrics to fsmonitoring
obsolete. Freedesktop recommends that all new Linux services are written using the new-style daemonizing that systemd offers\[44\], making them simpler to supervise, implement, and control at runtime. The application systemctl is used to control a running service.

In order to build a systemd daemon we only need to implement a few simple tasks:

- **Signal handling** - Handle the SIGTERM\(^5\) POSIX signal.
- **Use correct exit codes from the program** - Follow the Linux Standard Base init defined exit codes.
- **Provide a .service file** - that encodes information about the process that systemd should handle.

### 3.11.1 Signal Handling

Perfswiper must handle one distinct POSIX signal to be systemd compliant. A POSIX signal is a limited form of inter-process communication that is used to notify processes or threads about events that occurred. Perfswiper must handle the SIGTERM signal. This signal should cause perfswiper to exit nicely and clean up after itself. Additional signals are not required to be handled, but it can be implemented and used together with the service file. One example of an optional signal could be SIGHUP. When this signal is processed, the application should reload the configuration file without restarting itself. When a signal is received, the OS interrupts the task, and the signal shall be handled instantly. To be able to process signals in POSIX systems a signal handler must be implemented. The `signal()` function defined in `signal.h` allows for signals to be caught. Calling `signal()` with the signal code (e.g. SIGTERM) as the first argument and a pointer to the function that should be called to handle the signal as the second argument will set up the signal handler for the process.

When perfswiper receives a signal after the handlers have been set up, the process will execute the appropriate handler. In Listing 3.8 we present the signal handler code of perfswiper. The given code is called when the SIGTERM signal is received. The signal handler cleans up memory allocated by ZeroMQ and the list of switches to monitor. Memory allocated by libibmad is also deallocated. The handler sets the globally defined volatile sig_atomic_t\(^6\) variable named got_signal to 1, and finally, the signal is reset to default behavior in case a new signal is received. Setting the got_signal variable to 1 will cause perfswiper to break out of the poller loop and exit

\(^5\)SIGTERM: Generic signal to terminate the program.

\(^6\)volatile sig_atomic_t: variable type used for signal handlers. Tells the compiler that while modifying this variable, it should not interrupt the task, leaving it half-complete.
void handle_signal() {
    SAFE_ZSOCK_DESTROY(publisher);
    SAFE_ZSOCK_DESTROY(cmd);
    SAFE_ZSOCK_DESTROY(topo_updates);
    SAFE_ZPOLLER_DESTROY(zpoller);
    lid_list_destroy(&lids);
    SAFE_MAD_RPC_CLOSE_PORT(srcport);

    // If an additional SIGTERM signal is received, default action.
    signal(SIGTERM, SIG_DFL);
    got_signal = 1;
}

Listing 3.8: SIGTERM handler in perfswiper

with the exit code EXIT_SUCCESS (0).

### 3.11.2 Actions and Exit Codes

According to the Linux Standard Base[45], perfswiper will need to support the following systemd actions: start, stop and restart. The configuration of these actions are done in the service file we discuss in the next section. For systemd to be able to control a process and to know if it was exited correctly or not, it is important that the program quits with an appropriate exit code. While writing perfswiper, we will try to be aware of all possible errors that could happen, and if fatal errors that need perfswiper to quit are discovered, perfswiper must exit with a general (or specific if applicable) error code. When a user is exiting perfswiper, we will make sure to clean up all allocated memory and close sockets before returning a 0 (which indicates success) to the shell.

### 3.11.3 .service File

The service file is an important part of the systemd service manager. For each system wide service that systemd is responsible for, a <unit_name>.service file needs to be stored in the /etc/systemd/system/ directory. This folder is where systemd looks for services to manage, and where it obtains information about the service. Systemd service files are used to describe a service and its behavior. A service is defined by configuring multiple variables under three different sections. The [Unit] section contains generic options that are not dependent on the
type of the unit. The next section is named [Service], and it holds service-specific directives. The last section is called [Install], it contains information about unit installation and startup and is used by systemctl.

The perfswiper service file (perfswiper.service) is given in Listing 3.9. Systemd service files support a large variety of settings. In the design of the service file, we have stuck to the basic and well-known features that would satisfy Fabriscale’s needs of daemonizing.

In the [Unit] section we have defined three statements. A short description of what this service does, and a pointer to where the user can find documentation about this service. Lastly, a statement that tells systemd that the network service should be running before starting perfswiper is given.

Following is the [Service] section. Here we specify where systemd will find the executable that should be launched. We explicitly state that the service is of type simple, even though this is the default type. This is done since we want to be verbose for users not familiar with systemd. In this section we could add a statement for ExecReload that sends an SIGHUP signal. This signal would cause perfswiper to reload its configuration, but this feature is not implemented in this first version. Another statement that were considered, was the Restart statement. By configuring this, systemd could restart the service automatically if it crashed. Since perfswiper depends on other services that might fail and cause perfswiper to crash, we do not want perfswiper to restart automatically if it fails. What caused the crash should be inspected and resolved before the service is started again.

In the last section ([Install]), we have specified that perfswiper is wanted by the multi-user systemd target. This means that perfswiper will be started up as soon as a machine has been booted up and the OS has entered the normal operation mode.

Listing 3.9: perfswiper.service file used by systemd
3.12 Debugging Functions and Asserting Correctness

The last specification that was presented at the beginning of this chapter was debugging functions. It is important that perfswiper is a robust software and that errors that occur are handled appropriately. To assert currentness in perfswiper, and to verify behavior, the function `assert()` from `assert.h` will be used during development. When a production binary is produced, the assert functions will not be executed to reduce computing overhead. A debugging print module will be used to print useful logs during development.

3.13 Summary

The purpose of this chapter was to archive the first goal of this thesis, to design and build a lightweight and efficient monitoring software for the IB interconnect. The designed software will be used as a plugin for the FFM developed by Fabriscale Technologies.

To be able to archive the first goal of this thesis, the following requirements of the monitoring software designed was set; Collect error and performance metrics from IB switches using low-level in-band communication. What metrics to collect should be configurable, and the software should integrate with existing modules in the FFM.

To fulfill these requirements, a set of specifications were given. The specifications were based on how perfswiper could archive the requirements and some additional specification related to robustness and debugging were given.

Each of the specifications of perfswiper was discussed in detail, and design choices that were made were justified.

The first specification that was given was how to store the metrics internally in perfswiper in-between subnet swipes. The C struct that was designed was included in this section.

It was presented how perfswiper would get knowledge about switches that are available on a subnet. A message protocol related to knowledge of switches was designed and explained. Further, this chapter also discussed what changes that had to be done in Fabriscale software to support our solution of finding switches.

The next section discussed the configuration file, and how it is parsed. How
perfswiper should handle user input errors in the configuration file were
given. The configuration file in its whole was included to show all settings
and what style was decided upon.

Next, the metric collection functions that use the libibmad library was
presented. The call flow was introduced, and detailed information about
the library was presented.

It was discussed how to push collected metrics to FFM, and our final design
by using ZeroMQ pipes were given.

The next section discussed daemonizing. We have designed a systemd
compliant software so that systemd can run perfswiper in the background
as a service.

The very last section of this chapter briefly presented the debugging methods
that we aim to use to reduce the number of bugs in the software.
Chapter 4

Experiments

In this chapter, we present the experiments we performed, what metrics we looked at and how we measured them. Parts of this chapter is dedicated to the first steps of the abstractions paradigm. It gives the models and some expectations of the performed experiments. We also present the different fabric topologies we used when performing experiments. We introduce the hardware and simulation tools we used, and we discuss the issues we encountered and the limitations we found. This chapter has five sections describing the experiments that were conducted, which, together with the technical background and the design of our metric collection software, gives enough information to replicate this study.

4.1 Background

In chapter 3 we described the design and implementation of a performance metric collector plugin for the FFM. The software we implemented will be used as a new module to the FFM and it will query performance counters on real hardware where FFM is deployed. It is important for the network administrators to know how equipment handles in-depth querying of performance counters and how it scales with the size of the subnet. We will, therefore, perform experiments on real hardware for the most accurate results. Unfortunately, we do not have a significant number of switches available that we can use to run large-scale experiments. We reached out to the owners of a HPC system with a network topology consisting of 37 Mellanox FDR switches. We asked if we could run our experiments on their fabric to gather large-scale results. The administrators were very interested in our research on the metric collection and said they expected our results to be very interesting. Unfortunately, due to their system being oversubscribed and heavily in use we could not be allocated a service window for running experiments. Running experiments while the system is in production could
Table 4.1: Switch specifications\cite{46}\cite{47}\cite{48}.

lead to instabilities in the network which they could not tolerate.

4.1.1 Hardware

Fabricscale Technologies only has a limited amount of IB switches in their lab, but we were able to borrow six switches from Simula Research Laboratory. Unfortunately, this meant that we had to run tests with switches of different models and from various manufacturers in our network topologies. On the positive side, all of the switches have the same model and version of the switching silicon (InfiniScale IV). In Table 4.1 we list all of the switches we managed to gather for testing and some of their specifications. Two of the switch models are unmanaged and produced by Mellanox. The last model is built by Sun Oracle and is a managed\footnote{Managed in this context means management more extensive than what the IBA provides using MAD.} switch. For fairness in our experiment, it is important that all of the switches have the same number of ports. 36 ports are the most common number found in these switches, and all of the switches we will utilize have 36 ports. Due to the limitation in the number of switches we have available, we must rely on simulation for large scale tests.

We give a presentation of the hardware and channel adapters inside our computing nodes in Table 4.2. These nodes will only be used for bandwidth generation and running the querying software.

4.1.2 Hardware Performance Limitations

The HCAs we used for conducting experiments in this thesis are Mellanox ConnectX 3 cards with dual FDR ports. As shown in Table 2.1, FDR-14 theoretically supports a data rate of 54.54 Gbit/s. The HCAs are connected to the compute node using Peripheral Component Interconnect Express (PCIe)\footnote{Lane: Two pairs of communication wires for duplex signaling.}. The compute nodes we used in our experiments has only PCIe version 1.1, and this version supports a transfer rate of 250 MB/s per lane\footnote{Lane: Two pairs of communication wires for duplex signaling.}.

In our setup, the HCAs are connected using an 8x PCIe 1.1 bus and are
### Table 4.2: Compute node specifications.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Model</td>
<td>AMD Opteron™ 2210</td>
</tr>
<tr>
<td>CPU Architecture</td>
<td>x86_64</td>
</tr>
<tr>
<td>CPU Speed</td>
<td>1 Ghz</td>
</tr>
<tr>
<td>CPU Cores</td>
<td>2</td>
</tr>
<tr>
<td>Threads per Core</td>
<td>1</td>
</tr>
<tr>
<td>CPU Cache</td>
<td>1 MiB</td>
</tr>
<tr>
<td>Memory Size</td>
<td>2 GiB</td>
</tr>
<tr>
<td>Memory Type</td>
<td>DDR2</td>
</tr>
<tr>
<td>Memory Speed</td>
<td>667 Mhz</td>
</tr>
<tr>
<td>HCA</td>
<td>Mellanox ConnectX 3</td>
</tr>
<tr>
<td>PCIe</td>
<td>PCIe 1.1</td>
</tr>
<tr>
<td>Operating System</td>
<td>CentOS 6</td>
</tr>
</tbody>
</table>

thus limited to a transfer rate of 2 GB/s (16 Gbit/s) between the HCA and the PCIe root complex\(^3\).

All of the switches we utilized in our experiments have a supported maximum link speed of QDR. As seen in Table 2.1, a 4x QDR link has a signaling rate of 40 Gbit/s. When running our HCAs and switch ports with 4x QDR, the PCIe 1.1 bus is limiting the bidirectional transfer rate between two hosts to close to 1500 MB/s (12 Gbit/s) in our max capacity tests. This test result is represented by the yellow line in Figure 4.1. In [50] the authors arespeculating that overhead caused by encapsulation of IB packets inside PCIe packets is limiting DDR over PCIe 1.1. We know that PCIe 1.1 uses 8b/10b encoding, so 20% of the channel is lost due to encoding at the physical layer. Like IB, PCIe also has a protocol set similar to what we see in IB that cause overhead in transmission. Besides, many motherboard chipsets\(^4\) today does not support more than 256-byte long transactions[51] even though the PCIe architecture supports up to 4096 bytes. Due to this limitation, a single IB frame must be encapsulated in multiple PCIe transactions causing extra overhead. In [51] the author explores factors of the PCIe architecture that can impact performance. He lists that in a PCIe environment, in addition to encoding, overhead is introduced by the transaction layer packet system, the link protocol, traffic overhead and the flow control. The HCAs we have are limited to a maximum of 128 bytes PCIe payload as shown in Listing 4.1. In [51] the author indicates that with 128-byte transactions the packet efficiency in PCIe is 86%.

We conclude that the overhead related to the PCIe protocols and the low

---

\(^3\)Root complex: The device that connects the PCIe switch to the CPU and memory subsystem.

\(^4\)Chipset: A set of components in a integrated circuit that manages the flow of data between the CPU, memory and peripherals.
PCIe packet size of 128 bytes is causing the missing 500 MB/s in measured application traffic in our test. Due to this limitation, we are not able to push data at 16 Gbit/s as specified by a 4x DDR link in a PCIe 1.1 environment. Still, we see better performance with DDR over PCIe 1.1 than the authors of [50] did in their tests.

In some of our experiments we need to be able to fully load the links in the fabric. To do that with the available hardware, we need to run all switch ports and HCAs at SDR with 4x link width. As shown in Table 2.1, SDR has a signaling rate of 2.5 Gbit/sec, thus a 4x SDR link will support up to 8 Gbit/s of application traffic. Forcing all ports to run in SDR mode will allow us to load the links using PCIe 1.1 entirely. As long as we can fully load the links in our experiments, it does not matter if the equipment is running on SDR, DDR or any of the other link speeds.

When we were running the HCAs and switch ports with 4x SDR links, we measured a transmit rate of 7.81 Gbit/s and a receive rate of 7.85 Gbit/s simultaneously as shown by the orange line in Figure 4.1. These results are very close to the theoretical max data rate of 8 Gbit/s which is represented by the blue line in Figure 4.1.

4.1.3 Traffic Generation

In the previous subsection, we discussed the maximum attainable data rate that we managed to push with the equipment at our disposal. For running the tests presented above and to gather the data shown in Figure 4.1 we were using an application called *ib_write_bw*. *ib_write_bw* is an OFED diagnostic utility that can be used for traffic generation in an IB environment. This tool is using the RDMA capability of the IBA to generate traffic. The tool establishes an RDMA connection between a server and a client. The client writes to the memory area that was allocated by the server using RDMA over the IB fabric. The data collected in Figure 4.1 was obtained running this utility with the `–a` parameter. This option is making the utility run multiple max bandwidth tests by using message sizes in the range 2 to $2^{23}$ bytes between the client and server.
For bidirectional traffic generation, which we did in the previous subsection, we used two pairs of this utility. One pair for each direction. `ib_write_bw` has an option for bi-directional testing, but the data that was outputted from this mode was aggregated, and we felt that we had a better data source and more control over the traffic generation when running one pair for each direction.

Some of our experiments require us to be able to control the bandwidth saturation of the network links. `ib_write_bw` does not have support for an inbuilt rate limiting or shaping. Thus, to control the transmit bandwidth we had to do this by controlling the size of the messages we sent. In Table 4.3 we show the observed message sizes and the corresponding bandwidth in MB/s and percent of maximum load in our system. We used these message sizes when doing traffic generation in our experiments.

### 4.1.4 Hardware Used for Simulations

Simulated runs were done on a desktop computer with the specification presented in Table 4.4.
<table>
<thead>
<tr>
<th>Network Load</th>
<th>Bidirectional Bandwidth</th>
<th>Message Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 %</td>
<td>958 MB/s</td>
<td>1048576 bytes</td>
</tr>
<tr>
<td>99 %</td>
<td>948 MB/s</td>
<td>600000 bytes</td>
</tr>
<tr>
<td>98 %</td>
<td>937 MB/s</td>
<td>60000 bytes</td>
</tr>
<tr>
<td>95 %</td>
<td>910 MB/s</td>
<td>1800 bytes</td>
</tr>
<tr>
<td>90 %</td>
<td>862 MB/s</td>
<td>785 bytes</td>
</tr>
<tr>
<td>80 %</td>
<td>686 MB/s</td>
<td>336 bytes</td>
</tr>
<tr>
<td>50 %</td>
<td>479 MB/s</td>
<td>230 bytes</td>
</tr>
</tbody>
</table>

Table 4.3: Network load with different message sizes used.

<table>
<thead>
<tr>
<th>CPU Model</th>
<th>Intel Core™ i5-4460</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Architecture</td>
<td>x86_64</td>
</tr>
<tr>
<td>CPU Speed</td>
<td>3.20 Ghz</td>
</tr>
<tr>
<td>CPU Cores</td>
<td>4</td>
</tr>
<tr>
<td>Threads per Core</td>
<td>1</td>
</tr>
<tr>
<td>CPU Cache</td>
<td>6 MiB</td>
</tr>
<tr>
<td>Memory Size</td>
<td>16 GiB</td>
</tr>
<tr>
<td>Memory Type</td>
<td>DDR3</td>
</tr>
<tr>
<td>Memory Speed</td>
<td>1333 Mhz</td>
</tr>
<tr>
<td>Operating System</td>
<td>Ubuntu 16.04</td>
</tr>
</tbody>
</table>

Table 4.4: Specifications of the machine running simulations.
4.1.5 Simulation Limitations

Due to the low amount of switches, we had to use a simulator for large-scale experiments. There exists no such tool that can simulate both the IB management model and network traffic, so we are not able to perform all of our experiments on both hardware and using a simulator. Developing such a tool is was considered to be too time consuming to include in this thesis.

In section 2.5 we presented IBSim, a simulation tool for simulating the IB management model. IBSim emulates the management model of a real IB subnet and devices connected to the fabric. IBSim is not a network emulator, and it has no concepts of links or network load. We investigated the IBSim source code to get some knowledge about how the simulation is handled and what conclusions we could draw from simulation results. We found that IBSim does not run any code if no MAD is inputted. From the source code investigation, we found that IBSim is a tool designed and developed for function testing of IB management applications. We can thus not argue that IBSim is a good simulator for us to use to measure the performance of the PMA, but it is still a helpful tool for us to verify that perfswiper also can handle larger topologies than what we can test using real hardware.

In the next section, when we present our test metrics, we will dig deeper into the issue with IBSim as the simulator for our experiments.

4.2 Test Metrics

Measuring the performance of the PMA inside an IB switch is not trivial. We are not able to see the CPU load or memory utilization inside the switches in general, nor on individual processes running. Not even the two managed switches we have available can output this information according to their manual. A direct performance sample is not possible to collect directly from the switch. These high-performance switches are built with a focus on fast switching of data between ports using application-specific integrated circuits (ASICs)\(^5\) and have a smaller general-purpose CPU for the management of the chassis. The implementation of the PMA and how tasks are prioritized inside the chassis management of the switch is not known to us. We do not have any knowledge about exactly how incoming packets are handled at the very lowest level of the chassis. How much of the incoming MADs are processed by ASICs? How much of it must the CPU handle? How is scheduling done? These are some questions we have not managed to answer by reading the IB specification. These details are very specific and are up to the manufacturers to design and implement in their equipment as they

---

\(^5\)ASIC: An integrated hardware circuit optimized for a particular usage.
Since we are unable to extract performance information directly from switches, we had to figure out another way of measuring it. We found that measuring the time elapsed between we issue a performance counter request to a switch until we receive an answer could be used as an indicator of performance. To get the most accurate results we must control the environment around the timing. We can control the network load and management traffic in the fabric. This control makes sure we can minimize the amount of work a switch has to do. We know that all of the switches we have has a port-to-port latency of 100 nanoseconds or less as shown in Table 4.1. We also know that the IBA defines a lossless network, so by monitoring the network for increased latency and drops, we can check if the increased response time from the switch relates to any discards or waits caused by the network.

In a controlled environment, an increased response time while we measure, is likely to be caused by how the switch processes a request it receives. What exactly is causing the delay inside the switch is unknown for us, but it is likely to come from an overloaded entity. Nevertheless, the increased response time, in an isolated scenario, will provide the best possible intelligence on how well the switch is performing.

### 4.2.1 Timing of the Response

Since we found that a response time measurement is the only way of collecting data about the PMA, it is important to know exactly what we are timing. Many factors will influence the result. The best approach for the most accurate results would have been to time stamp packets when they are sent and check when received. We could then easily subtract time spent on the network and the result we would have then would be the time it took for a switch to process the MAD. Unfortunately, there is no support for adding time stamps in MADs with the current OFED libraries, and even if we could time stamp on the client side, the switches would not be able to time stamp when they transmit the response. And ultimately, even if they could, we would have a hard time of keeping the two clocks in sync.

We must, therefore, rely on timers on the host running the querying software, and measure before and after the RPC is issued. In all of our experiments, we will be using the function `clock_gettime()` for measuring the elapsed time. In [52], `clock_gettime()` was found to have lower time penalty than `gettimeofday()` using a single thread. `clock_gettime()` is offered in Linux through the POSIX library, and it provides an interface to multiple system clocks. The resolution of the result from this function is in nanoseconds. The choice of any of the provided clocks are specified as the first argument to the call, and current time since a start point is
returned to the caller in a timespec struct given as the second argument. The starting point of the timer differs between the different clocks offered with clock_gettime().

We have decided to use two different clocks for timing in our experiments. CLOCK_PROCESS_CPUTIME_ID for calculating CPU time\(^6\) used and CLOCK_MONOTONIC for calculating WALL time\(^7\). With clock_gettime() it is also possible to use CLOCK_REALTIME for WALL time measurement, but this clock is affected by time changes by for instance Network Time Protocol (NTP) or changes done by the OS. CLOCK_MONOTONIC, on the other hand, gives an absolute wall-clock time since a fixed point in the past. It is likely that the querying process will be preempted while waiting for the RPC response from the PMA, and when this happens the CPU timer will be paused when the process is removed from the CPU. How the process is scheduled, and how fast the interrupt when the RPC response arrives is handled, will most likely affect the result of this timer. The CPU counter only counts time used while the process is active in the CPU. Therefore, when we discuss the results of our experiments, we will mostly rely on the WALL time used. By also sampling time used in the CPU means we can easily see if a spike in the WALL time can be related to OS load. Load on the compute node could cause the querying process to be slower.

A start time is sampled before executing the get_perf_counters() call which we introduced in chapter 3. This call is responsible for querying both 32-bit and 64-bit counters of a switch, and it parses the response from switches. An additional time sample is taken after it has been completed. We have avoided code inside the critical sections that are time-consuming, such as debugging prints to the console or writing to files. The elapsed time for the experiment will be calculated from \(\text{struct timespec start} - \text{struct timespec end}\).

\section{4.2.2 OS Scheduling}

Since we are relying on timers to measure the performance of the PMA, it is important that we know what we are timing, and what affects the timing. How the OS is scheduling our measurement software will affect the results we get. To eliminate some scheduling factors, we will run querying software on a separate compute node from the bandwidth generation nodes. Running it on a separate node will give us more control over how the CPU distributes its resources. We will also make sure that the compute nodes are not running any tasks that could be avoided.

We will use the Linux application chrt\(^8\) to give our querying software

---

\(^6\)CPU time: Time that a process has been active in the CPU.

\(^7\)WALL time: Time spent measured with a wall-clock.

\(^8\)chrt: An application able to manipulate the real-time attributes of a process.
"real-time" resources from the operating system during execution. Linux
does not support proper real-time scheduling, but we can give the querying
software the highest possible priority. The compute nodes we have available
for running experiments only have two CPU cores with one thread. It is
inescapable that our measuring software will be preempted.

The scheduling policy `SCHED_FIFO` will be used together with the highest
priority level (99) of this policy. Tasks scheduled with this policy will only
be preempted\(^9\) if it waits for any resource, such as input data, any thread
related synchronization mechanisms temporarily stop it or if it volunteers to
give up CPU by calling `sleep()` or `yield()`. The only exception of kernel
threads, which have a higher priority than it is possible to set with `chrt`.
In our case, the task will have to wait for an I/O interrupt when the RPC
returns with a MAD on the machines IB device. When this I/O interrupt
is happening, our querying software will be scheduled immediately.

The scheduling policy will become even more important when we run
experiments using a simulator. It is important that both the querying
software and the management simulator (IBSim) are running with the
highest priority and is not being interrupted unnecessarily.

### 4.2.3 Data Collection

To perform our measurements, we use a slightly modified version of the FFM
plugin described in chapter 3. We start the timing inside a function named
`timed_swipe()` containing a loop for granularity of the results. Inside the
loop, the libibmad function `pma_query_via()` is called for each switch and
port pair. Our response time measurement does, therefore, include time
spending executing calls inside the libibmad library. `pma_query_via()` is an
RPC function and is limited to query one LID and one port at the time, and
this function is blocking until results are received from the fabric. In most of
our experiments, the software will execute 50 queries and use an average as
the plotted value for better confidence in the results. The gathered response
time data will, therefore, be the time it took to poll metrics from all 36 ports
of the switch.

### 4.2.4 Data Processing and Plotting

The response time values outputted by our querying software while running
experiments is used as input to a script that calculates the average response
time. These average times are saved in `.dat` files. We are going to use

\(^{9}\)Preempt: A task forced to stop by the OS so another task can be allocated CPU
resources.
Gnuplot[53] to make vector graphics from the collected data. The generated plots will be presented and discussed in chapter 5.

4.3 Experiment Design

In this thesis, several experiments will be designed and conducted using a slightly modified version of the metric collector we described in chapter 3. Some minor support functions such as functions for timing of queries, and data structures to facilitate storage of samples, must be created. This section is dedicated to the first stages of the abstractions paradigm. We design multiple models which we refer to as experiments later on. Some of the experiments we have designed in this thesis will be carried out both on hardware and by means of a simulator. Some experiments are only done using IBSim due to the lack of hardware to support large-scale topologies.

To achieve the second goal of this thesis, examining how in-depth performance monitoring of an IB subnet affects the network, we will inspect five scenarios. First, in Base Experiment (Section 4.3.1) we will investigate if we can find any differences in how the switches we presented in Table 4.1 handles being queried for performance counters. In Experiment A (Section 4.3.2) we want to examine how response time of monitoring grows when the requests and responses have to travel through multiple switches. We will also introduce load on the network to see how this affects the response time. Next, in Experiment B (Section 4.3.3) we will focus on the pattern used for querying. We want to examine what could be the most efficient pattern to use for querying. This scenario will determine which pattern to use as default in the FFM plugin. Experiment C (Section 4.3.4) is trying to investigate how the PMA and network handle being flooded with queries over time. In the last couple of experiments, which are presented in Experiment D (Section 4.3.5), we are investigating how the metric collector is performing in a real life monitoring situation. We will build the biggest possible fabric with the switches we have available and also run a simulated test with a large cluster with 2,044 IB ports. Combined, the results of these five main sets of experiments should give the reader insight into how performance monitoring affects the network and how the metric collection software designed in chapter 3 performs.

4.3.1 Base Experiment

As a starting point, and since we have switches produced by multiple manufacturers, we wanted to get some insight before designing and conducting our experiments. To accomplish this, we carried out a base experiment. In this experiment, we wanted to investigate if there were any
noticeable differences between the different switches. Specifically, we wanted to know if the switches behaved differently in regards to being performance monitored. Since we have equipment from various manufacturers, knowing how these performs compared to each other is necessary. To determine if there are any differences among them, we directly connected a compute node into one of each of the three different switch types we have. When this was set up, we ran the querying software on each compute node and did a timed query of all performance counters of all ports on the switches. Each device was queried repeatedly without any added delay between queries for 50 times. We calculated an average of these 50 and used the average to achieve better confidence in the data. This test provided us with some insight on the differences between the switch models regarding performance.

The result from the base experiment is given in Figure 4.2. In the data source used for this plot, the CPU time has been subtracted from the WALL time data to make the differences more visible. It is not the local CPU time spent we are interested in measuring, but the response from the switch. The error bars included in this plot is the standard error calculated from all the 50 samples. From this analysis, we can see that there are some tiny variations between them. But, we do see that the Mellanox produced MTS 3600 switch has around one millisecond higher response time than the two others. This increased response time could imply that the MTS 3600 switch is using more time on processing received PerfMgt MADs than the other two switches that we have. It is also worth noticing that the 50 queries of the SUN DCS switch resulted in a higher standard error than the two others. The difference in the CPU time must be related to how the OS scheduled the process.

Figure 4.2: Measured response time from the PMA on the different switches.
4.3.2 Experiment A: Chained Topology

This section introduces the first experiments that we are going to perform on our equipment that is directly related to our research questions. In the experiments in A, we are interested in seeing how the response time measured in CPU and WALL time is increasing with regards to the number of switches the MADs are passing through on its way to the designated switch.

In the experiment model it will also be introduced different levels of load on the network to stress the switches. During the experiments, we will monitor the PortXmitWait, PortXmitDiscards and VL15Dropped performance counters on all switches. If any packet drops are seen, or if packets were delayed we would know. The VL15Dropped counter will indicate if general subnet management traffic is affected by our experiments. The experiment model will use a chained topology to achieve the longest possible distance for a MAD to travel with the set of switches we have available. The designed experiments in this section will provide us with information about how switches handle MADs and how traffic in the network affect MADs. They will also give insight into how much time a query of performance counters increases with regards to the travel distance in the network. Where possible, we will run the following experiments on a simulator as well. As discussed in Section 4.1.5 we are not able to simulate scenarios with network load.

In this section, we will perform three experiments. In all of these three, we will use all 10 IB switches listed in Table 4.1, and four of the compute nodes listed in Table 4.2. The switches and compute nodes will be connected to the fabric with copper wires and we will force the link width and speed to 4x SDR. For each of the three experiments, we will provide an illustration of the topology, the nodes inside it and explain their responsibilities.

Experiment A1: No Network Load

In this very first experiment, we will perform the experiments on both hardware and use IBSim for simulated runs. The switches and compute nodes are connected as shown in Figure 4.3. The querying software will be run on c-0-0, and there will be no other traffic on the network other than the MADs we are generating while querying. As described in Section 4.2.3, we will take 50 samples of the time elapsed between we issue the RPC until we get the response from the switch. The RPC function is limited to query one LID and one port at a time. The querying pattern we will use is as follows. The first port of the first switch, the second port of the first switch ... the 36th port of the tenth switch.
In this next experiment, we will set up the fabric as shown in Figure 4.4. The compute nodes c-0-1 and c-0-2 will run the traffic generation software described in Section 4.1.3. The querying software will be run on a separate node, c-0-0. The querying part of this experiment will be done the same way as given in Experiment A1. In addition to the changes in the fabric, this time we will introduce artificial network load on one of the network links and do multiple samples. The link between switch 01 and 02 will be loaded with these levels 50%, 80%, 90%, 95%, 98%, 99% and 100%. For each of the load levels we mentioned above, we will do 50 response time samples.
Experiment A3: Load on all links

In this third experiment in A, the number of links with network load will be increased. We will use the same load levels as we described in Experiment A2. The fabric will be set up similar to A2, but one of the compute nodes used for traffic generation will be moved to the end of the chain. We will use the same load levels as in the previous experiment. This placement will let us introduce network load on all links between all of the switches. Figure 4.5 gives a presentation of the fabric topology that will be used in this experiment.

Figure 4.5: Experiment A3: Subnet topology with switches and compute nodes.

4.3.3 Experiment B: Querying Method

Since the RPC issued by our querying software is limited to asking one switch and one port at a time, we want to investigate which pattern is the most effective, and we intend to explore if the pattern makes any difference on the response times from the PMA. So far, the experiments we have designed will be using one particular querying pattern. In Experiment A1 (section 4.3.2) we described the pattern like the following: The first port of the first switch, the second port of the first switch ... the 36th port of the tenth switch. In the experiments presented in this section, we will introduce two new querying patterns. The primary purpose of these experiments is to see if the querying pattern impacts the response time from the PMA, and investigate which one of these are spending more time on the CPU. We hope to understand which pattern is the best so we can use it as the default pattern for the FFM plugin.

In this section, we will have three experiments. In all of them, we will use the same chained topology as we used in Experiment A1 (section 4.3.2). The querying software will be run on c-0-0. The only difference between these two experiments is the pattern of which we query the performance counters. We will not run this experiment using IBSim due to the nature of how IBSim is running simulations. The pattern has no impact on what
code is run in IBSim. The results from a simulated run of this experiment would then not give us any useful information.

Experiment B1: Port Sequentially

In this first experiment, we will be using the same querying pattern as we designed in the previous set of experiments. The main purpose of this pattern is to completely query all ports of one switch before moving to the next switch. The software will start with switch #1 and port #1, then it will do switch #1 and port #2, and last it will do the last and tenth switch, and port #36. This experiment is identical to Experiment A1, but we are now going to compare its results to the results of the other patterns.

Experiment B2: Switch Sequentially

This next experiment will try out a new pattern where we focus on the port numbers, and moving on to the next switch after one port has been queried. The pattern the metric collection software will use is as follows: The first switch and port #1, next it will ask switch #2 and port #1. This continues until we hit the tenth and last switch and port #36.

Experiment B3: Random Pattern

So far in Experiment B, we have designed experiments that look at two different patterns for collecting metrics. In this next experiment, we will use a random pattern for querying, but still be sure that all switches and ports are queried within one subnet swipe. Both the switch and port sequence will be conducted randomly. The software will pick a random switch from the list of available switches, and before collecting metrics, it will randomly shuffle all the port numbers. We use the \texttt{rand()} function seeded with the current time to generate the random numbers. It exists multiple versions of \texttt{rand()}. We will use the one that is part of \texttt{stdlib.h} and seed it with the current time.

4.3.4 Experiment C: Flooding

The purpose of the experiments in this section is to try to push the switches and their PMAs to their limits by flooding the switches with requests as fast as possible. These flooding experiments should give the reader an answer to parts of the second research question related to the metric collection in this thesis. We hope to find out how well switches handles being queried for its
counters, and how often it is possible to do the query without overloading them and interrupt application traffic.

In the previous experiment designs, we have planned to query the same switch and port 50 times to get better confidence on the collected data. We do not expect this to be enough to burden the switches, so in this section, we plan to increase the number of queries per port. Choosing the number of queries to be sent beforehand, and predicting the behavior is impossible. We will have to pick a starting point and after seeing the result, try to increase the number of queries up to a reasonable amount. As a foundation, we will start by timing one port sample, then go for 50 samples since that is what we have used in the previous experiments. We will continue to increase the number up to a reasonable amount based on the results we are seeing.

In this section, we will perform two experiments. None of the experiments require network load so we can conduct them on both hardware and by using IBSim. All 10 IB switches from Table 4.1 will be used, and one of the compute nodes listed in Table 4.2.

**Experiment C1: One switch**

In this first experiment, we will only investigate the behavior of one directly connected switch. The switch will be directly connected to a compute node running the querying software. A sleep of two seconds is added between each level of load and functions as a cool-down period.

**Experiment C2: Chained topology**

In this next experiment, we will use the same chained topology as we did in Experiment A1 (section 4.3.2), with the same placement of the compute node. The compute node will run multiple instances of the querying software, one instance for each switch. This will make sure that the performance metric collector does not limit the flooding of queries.

**4.3.5 Experiment D: Testing Perfswiper**

This last section is dedicated to the last two experiments in this thesis. These experiments aim to further answer the second research question of the metric collection in this thesis: How often can we query a switch for its performance counters? Both of the two experiments in this section will be executed on a network topology named fat-tree[54]. The fat-tree topology is one of the most used network topologies in HPC systems, and it is dominating in
systems where IB is deployed[55]. No network load will be introduced in this experiment.

The first experiment in this section is dedicated to producing a real world test of perfswiper in a fat-tree topology, and the second is dedicated to large scale simulation of a fat-tree topology. The results of these two experiments should give the reader an indication to how often it is possible to query a switch for its counters in a fat-tree topology.

Since we at this point still have no knowledge of which querying pattern is best, Experiment D will use the same pattern which was used in Experiment A. The switch order is the order of which the OFED ibswitches observes the switches in. The closest switch is the one that will be queried first.

**Experiment D1: Fat-tree Topology**

In this experiment, we will build the biggest possible fat-tree with the switches we have available. We will run the querying software on a host connected to one of the edge switches. The network topology and the placement of the querying node in this experiment is given in Figure 4.6.

Since a fat-tree topology is the likely network configuration where the FFM will be deployed, it will be of great value to see how it performs in such an environment. This specific experiment would in a real life FFM situation answer the question to how often we can poll for performance metrics.

![Figure 4.6: Experiment D1: Subnet topology with switches and a compute node.](image)
 Experiment D2: Large Scale Simulation

This last experiment in this thesis will only be conducted using the IB management model simulator IBSim. It will use a two-level fat-tree topology such as the one used in Experiment D1, only this time it will be larger. The purpose of this experiment is to test the metric collection software designed in chapter 3 and see how it behaves in a large cluster consisting of more switches than we have available to test in our hardware lab. We hope to verify that our design also works in larger topologies.

Previously in this chapter, we mentioned that we reached out to the owners of a fabric consisting of 37 switches, but that we were unable to run tests on their cluster. The owner gave us an ibnetdiscover output of their fabric which we can use in IBSim to simulate their network. The network is a two-level fat-tree and has a total of 2044 FDR IB ports.

In this experiment we will poll each switch and port repeatedly 50 times and calculate an average response time. The results will be scaled to a real life average approximation and estimate how long the swipe would take on a real hardware subnet. The estimate will hopefully give us some insight on how low swipe interval perfswiper realistically could support in this network configuration.

4.4 Summary

The purpose of this chapter was to achieve the second goal of this thesis. To be able to answer the second group of research questions given in section 1.2 this chapter designed multiple experiments. The background and the design of the experiments in this thesis were presented. The hardware we used to perform experiments was presented, and the chapter explained what limitations we found and what test metrics we used in the experiments. Furthermore, it described how data was going to be collected and how we would plot it.

Next, this chapter presented the five main experiment sets we conducted in this thesis. The first experiment was designed to give insight on how the three different switch models that we had performed. The second experiment aimed towards getting a view of how switches handle MADs and how management traffic is affected by network load. The third experiment was designed so we could learn what querying pattern is the most optimal for the FFM plugin designed in chapter 3. The next experiment had the purpose of trying to push the switches and their PMA to their limits by flooding the switch with MADs asking for performance counters. The fifth and last experiment was designed to give insight on how the metric collection software designed in chapter 3 handles larger scale topologies than what we
were able to test with hardware.
Chapter 5

Results and Discussion

In this chapter, we present the results of the experiments that were described in chapter 4. It discusses and interprets the observations that were made. This chapter should give the reader an idea of how well the switches handle being queried, and how querying impacts the network. It also shows how well the metric collection software designed in chapter 3 works. In the end of this chapter, we discuss the overhead introduced by the monitoring software.

5.1 Base Experiment

Before interpreting the results, we want to quickly reiterate about the base experiment, that was conducted before designing the experiments in the previous chapter. In Figure 4.2, presented in the previous chapter, we compared the response times between the three different switches that were used when conducting experiments. Some differences were noticed among them, and the largest deviation we found was just below one millisecond. The MTS3600 has an average response time (without CPU time) of 2.4 milliseconds, the IS5025, and the Sun Oracle DSC has an average response time (without CPU time) of 1.5 milliseconds and 1.4 milliseconds. These differences are worth noting when interpreting the results presented in this chapter.

5.2 Experiment A

In Experiment A we were interested in seeing how response times are growing with regards to the number of switches the messages have to pass through. We were also investigating if and how network traffic impacts the response times.
5.2.1 Results A1

In Experiment A1, we queried ten switches in a chained topology with no network load for its performance counters. Four plots have been created to visually display the data collected from this experiment. Figure 5.1a and Figure 5.1b present the response times that were logged while querying the switches in the chain. Plot (a) shows the results of experiments carried out on hardware, and plot (b) show the results from simulations. Each data point in these two graphs is an average of 50 queries. In Figure 5.2a and Figure 5.2b, each of these 50 queries is displayed for the first switch in the chain.

In the test results from the hardware run, it looks like there is a slight increase in the response time between the first and the last switch. However, after removing the time perfswiper spent on the CPU, it shows that queries of the farthest switch responded faster than the one nearest to the measuring software. According to the specifications of the switches we have, one switch operation should take less than 100 nanoseconds. We expected that we would be able to see this increase between the switches in the start and the end of the chain. Our results show that the farthest switch in the chain (switch 10), had a response time (not including CPU time) of 2.09 milliseconds, from the closest switch a response was received after 2.34 milliseconds. From the plots, we observe that switch 05 and 06 in the chain have an unexplainatory lower response time than the others. These Sun Oracle switches performed the same as the Mellanox IS5025 switches in the base experiment.

Results from the simulation are presented in Figure 5.1b. It shows that the response times are quickly rising on switch 02 and 03 in the chain. After this, the response times decrease again to about the same level as the response from the first switch. After interpreting the simulated result, it is clear that IBSim is not simulating a real network. It does not simulate the port-to-port latency, and the measured results show a more steady trend than the same tests run on hardware. The average of 50 queries of the first switch shows a response time of 223,494 nanoseconds (not including CPU time), the highest response time is from switch number 03, with 427,314 nanoseconds. The reply from the last switch returned after 192,958 nanoseconds.

In Figure 5.2a and Figure 5.2b, we compare the individual 50 responses from hardware and simulated runs that were used to calculate the average response times displayed in the previous plots. We only looked at the results from the first switch. The standard error in the CPU and WALL time data source for 50 hardware samples are 0.09 milliseconds and 0.15 milliseconds. In simulated runs, the standard error is 13,774 nanoseconds and 24,986
nanoseconds. The plots show that the response times from simulated runs have less deviation in a set of 50 samples than runs on hardware. The simulated results show that IBSim does not add any latency to simulate a MAD in network transit. The WALL time (which includes CPU time in this plot) is almost exactly double the CPU time measured in every run.

Based on all of these results, and with the standard error that we saw in the base experiment in mind, it is clear that we measure with a very high resolution. The port-to-port latency that should be lower than 100 nanoseconds is insignificant in this experiment, and we are not able to see any traces of it. Differences in how the OS scheduled our monitoring software is probably causing the diversity that we see in our data. Specifically how the OS handles the blocking RPC and interrupts when the response is received. Having data on these components would be interesting, but it is not available to us.

Since we do not see any distinct growth in response time while sending MADs through multiple switches, we can conclude that with zero network load, the travel distance of MADs in a typical HPC network topology is not of substance. The switches we have tested show no impact on being queried for its performance counters 50 times in a row. From this experiment, we can also conclude that IBSim is not simulating a real IB subnet.

5.2.2 Results A2

In this section, we present the results from Experiment A2. This experiment was roughly identical to Experiment A1, but in stead of collecting data from a network without any load, multiple levels of network traffic were injected into the fabric. The load was placed between the first two switches in the chained topology. Figure 5.3 shows the variance in the time the querying software spent on the CPU and Figure 5.4 shows the WALL time measured with CPU time removed from the data source.
Figure 5.2: Experiment A1: The 50 individual queries that compose the average of the first switch in the chain.

The lowest recorded response time found in the CPU time data is about 2.2 millisecond, and the largest is just above 2.9 milliseconds. The differences between them are tiny and are most likely due to the variance in how the task was scheduled on the CPU. The plot shows a pattern where the increased network load leads to a lower amount of time the process spent on the CPU. We do not know what exactly is causing lowered processing time of the metric collection software when there is more traffic in the network, but after looking the plot of WALL time measurements, we have one theory. We suspect that when there is a low load level on the network, the process running the metric collection software is idle-waiting (and counting CPU time) when it is blocked waiting for the RPC reply. But when the network load increases above 80%, and the response times also increases we suspect that the process is unscheduled by the OS. Removing the process from the CPU causes the CPU time timer to stop.

From these results, we can see why the time spent on the CPU should be removed from our WALL time plots. As suspected, the CPU time varies between our runs, removing it provide us with results that are focusing on the time spent by MADs and the PMA.

In the plots of elapsed WALL time (Figure 5.4), the time consumed on the CPU is removed from the data source. Thus, it only presents the actual time that the MADs were out in the network and being processed by switches. The graph clearly shows a rapid growth between the first and second switch when the link saturation reaches somewhere between 80% and 90% load. Increasing the link saturation above 90% has no significant effect on the response times. The response times flatten out and only minor changes are seen after this point. With a load level of 90%, we see an increase of approximately 2.4 milliseconds when the MAD passes through the link with network load compared to the results from 80% load. This increase is close to a doubling of the response time. At some point between these two levels, the MADs are slowed down significantly. No PortXmitWait, PortXmitDiscards, or VL15Dropped have been logged during this test. We
believe that the increased response time is caused by the rising demand for buffering in switch ports. Increased amount of data in buffers will lead to longer storage time before MADs are transmitted out on a port. Since nothing is logged in PortXmitWait, we know that congestion on the receiver is not causing a delayed sending.

Further into the WALL time plot, when the MADs is passing through switch 05 and 06 in the chain, a slight drop is recorded for all the load levels. The largest drop is at 95% load and is approximately 1.1 milliseconds. We have not been able to explain what is causing the declines in response time when MADs are passing through these two switches, but there is a visible pattern. These two switches are the Sun Oracle DCS switches which performed the same as the Mellanox IS5025 switches when we looked for differences in the base experiment.

Figure 5.5 is presenting how the response time from the tenth switch developed during testing with multiple levels of network load. CPU time has been removed from the data source. The plot shows the rapid increase in response time that occurs with a load between 80% and 90%. It also shows how the response time is stabilizing at around 4.75 milliseconds with higher load levels.

From the plots presented in this section, we can observe that fabric monitoring in a subnet is affected by the amount of load in the network. MADs transmitted to gather metrics are delayed. By interpreting the test results from Experiment A2, we can conclude that one single IB link with a network load of somewhere between 80% and 90% is causing a close to doubling in the response times from the PMA. The increase is not present when querying the directly connected switch. This means that the increased load level is not causing an overloaded CPU or another entity that is responsible for handling the MADs, but it is likely due to a longer period of which the MADs sits in buffers.
Figure 5.3: Experiment A2: Measured CPU time.

Figure 5.4: Experiment A2: Measured WALL response time.
5.2.3 Results A3

This next section is dedicated to present the results of the third and last part of Experiment A. This experiment is identical to A1 and A2 except for one change. The number of links in the chained topology with load was expanded. In A3, all of the horizontal links between switches was injected with network traffic. Figure 5.6 shows the time the querying software spent on the CPU, and Figure 5.7 shows the WALL time measured with the CPU time removed from the data source.

The CPU time results are similar to what we saw in the previous experiment, no significant changes. But we do see a new recorded low of 1.5 milliseconds. The highest value is the same as before at 2.9 milliseconds. Interestingly, both the lowest and highest value is from the run with 0% network load. The querying software and the traffic generation were run on separate hosts, so the variance is not related to the traffic generation. Again, as we discussed in the previous section (Section 5.2.2), the CPU time is not relevant to the performance of the PMA, and the variances we see here must be linked to task scheduling in the OS and has little relevance to the responses from switches.

When we take a look at the results from the WALL time after removing the time spent on the CPU in Figure 5.7, we do see an increase in the response time compared to the previous experiment. Similar to the results
from the previous experiment, we see a rapid increment somewhere between 80% and 90% load at the second switch in the chain. But, it is not quite as accelerated in this experiment. When a MAD passes through one switch with 90% saturation, we see a lower response time in this experiment than before. When passing two switches at 90%, the result is roughly the same as in the previous experiment, but after this, the response time is increasing further. As in the last experiment, increasing the saturation above 90% has minimal effects. The highest response time logged in this experiment was 8.4 milliseconds and originates from when passing switch number eight at 90% load. This is 3.7 milliseconds more than the same load level and the same number of switches the MAD passed through in A2. No PortXmitWait, PortXmitDiscards, or VL15Dropped were logged during the run. Just as in Experiment A2, we observe that switch 05 and 06 in the chain have an unexplanatory lower response time than the others. These Sun Oracle switches performed the same as the Mellanox IS5025 switches in the base experiment, but from our results now, it looks like they handle MADs faster than the other switches when links are loaded.

Figure 5.8 shows how the response times grew when the tenth switch was queried. The pattern in this plot is similar to the corresponding plot from Experiment A2, but the increase between 80% and 90% in A3 is significantly higher. Load levels above 90% are causing the response times to drop down to just above five milliseconds which is almost the same response time we saw on the same switch in Experiment A2.

By looking at the test results in Experiment A3, we can see that expanding the number of links with network traffic causes an increase in the response time from the switches even more than we saw in A2. In the results from A3, we can see that the number of switches that the MADs are passing through is becoming more prominent than in A1. The highest increase in response time is observed with a load level between 80% and 90% in this experiment just as in A2.
Figure 5.6: Experiment A3: Measured CPU time.

Figure 5.7: Experiment A3: Measured WALL response times.
Figure 5.8: Experiment A3: Response times based on the degree of network load.

5.2.4 Experiment A Summary

After analyzing the results of Experiment A, we can conclude that the amount of network load in the IB fabric does affect how fast our measurement software will receive the response MADs. It is evident that a load level of somewhere between 80% and 90% on a single link doubles the time of reply from the neighboring switch. When the number of network links with network load is expanded to ten, the increase in the response time between the directly connected switch and the farthest is just above three times. We have shown that the increases in response times are not caused by an overloaded switch CPU, or entity responsible for handling MADs. A higher network load is probably causing MADs to sit in switch buffers for a longer period, thus increasing the response times.

We did also observe that the CPU time can vary with as much as one millisecond, but we conceded that the CPU time is not very compelling in this case as it is most likely due to the process being differently scheduled by the OS when the process is waiting for the RPC reply.
5.3 Experiment B

In Experiment B we wanted to investigate if the pattern of switch and port pairs to query impacted the response times from the PMA. We wanted to find the most effective pattern so it could be used as default in perfswiper. Furthermore, we wanted to examine if the pattern impacted the CPU time. This experiment was conducted using one single switch, the Mellanox IS5025.

In this section, we have combined the results from all three experiments that were part of B in two plots, one plot for WALL time, and one plot for CPU time. Figure 5.9 presents the WALL time data, and Figure 5.10 presents the CPU time data.

5.3.1 WALL Time Results

In the plots presenting the WALL time observed in the experiments in B, the time spent on the CPU has been removed. We are only interested in the actual response times, and as we have discussed earlier in this thesis, the variance in the CPU time is distorting the results if it is included.

In the WALL time results, a difference of around one millisecond is observed in the measurements. The lowest recorded response time is 2.1 milliseconds, and the highest logged response time is 3.1 milliseconds. It is not clear which of the three querying patterns is the best, none of them stand out.

5.3.2 CPU Time Results

The CPU time results on the other hand, unquestionably show significant differences between the patterns. What is interesting here is that the collected data suggest that pattern used in B1 is the best, the same pattern that we used in A1. Here in this experiment, the pattern used in B1 is giving us a steady average of about 1.55 milliseconds, but in A1, the very same pattern gave us an average of around 2.5 milliseconds. The OS of the compute nodes was reinstalled by Fabriscale between we performed Experiment A and B, and we suspect that some local changes can have affected the scheduling.

We were skeptical of this result and went back and inspected the source code used for timing. The inspection reminded us that the amount of CPU time spent in perfswiper when executing the different pattern is not measured. The timing wraps the `get_perf_counters()` and thus only calculates the time used to call libibmad and the decoding of the reply. The CPU time
results in this experiment can therefore not be used to conclude which pattern is the most efficient regarding CPU usage.

Due to the way the CPU time was measured in this experiment, all three experiments should use the same amount of CPU resources. All these three experiments were run as separated processes and with two seconds sleep between each process was executed. Since we observe significant differences in these three runs, we must assume that each execution was scheduled differently by the OS. In retrospect, we should have run all three configurations from the same process, and also collected data on the time it took to execute the different patterns in perfswiper.

\[ \text{Figure 5.9: Experiment B: WALL time results.} \]

### 5.3.3 Experiment B Summary

From Experiment B we can conclude that the pattern has no noticeable influence on the response times from the switches. We must also conclude that the CPU time results can not be used to determine which pattern is the most efficient regarding CPU usage.

### 5.4 Experiment C

Experiment C was dedicated to studying if there was a limitation to how many PerfMgt messages our switches could handle. To investigate this, we
used flooding. First, a directly connected switch was flooded, then multiple switches in a chained topology.

5.4.1 Results C1

In this experiment, the compute node running the querying software was directly connected to one single switch. We choose to use the first switch in the chained topology from the previous experiment to avoid having to re-cable the cluster. This switch was a Mellanox IS5025. We ended up with four scenarios for the number of requests to send to the switch. The most intensive scenario had a duration of just over one minute and consisted of 50,000 port counter requests for each port, 1,800,000 requests in total. The collected data from this experiment is presented in Figure 5.11 and Figure 5.12. The first plot presents the results from hardware runs, and the second plot shows data from simulated runs. As before, the time spent on the CPU has been removed from the wall time data source. We expected this flooding to be CPU intensive, so we kept an eye the CPU usage during the experiment.

The plots from the hardware run show a slight increase in the average response time from the switches as the number of requests is increasing, but the PMA was handling it well. In the case of an overloaded PMA, it would be a visible spike in the response times in this plot. In the first run, we issued one single query. The test result of a single query showed a
response time of 2.1 milliseconds. The number of queries was increased to 50 and the average response time rose to 2.2 milliseconds. We wanted to try to push the switches even harder, so the number of queries was increased to 5,000 in the next run. This run resulted in an average response time of 2.6 milliseconds.

When this experiment was designed, we had little knowledge of what to expect and how many queries would be needed to flood the PMA adequately, so the configuration of each flood level was not decided beforehand. At this point in the experiment with 5,000 samples requested sent, we still saw increased response times, and the time it took to query was under one minute. Due to these reasons, we increased the number to 50,000. Results showed that the average response time from this run was 2.6 milliseconds, which is the same result as in the previous experiment. The duration of the flooding was just above one minute. We believe that sending 50,000 queries with a duration of just above one minute is a significant enough stream of queries for this stress test of the PMA. It is way above what anyone would require for monitoring of a subnet. The traffic overhead injected in the subnet of such rapid monitoring is not wanted. More on this later in this chapter.

When we look at the plots, as mentioned above, a slight increase in the average response time is seen when the number of queries was increased. It stabilizes at 2.6 milliseconds with 5,000 and 50,000 queries. At the same time, the plot shows that the standard error in the WALL time is decreasing with an increased number of queries. The time used in the CPU has a decreasing trend, but the standard error is growing slightly.

The plots from the simulation present a trend opposite from the hardware run. Both the CPU time and WALL time decreased when we increased the number of queries to flood with. The standard error in each flood level did also decrease. In the toughest flooding experiment, with the switch being repeatedly queried for 50,000 times, the standard error was 4,831 nanoseconds. It is again evident that IBSim is not a simulator that we can extract useful information from, with regards to the performance of the PMA. We will not include results from simulated runs for Experiment C2 and C3 in this thesis due to these results.

After analyzing this experiment, we can conclude that hardware switches are handling what we consider a fair amount of requests over time very well, and that we by using the libibmad RPC functions from a single host are not able to overload it by flooding it in a time span of one minute. It was noted that perfswiper used around 20% CPU, and libibmad used around 30% during these experiments.
Figure 5.11: Experiment C1: Results from hardware.

Figure 5.12: Experiment C1: Results from simulation.
5.4.2 Results C2

In Experiment C2, the number of switches to flood with requests was increased. We used the same chained topology as introduced in Experiment A. In the previous experiment we saw that the flooding resulted in high CPU usage by both perfswiper, and the libibmad library. It became apparent that we in this experiment would not be able to overload any of the switches since we would be limited by the CPU on the compute node. The plan to spawn multiple processes of perfswiper would not help. To illustrate the limitation that we encountered, we present the results from 10,000 queries in Figure 5.13. During this experiment, we saw a 100% load on the CPU of the compute node used for flooding.

The CPU time results from this experiment are similar to previous results, sitting on around two milliseconds. The boxes representing the WALL time response times (excluding CPU time) have a distinct curve were the highest boxes are in the middle. This pattern is explained by the way the flooding was executed. Each of the querying processes was manually started in multiple terminals, thus getting a delay of around one second between each process were started. This caused the pressure on the CPU to be lower during the first and last switch compared to the middle. The WALL time results show an enormous increase in response times from what we have seen before in this thesis. This increase is not caused by overloading the switches, but overloading the CPU on the compute node we used. The largest average response time logged in this experiment is just over 13 milliseconds.

We see now that we in this experiment should have used multiple compute nodes for traffic generation in this experiment. This is left as future work.

5.4.3 Experiment C Summary

After working through the data collected in Experiment C, we can conclude that the switches we have, handled being queried by our software well. We are not able to find any data that suggest we have an overloaded switch or PMA during our tests. We did see a slight increase from 2.2 milliseconds to 2.6 milliseconds from a single query to 50,000 queries, and our data suggest that the response time is stabilized at 2.6. Increasing the number of queries also leads to a result with a data set with less deviation.

From Experiment C we can again see how the data sets from simulated runs using IBSim not relate to the real world. The more queries we send when simulating a subnet with IBSim results to a lower response time. We also see that the deviation is almost reduced to zero when sending 50,000 queries.
5.5 Experiment D

In Experiment D we were interested in testing the software designed in chapter 3, and to further collect data on how often switches can be queried for its performance counters. To investigate this, we tested perfswipe on a typical IB network topology. We also simulated a large cluster using IBSim to collect some knowledge on scaling.

5.5.1 Results D1

In Experiment D1, a fat-tree topology was introduced. The fat-tree topology is one of the most used topology configuration seen in IB networks. This experiment was executed using one compute node and one instance of perfswipe. Each switch and port combination was queried in three intervals, 1x, 50x, and 5,000x. The main results of this experiment are presented in Figure 5.14. An average response time for each load level was calculated and is presented in Table 5.1.

The plots of the individual switches and their response times are found in Figure 5.14. It presents no new data of substance, but we can use the response times from this experiment to try and calculate how often it will be possible for perfswipe to swipe the fat-tree subnet that was used in this experiment.
In Table 5.1, an average response time of the whole subnet, for each load level is given. The data collected shows that on average, all metrics are gathered from a 36-port switch in about two milliseconds. We know that the travel distance, and that the number of switches a MAD has to pass through have a minimal impact when the network has under 90% load on links. From previous experiments, we have found that the time perfswiper spends on the CPU is about two milliseconds and that the amount of time is not increasing when the level of load in the network is. Simplified and with rounded numbers, this means that perfswiper can collect metrics from 250 36-port switches in one second.

If multiple links in the network have a load of more than 90%, the number of switches perfswiper can query is going to be lower. In a two-level fat-tree, the most switches a MAD needs to pass through before hitting the switch it is destined for is two. When looking at the result from Experiment A2, we can see that we in this case with a MAD having to pass two switches, where both links have a load between 80% and 90%, we can expect a switch to respond within about five milliseconds. When including CPU time, the total response time is about seven. These numbers proclaim that in a network with between 80% and 90% load on all links, perfswiper will only be able to query about 142 switches in one second.

Figure 5.14: Experiment D1: Real-life test on a fat-tree topology.
<table>
<thead>
<tr>
<th>Number of Queries</th>
<th>Average Response Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>2,190,643 ns (2.1 ms)</td>
</tr>
<tr>
<td>50x</td>
<td>2,089,465 ns (2.0 ms)</td>
</tr>
<tr>
<td>5,000x</td>
<td>2,270,566 ns (2.2 ms)</td>
</tr>
</tbody>
</table>

Table 5.1: Experiment D1: Average response times

5.5.2 Results D2

Experiment D2 was aiming to execute a large-scale simulated test of perfswiper on a fat-tree topology. We were able to simulate a two-level fat-tree topology with 37 switches a total of 2,044 ports. We wanted to verify that perfswiper also could work with a larger amount of switches and to learn more about the swipe interval.

Perfswiper worked well in this test. Two groups of three switches had a spike in their CPU time, even though perfswiper was executed with the best possible scheduling algorithm, and with no other applications competing for CPU resources. In this simulated run, polling the very first switch used 477,551 nanoseconds of CPU time and the WALL time with CPU time subtracted was 302,581 nanoseconds. Without looking at the very first query, and the two groups with spikes, the average measured WALL time results is at 208,141. The response times measured from simulations using IBSim in this experiments are not very useful, but the test itself verifies that perfswiper can operate in this larger topology.

Based on results in this thesis, we can claim that on average, all of the switches in a fat-tree topology will have a response time of two milliseconds when links have less than 80% load. With a link load above 80%, the response times increase to five milliseconds. We can claim this since we know that in this topology, the most switches that the MADs have to pass through is two. Based on this we can apply the same formula as in D1 which showed that perfswiper could gather metrics from about 250 36-port switches in one second. This means that in the fabric we simulated in this experiment, perfswiper would be able to collect metrics from the whole subnet between six and seven times in one second, which is between 166 and 142 milliseconds per subnet swipe. From our flooding experiments, we know that this is far from an amount of queries that would cause a switch to be overloaded.

5.5.3 Calculate Minimum Swipe Interval

In this section, we present a formula that can be used to calculate the theoretical minimum swipe interval in milliseconds to be used in perfswiper.
when deployed in a two-level fat-tree topology. This formula is based on a
couple of assumptions, and it is a bit simplified. Since we have no data on
time spent to transmit collected metrics to fsmonitoring, nor data on time
spent processing topology updates we assume that this is zero to be able to
make this simplified formula. In the formula, $I$ is representing the resulting
minimum interval, $T_c$ is the CPU time perfswiper is using per port. $T_q$ is
the time elapsed from the RPC request was sent until perfswiper receives
the metrics. $P$ is the number of ports present in the fabric.

$$I = (T_c + T_q)P$$

In this thesis, we have observed a $T_c$ value of approximately two milliseconds.
The $T_q$ was observed to be about two milliseconds in a network with less
than 80% load on links. With multiple links above 80%, we observed a $T_q$
of approximately five milliseconds.

5.5.4 Experiment D Summary

The last experiments in this thesis were designed to test the functionality
of perfswiper and to get insight on how often we could query switches for
their for performance metrics. The experiments in this section did this by
testing perfswiper in a typical network topology used in IB networks. We
used rounded numbers and ignored the time that perfswiper is using to talk
with fsmonitoring to calculate the number of switches that perfswiper can
swipe in one second. It was estimated that perfswiper could swipe a subnet
of up to 250 36-port switches in one second when there is under 80% load
on network links. If the network has more than 80% load on one or more
links, the number is reduced to 142 switches in one second.

In the largest fat-tree topology we were able to build in our lab, it was
estimated that perfswiper could swipe the entire subnet, in intervals as small
as about 37 milliseconds.

We simulated a subnet with 37 36-port switches and verified that perfswiper
could operate in this fabric. The response times from this experiment could
not be used to calculate the lowest possible interval to use, due to how
IBSim is built. Instead, based on previous experiments, we used the average
response times from Experiment D1 to calculate the interval. We found that
in a topology with 37 of these switches, perfswiper could collect metrics from
the whole fabric about every 166 milliseconds. At the end of this section,
we gave a simplified formula that can be used to calculate the minimum
possible swipe interval in a two-level fat-tree topology.
5.6 Query Size and Introduced Overhead

As presented in the technical background in chapter 2, each MAD has a fixed size of 256 bytes. And we introduced the capability mask and its features.

During all of our experiments, collecting metrics from a port involves three MADs. A total of 768 bytes has to be transmitted to each switch and back again. The first query is for the mandatory port counters. Subsequently, the switch capabilities have to be requested to check if it supports 64-bit counters. If it does, a third MAD has to be sent to collect them. The IBA offers no way of collecting 32-bit and 64-bit counters at the same time using a single MAD.

Back in Experiment D1 (Section 5.5.1), we found that when running perfswiper on a compute node with the specifications we listed in Table 4.2, it was able to query metrics from 250 36-port IB switches in one second. This means that when perfswiper was operating at its maximum capacity in our experiments, it produced 6,912,000 bytes (6,912 Kilobytes) of outgoing MADs per second. In our lab, 4x SDR (16 Gbit/s) was used on links. The injected MAD traffic in Experiment D, constituted to about 0.35% of the total capacity of the link. In an IB fabric with 4x FDR (54.54 Gbit/s), the traffic would constitute to about 0.10% of the available bandwidth.

Back in Experiment C1 (Section 5.4.1), when we discussed the flooding results of Experiment C1, we mentioned that such rapid monitoring is not wanted. In that experiment, at the heaviest level of flooding, it was issued 1,800,000 requests to a switch. One request consisted of 3 MADs, each 256 bytes long. A total of 1.38 Gigabytes of data was generated. The flooding lasted for about one minute, and during this flooding, the egress rate from the compute node was about 184 Mbit/s. This generated overhead is why we in C1 said, that such rapid monitoring is not wanted as it steals a lot of bandwidth.

To round off; it is a minuscule amount of overhead is added to the network by our monitoring software. In requital, our in-band monitoring solution does not need any of the valuable computing time on compute nodes, which is demanded by running HPC applications. The developed monitoring software also offloads the FSM, ensuring that it can recalculate paths and update routing tables faster when a fault in the network is occurring.
5.7 Summary

In this thesis, there were performed five experiments. The results from these five provided data that we used to answer the second group of research questions in this thesis.

We concluded that the CPU time results in itself should not be given any emphasis, since we suspect that scheduling of the monitoring software was different between runs, and even in runs.

We concluded that network load in the fabric does affect how fast the monitoring software was able to query a switch for its performance counters. It was discovered that on a single link with a network load between 80% and 90%, the time of the reply was doubled. Additional traffic did not further increase the response time significantly. We found that the number of switches that a MAD has to travel through has little impact on the time of reply. In a network with multiple links with load above 80%, the effects were more apparent. The number of links that the MAD has to pass through started affecting the response times. We found that this increase in response times seen in these experiments, did not originate from an overloaded switch, but probably caused by the management traffic being held in the switch buffers together with the artificial traffic for a longer period than when there was no or little load.

We also found that the querying pattern of the switches and ports does not influence the response times from the switches. We did not manage to collect data that we could use to decide which pattern spent the most CPU time.

From our flooding experiments, we found that the switches we used in experiments handled being queried well. Even when flooding switches with 50,000 requests in a span of just about one minute, could we not find any data that indicated an overloaded switch or PMA. From the flooding, it was also shown how IBSim is not a simulator that we can get useful real-life data from. The more queries we sent to IBSim, the lower response time we got, and the deviation in the results were almost zero.

The last experiments that we conducted aimed to test perfswiper in a typical HPC network topology. We found that perfswiper was able to collect metrics from 250 36-port IB switches per second when there was under 80% load on network links. When the network load was increased to a point between 80% and 90% and above, the number of switches that perfswiper could monitor in one second was reduced to 142.

At the end of this chapter, we discussed the data overhead that perfswiper is introducing in the fabric it monitors. We found that when perfswiper is operating at its highest capacity with 250 switches queries in one second,
it produces 6,912 Kilobytes of outgoing MADs. This constitutes to about 0.35\% of the 4x SDR (16 Gbit/s) pipe we used in this thesis. In a fabric with 4x FDR (54.54 Gbit/s), perfswiper uses about 0.10\% of the pipe.
Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis has shown how to build an in-band IB fabric monitoring plugin for the FFM which is developed by Fabriscale Technologies AS, using libibmad and libibumad. This thesis also showed how application traffic in an IB fabric affects the performance of the monitoring software. We demonstrated how switches handled in-depth querying over time and pushed the switches to their limit by flooding them with requests for port counters. This thesis found that when the network has a load of between 80% and 90%, the response times from switches was doubled. It was learned that an overloaded switch entity did not cause the slowdown, but it was believed that it is caused by packets sitting in buffers longer.

The built monitoring software was tested in a typical IB network topology and tested in a large scale simulated environment using IBSim. Results from these tests show that perfswiper works well. We found that the number of switches that perfswiper can collect metrics from within one second varies with the amount of load in the network. With the hardware we had available, configured in a two-level fat-tree topology with a network load that is under 80%, perfswiper was measured to be able to collect metrics from 250 36-port switches in one second. With a network load of somewhere between 80% and 90% and above on two links, this number is reduced to 142 36-port switches.

Furthermore, this thesis calculated how much overhead that is introduced in the IB fabric when doing in-band monitoring, and we looked at how this scales with the number of nodes to monitor. To query both 32-bit and 64-bit counters from an IB switch, a total of 768 bytes for each port has to
be transmitted to and from the switch. When perfswiper was operating at
its maximum capacity in our experiments, querying 250 36-port switches
each second, a total of 6,912 Kilobytes was transmitted to the network from
the monitoring host every second. This constituted to about 0.35% of the
4x SDR link we used in our experiments. With a 4x FDR, this number
is reduced to 0.10%. We consider this amount of introduced overhead as
minuscule and acceptable. Our in-band monitoring is not putting any load
on the CPUs of compute nodes used for HPC applications since we use IB
MAD for collecting metrics.

6.2 Future Work

The performance metric plugin to the FFM developed in this thesis is built
with IB management in mind, but the communication protocol between
perfswiper and fsmonitoring is designed to support Ethernet as well. One of
the advantages of developing the monitoring plugin as a separate entity
was that it could also monitor clusters with an Ethernet interconnect.
In the future, perfswiper can be extended to support Ethernet. It can
also be of great value to Fabriscale to build thread support in perfswiper.
By adding additional threads to handle incoming topology messages, and
transmission of metrics to fsmonitoring, perfswiper can sample ports with a
higher resolution.

While analyzing the data sets from the experiments performed, we learned
more about the uncertainties with regards to data samples based on timers.
We suggest that all of our experiments are executed again on compute nodes
with better hardware, and with a set of identical switches. Having more
CPU cores available could reduce the number of preemptions of perfswiper.
We would also like to have information about the scheduling to learn more
about the variance in CPU time. Specifically, it would be interesting to
see if the number of context switches differs, and get knowledge about how
much time elapsed in perfswiper to wait for I/O.

In Experiment B we failed to wrap the code executing the switch and port
pattern in a timer, so the CPU time results said nothing about which
pattern was the most effective. The data that was collected (which we
could not use anyway) was affected by scheduling since each pattern were
run as a separate processes. Rerunning experiment B with a timer set up
correctly, and running all three patterns from one process, could generate
some interesting data.

In Experiment C we expect that better hardware in the compute nodes used
for experiments would benefit our attempt to overload switches with MADs.
Further investigations by using multiple compute nodes to query one single
switch would give better knowledge on how many MADs the switches can
handle. Running these experiments with a higher data rate enabled on ports could also benefit the flooding.

From the data collected in this thesis, we found that perfswiper is theoretically able to collect metrics every second from a cluster with 250 36-port switches. We would like to see the real performance of perfswiper in such a network, and see how application traffic would impact the number of switches it can collect metrics from.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM</td>
<td>Association for Computing Machinery.</td>
</tr>
<tr>
<td>API</td>
<td>an application programming interface (API) is a set of functions and procedures that allow the creation of applications which access the features or data of an operating system, application or other services.</td>
</tr>
<tr>
<td>ASIC</td>
<td>application-specific integrated circuit.</td>
</tr>
<tr>
<td>Berkely sockets</td>
<td>is an API for Internet or Unix sockets.</td>
</tr>
<tr>
<td>BMA</td>
<td>baseboard management agent.</td>
</tr>
<tr>
<td>BSD</td>
<td>Berkeley Software Distribution.</td>
</tr>
<tr>
<td>BSD license</td>
<td>is an open-source license with minimal restrictions on redistribution of the covered software.</td>
</tr>
<tr>
<td>BTH</td>
<td>base transport header.</td>
</tr>
<tr>
<td>Compute node</td>
<td>is one computer which is interconnected in a cluster of multiple machines.</td>
</tr>
<tr>
<td>CPU</td>
<td>central processing unit.</td>
</tr>
<tr>
<td>CPU time</td>
<td>is the time that a process has been active in the CPU.</td>
</tr>
<tr>
<td>CRC</td>
<td>cyclic redundancy check.</td>
</tr>
<tr>
<td>DDR</td>
<td>double data rate.</td>
</tr>
<tr>
<td>EDR</td>
<td>enhanced data rate.</td>
</tr>
<tr>
<td>EE context</td>
<td>end-to-end context.</td>
</tr>
<tr>
<td>ETH</td>
<td>extended transport header.</td>
</tr>
<tr>
<td>FDR-10/FDR-14</td>
<td>fourteen data rate.</td>
</tr>
<tr>
<td>FFM</td>
<td>Fabriscale Fabric Manager.</td>
</tr>
<tr>
<td>FLOOPS</td>
<td>floating-point operations per second.</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>FRE</td>
<td>Fabriscale routing engine.</td>
</tr>
<tr>
<td>FSM</td>
<td>Fabriscale subnet manager.</td>
</tr>
<tr>
<td>GB/s</td>
<td>gigabyte per second.</td>
</tr>
<tr>
<td>GbE</td>
<td>Gigabit Ethernet.</td>
</tr>
<tr>
<td>Gbit/s</td>
<td>gigabit per second.</td>
</tr>
<tr>
<td>GMP</td>
<td>general management packet.</td>
</tr>
<tr>
<td>GPU</td>
<td>graphics processing unit.</td>
</tr>
<tr>
<td>GRH</td>
<td>global route header.</td>
</tr>
<tr>
<td>GSA</td>
<td>general service agent.</td>
</tr>
<tr>
<td>GSI</td>
<td>general service interface.</td>
</tr>
<tr>
<td>GUI</td>
<td>graphical user interface.</td>
</tr>
<tr>
<td>GUID</td>
<td>global unique identifier.</td>
</tr>
<tr>
<td>HCA</td>
<td>host channel adapter.</td>
</tr>
<tr>
<td>HDR</td>
<td>high data rate.</td>
</tr>
<tr>
<td>HPC</td>
<td>is a term used for a computer or computer system with a high level of computing performance compared to a general-purpose computer.</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol.</td>
</tr>
<tr>
<td>IB</td>
<td>InfiniBand.</td>
</tr>
<tr>
<td>IBA</td>
<td>the InfiniBand Architecture (IBA) is the name of the InfiniBand specification.</td>
</tr>
<tr>
<td>IBTA</td>
<td>InfiniBand Trade Association.</td>
</tr>
<tr>
<td>IEEE</td>
<td>the Institute of Electrical and Electronics Engineers (IEEE) is the world’s leading professional association for the advancement of technology.</td>
</tr>
<tr>
<td>INQS</td>
<td>InfiniBand Network Querying Service.</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol.</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization.</td>
</tr>
<tr>
<td>ITU-T</td>
<td>Telecommunications Standardization Sector International Telecommunication Union.</td>
</tr>
<tr>
<td>JSON</td>
<td>Javascript Object Notation (JSON) is a human-readable text format for data encoding, originally, JavaScript objects in text form.</td>
</tr>
<tr>
<td>LAN</td>
<td>local area network.</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>LED</td>
<td>light-emitting diode.</td>
</tr>
<tr>
<td>LID</td>
<td>local identifier.</td>
</tr>
<tr>
<td>LRH</td>
<td>local route header.</td>
</tr>
<tr>
<td>MAD</td>
<td>management datagram.</td>
</tr>
<tr>
<td>MB/s</td>
<td>megabyte per second.</td>
</tr>
<tr>
<td>MME</td>
<td>module management entity.</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface.</td>
</tr>
<tr>
<td>MTU</td>
<td>maximum transmission unit.</td>
</tr>
<tr>
<td>MVAPICH</td>
<td>message passing interface (MPI) over InfiniBand.</td>
</tr>
<tr>
<td>NTP</td>
<td>Network Time Protocol.</td>
</tr>
<tr>
<td>OFED</td>
<td>OpenFabrics Enterprise Distribution.</td>
</tr>
<tr>
<td>OS</td>
<td>operating system.</td>
</tr>
<tr>
<td>OSI</td>
<td>Open Systems Interconnect.</td>
</tr>
<tr>
<td>OSU</td>
<td>Ohio State University.</td>
</tr>
<tr>
<td>PCIe</td>
<td>Peripheral Component Interconnect Express.</td>
</tr>
<tr>
<td>PerfMgt</td>
<td>performance management.</td>
</tr>
<tr>
<td>PHY</td>
<td>physical layer.</td>
</tr>
<tr>
<td>PMA</td>
<td>performance management agent.</td>
</tr>
<tr>
<td>POSIX</td>
<td>is a family of standards specified by the IEEE Computer Society for maintaining compatibilities between operating systems first.</td>
</tr>
<tr>
<td>QDR</td>
<td>quad data rate.</td>
</tr>
<tr>
<td>QoS</td>
<td>quality of service (QoS) is referring to the ability to achieve maximum bandwidth in regards to performance elements such as latency, error rate, and uptime.</td>
</tr>
<tr>
<td>QP</td>
<td>queue pair.</td>
</tr>
<tr>
<td>RC</td>
<td>reliable connected.</td>
</tr>
<tr>
<td>RD</td>
<td>reliable datagram.</td>
</tr>
<tr>
<td>RDMA</td>
<td>remote direct memory access.</td>
</tr>
<tr>
<td>REST</td>
<td>representational state transfer.</td>
</tr>
<tr>
<td>RPC</td>
<td>a remote procedure call (RPC) is a service request from a remote server issued using a network.</td>
</tr>
<tr>
<td>Acronym</td>
<td>Definition</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>SDR</td>
<td>single data rate.</td>
</tr>
<tr>
<td>SL</td>
<td>service level.</td>
</tr>
<tr>
<td>SM</td>
<td>subnet manager.</td>
</tr>
<tr>
<td>SMA</td>
<td>subnet manager agent.</td>
</tr>
<tr>
<td>SMI</td>
<td>subnet manager interface.</td>
</tr>
<tr>
<td>SMP</td>
<td>subnet management packet.</td>
</tr>
<tr>
<td>Subnet</td>
<td>is short for subnetwork. A logical sub-division of a larger network.</td>
</tr>
<tr>
<td>Supercomputer</td>
<td>is a computer or computer system that operates at or near, the currently highest operational rate for computers.</td>
</tr>
<tr>
<td>SysV</td>
<td>UNIX system V.</td>
</tr>
<tr>
<td>TACC</td>
<td>Texas Advanced Computing Center.</td>
</tr>
<tr>
<td>TCA</td>
<td>target channel adapter.</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol.</td>
</tr>
<tr>
<td>UC</td>
<td>unreliable connected.</td>
</tr>
<tr>
<td>UD</td>
<td>unreliable datagram.</td>
</tr>
<tr>
<td>UNIX</td>
<td>is a family of multitasking, multi-user computer operating systems that derive from the original AT&amp;T Unix first.</td>
</tr>
<tr>
<td>UTC</td>
<td>Coordinated Universal Time.</td>
</tr>
<tr>
<td>VL</td>
<td>virtual lane.</td>
</tr>
<tr>
<td>WALL time</td>
<td>is the time measured with a 'wall-clock'.</td>
</tr>
<tr>
<td>WVI</td>
<td>Web-based Visualization Interface.</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language (XML) is a flexible human-readable text format for data encoding.</td>
</tr>
<tr>
<td>XRC</td>
<td>extended reliable connected.</td>
</tr>
<tr>
<td>Zero-copy</td>
<td>is a data transfer mechanism where the transfer of an upper layer buffer payload between peers is done directly into the upper layer buffers, thus avoiding copying of the payload multiple times.</td>
</tr>
<tr>
<td>ZeroMQ</td>
<td>Zero Message Queue.</td>
</tr>
</tbody>
</table>
Bibliography


