Development and implementation of data acquisition software for proton computed tomography

Alf Kristoffer Herland

Master's thesis in Software Engineering at

Department of Computer science, Electrical engineering and Mathematical sciences,
Western Norway University of Applied Sciences

Department of Informatics,
University of Bergen

June 16, 2021

Western Norway
University of
Applied Sciences

Abstract

The proton Computed Tomography (pCT) project is a collaboration between the University of Bergen (UiB), Western Norway University of Applied Sciences (HVL) and several international institutions. pCT is an imagining technology used to plan treatment dosages for proton radiation therapy. This type of radiation therapy is more accurate on target than the conventional photon-based radiation therapies currently offered here in Norway. The goal of utilizing proton therapy is to have less damage to healthy tissue surrounding the tumor, than what can be achieved with photon-based therapy. The main component that allows the detection of protons is the ALICE PIxel DEtector (ALPIDE) sensor chip that is developed at CERN in Switzerland and France.

In large projects like the pCT collaboration, there are many smaller subsystems that need to work together to complete the goal of having a working pCT detector prototype. There have been numerous people who have worked on this goal before this thesis and there is still more work to be done after. The work that has laid the foundation for this thesis solves an important step in the chain of data acquisition in the pCT detector. This step is the software that handles the custom proton Data Transfer Protocol (pDTP) used to read out data from the proton Readout Unit (pRU), and also parts of the software that performs the read and write operations for the control system is covered in this thesis.

This thesis presents the software theory that is the foundation to design and implement an efficient protocol client to handle the high volume of incoming data on the 10 Gigabit Ethernet (GbE) Network Interface Card (NIC) used in the host computer. The theory behind the design methodology that is used to achieve modular and adaptable software components.

Finally, an analysis of the system and benchmark test results shows the system's ability to perform at the maximum 10 gigabits per second rate of the hardware interface. There is also a discussion on how to adapt the Operating System (OS) to find the optimum settings, allow data to pass the internal workings of the OS.

Acknowledgements

I would extend heartful thanks to my two supervisors, *Professor Håvard Helstrup* and *Associated Professor Johan Alme*. You have taught me a lot about the art of academics. As supervisors, you both went above and beyond your responsibilities. If it was not for the support you offered, I would not have been able to complete this thesis.

A big thanks to *Dr.Matthias Richter* for his insights on software design and guidance with C++ programming.

Thanks to *Dr.Ola Slettevoll Grøttvik* for all the exciting tasks, advice, and inspiring me to elevate the quality of my work.

I am grateful for the collaboration and pair-programming sessions I have had with *Viljar Eikeland*. Thank you to *Øistein Skjolddal* for all our discussions during the work with our theses.

Also, a big thanks to the whole of the team involved with the pCT project. It has been a pleasure to have had the opportunity to contribute to it.

For my friend *Tea Bodova*, thank you for all the good times, laughter, and all the technical adventures we have had for the last 5+ years during our studies.

I would like to thank *Kjersti* for all her patience and understanding during my studies and thesis work. Finally, I want to thank the rest of my family for their support and love.

Alf Kristoffer Herland
Bergen, June 2021

Contents

Acronyms

ALICE A Large Ion Collider Experiment at CERN.
ALPIDE ALICE PIxel DEtector.

BDD Behavior-Driven Development.
BSP Board Support Package.

CPU Central Processing Unit.
CT Computed Tomography.

DAQ Data Acquisition.
DCS Detector Control System.
DHCP Dynamic Host Configuration Protocol.

ECC Error Correcting Code.
Ethernet Ethernet.

FIFO First-In-First-Out.
FPGA Field Programmable Gate Array.
FSM Finite State Machine.

GbE Gigabit Ethernet.
GCC GNU Compiler Collection.

HVL Western Norway University of Applied Sciences.

IPBus IPbus Control Protocol.
IPv4 Internet Protocol v4.
IRQ Interrupt request.

LSB Least Significant Bit.

MCU Micro Controller Unit.
MSB Most Significant Bit.

NIC Network Interface Card.

OpCode Operation code.
OS Operating System.

PCB Printed Circuit Board.
pCT proton Computed Tomography.
pDTP proton Data Transfer Protocol.
PHY Physical Layer Device.
POSIX Portable Operating System Interface.
pRU proton Readout Unit.
PTB Production Test Box.

QSFP+ Quad Small Form-factor Pluggable.

RAM Read Access Memory.
RTOS Real Time Operating System.
RU Readout Unit.

SFP+ Small Form-factor Pluggable.
SPSC Single Producer Single Consumer Queue.

TCP Transport Control Protocol.

UDP User Datagram Protocol.
uHAL micro Hardware Access Library.
UiB University of Bergen.
UML Unified Modeling Language.
USB Universal Serial Bus.

XML Extensible Markup Language.

List of Figures

List of Tables

Listings

CHAPTER 1

Introduction

1.1 Background

In 2017 there were 33564 new cases of cancer in Norway, and every year there is a small increase in number of cases [1]. These numbers reflect the need for new ways to both diagnose and treat cancerous cases. In 2018, the Norwegian Parliament decided in the revised national budget to allocate funds for proton therapy centers both in Oslo and in Bergen [2].

The Department of Physics and Technology (IFT) at the University of Bergen (UiB) and the Department of Computing, Mathematics, and Physics at Western Norway University of Applied Sciences (HVL) is currently developing a proton Computed Tomography (pCT) system in collaboration with several international institutions. The project aims to explore the feasibility of a medical detector system. New technology from CERN in Switzerland and France makes it possible to detect particles and reconstruct these particles' trajectories from a source through a phantom and into a detector with greater accuracy than in the past.

1.1.1 Radiation Therapy

In order to diagnose patients with cancer, there are several imaging tools available: Magnetic Resonance (MR), Positron Emission Tomography (PET), Computed Tomography (CT), to name a few. Normally, when radiotherapy is selected as the primary form for treatment, a treatment plan is usually developed based on the CT imaging that needs to be performed before the treatment sessions. Since these images are captured in different rooms than the radiotherapy equipment and at different times than treatment, this introduces several error sources. One of the sources is that organs and tissues might shift in location in the time frame between imaging and therapy [3].

A significant side effect of radiotherapy is that the source of the radiation used is photons that are accelerated in a beam into the patient. These photons have very little stopping power. This means that even though the target (the cancerous tissue) might be located near to the skin surface, the particle will pass through the rest of the body, damaging the healthy tissue in its path [4].

1.1.2 Photons vs Protons

The primary behavior of the two types of particles regarding attenuation is that photons have a high value that drops off slowly over distance. However, the proton has a relatively low value until something that is called a Bragg peak occurs. At this Bragg peak, most of the energy of the proton is deposited. In other words, it will deposit most of its energy into the target at this peak.

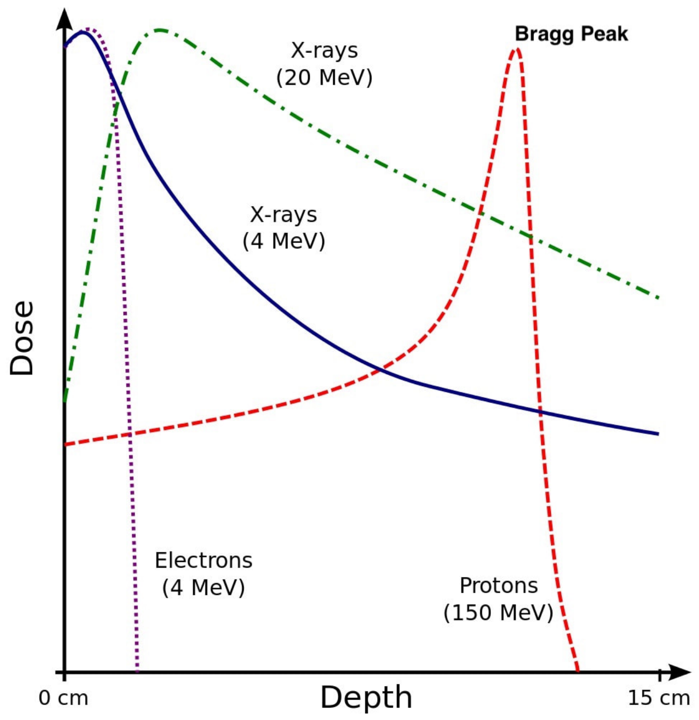

Figure 1.1: Graph with a Bragg peak in comparison to photon and electron attenuation [5] .

1.1.3 Proton Therapy

Today dose planning for proton radiation therapy relies on regular CT and the use of a technique called Hounsfield unit conversion. This conversion calculates the proton stopping power in the tissue where a tumor is located. Due to the body containing several types of tissue and these different types of tissue has properties that affect the stopping power when the radiation passes through it[6]. When performing the conversion calculation, several uncertainties might affect the precision of where the beam directs the radiation. The effect of this is that it might end up damaging the healthy tissue that surrounds the tumor. One of the motivations for the Bergen pCT detector is that it can produce imaging that has a lower percentage of error so that radiation dosage gets delivered with higher accuracy.

1.2 Problem Description

1.2.1 Bergen proton Computed Tomography (pCT) Detector

The active pixel sensor (ALPIDE) chip was developed for the A Large Ion Collider Experiment at CERN (ALICE) Inner Tracking System [7]. These sensors are used as the main component in the detector signal chain for pCT. The detector chips are setup in several arrays in a slice configuration (as seen in figure 1.3). There is also provided a software package for Data Acquisition (DAQ) and Detector Control System (DCS) of the ALPIDE from CERN. There is currently work underway to adapt and adjust the readout system with the DAQ and the DCS for the ALPIDE detectors for use in a medical pCT system.

The ALPIDE is a Monolithic active pixel sensor, fabricated through a 180 nm CMOS Imaging Sensor process. The sensor consists of a pixel matrix with 512x1024 pixels, an individual pixel measures 28μm x 28μm in size [7].

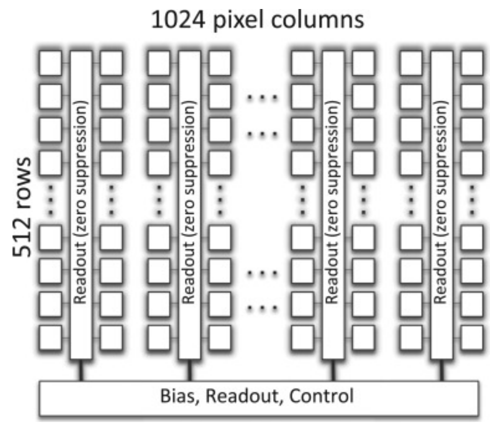
1024 pixel columns
512 rows
Readout (zero suppression)
Readout (zero suppression)
Readout (zero suppression)
Readout (zero suppression)
Bias, Readout, Control

Figure 1.2: Architecture of the ALPIDE chip [7, Figure 1].

The pCT detector unit as a whole will contain over 4100 individual ALPIDE chips. The detector is organized in 43 layers consisting of 12 staves with each 9 chips. Each layer is controlled by a proton Readout Unit (pRU) that handles the data offload and the control of the chips. Several issues arise regarding such a high number of sensors. The scope of this thesis will cover the aspect of data flow and the control system in software.

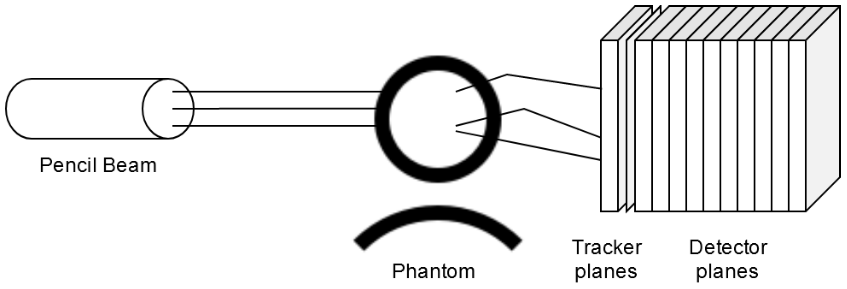

Figure 1.3: Setup for a proton CT with tracking calorimeter telescope.

A single ALPIDE chip has the theoretical potential to generate 1.2 Gbps of readout data if all pixels receive a particle hit. This theoretical number is somewhat unrealistic since the occupancy (number of hits per ALPIDE) will be much lower. A simulation of the pRU shows a maximum speed of 1.4 Gbps per layer for the front layer dropping off to under 1.2 Gbps at the back [8]. Still this number is above what a single 1 GbE NIC can handle. Therefore, a faster NIC must be used and the next Ethernet standard NIC that is larger than 1 Gbps is the 10 GbE standard.

1.3 Research Question

Can a User Datagram Protocol (UDP) network offload client be designed that handles 10 Gbps on running on a Linux Operating System (OS) and still be reliable?

This thesis will investigate the challenges that come with designing a UDP network client that handles the offload between the readout system hardware and a computer, what can be done to mitigate any issues that might arise when the system is running.

Can the IPBus suite be employed as a replacement for an embedded control system? The thesis will also look up the feasibility of using the IPbus Control Protocol (IPBus) suite as a tool to transmit and receive control messages to the pRU.

1.4 Thesis Outline

Chapter 2 - Theoretical Background

This chapter covers the theoretical background that this thesis builds upon, namely the workings of the Ethernet, IP and UDP protocols in addition to the foundation of the Linux operating system. This chapter introduces the two project specific protocols proton Data Transfer Protocol (pDTP) and the IPBus. The chapter also includes a description of work from two previous master thesis and a description of work done for a PhD thesis.

Chapter 3 - Design and Implementation

This chapter describes the rationals behind the design decisions that were made before the implementation work was done.

Chapter 4 - Analysis and Assessment

This chapter covers the analysis of the pDTP client software and the assessment of it in order to perform at maximum.

Chapter 5 - Conclusion

This chapter is dedicated to the discussion of the result and findings that where achieved during the thesis work.

Chapter 6 - Further Work

This chapter is the authors suggestions on what improvements that can be made to the software and system to increase stability and performance.

CHAPTER 2

Theoretical Background

2.1 Software Engineering Foundation

2.1.1 Software Design Patterns

Ever since the early days of software development, designers have searched for a way to reuse and structure software projects. A solution to solving and break down some of these complexities is to use software design patterns. In 1994 the authors Gamma, Helm, Johnson, and Vlissides published the book Design Patterns: Elements of Reusable Object-Oriented Software [9]. For their contribution to software engineering, they got dubbed the "Gang of Four." The book presents design patterns grouped into three different categories creational, structural and behavioral, after the publication of the Design Patterns book, the concept of concurrency patterns to the list of design patterns.

2.1.2 Policy-based Programming

Today many software projects have a high grade of complexity due to the use of object-oriented design. A significant investment in both time and work is needed to design and maintain a system architecture designed this way.

Another approach is to design the system components as policies rather than objects. In the early 2000s, the author and programmer Andrei Alexandrescu published his book Modern C++ Design [10], where he introduces the theory of using policies . This concept of policies is mostly known as the strategy pattern in the behavioral group of patterns. However, policy-based design is much more than just as the strategy pattern implies.

The primary motivation for using the more dynamic Policy-based design is to keep the software modular without coupling independent code into large static classes. Something that might need rewriting when expanding and adding new functionality. When applied correctly, a policy-based design might significantly increase flexibility in the development cycle when applying these template design elements.

Even though the policy term first got introduced over 20 years ago, it has made its way into libraries like the BOOST library—the Math Toolkit version 1.38 released in 2008, this version added the concept of policies to handle how numbers get converted to their respective data types [11]. Policies have even made their way into the C++

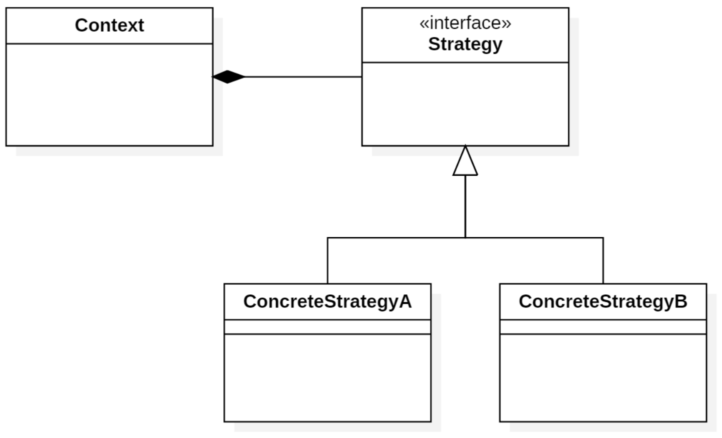

Figure 2.1: Example of the behavioral strategy pattern.

language itself. One example is in the C++11 standard, the way a `std::async` function is executed and gets specified through different policies [12].

The policy-based design utilizes the generic template system, which is a construct in the C++ language. Templates also generate byte code at compile-time, so there is more significant code optimization than traditional object-oriented designed programs. Typically, the policies are added into the template as a single policy or as tuples of two or more[10].

```
template <typename InputPolicyType, typename FilterPolicyType, typename
    OutputPolicyType>
```

Listing 2.1: The pCT readout software contains a 3-tuple policy declaration.

If a template contains more than one policy, the designer should define an interface between these policies that contain input parameters and return types. Each policy on its own can then be implemented as a class or more favorably as an anonymous lambda function. This means that policies can have several different implementations as long as it conforms to the interface between itself and other policies, e.g., the InputPolicy and the FilterPolicy.

Further explanation on how the pCT project utilizes policy-based programming can be found in section 3.4.

2.1.3 State Machine Based Design

When designing a software client for a custom network protocol like the one that will be introduced in the upcoming section 2.2.5, there are many features and functionalities to be taken into consideration when implementing it.

Within other engineering disciplines like electronics and communications, the concept of Finite State Machine (FSM) is almost compulsory when designing firmware modules or code for hardware. FSMs is also an excellent design tool for software applications, both for modeling behavior in a state machine diagram and implementing FSMs into code. An FSM has as its name indicate, finite possible states that the program can have. The FSM get implemented from a table that gets defined by the model. In addition, the FSM can have several events that act upon the FSM, events are usually external inputs from the network or users. These events trigger transitions between the states. Transitions will usually get defined as legal and illegal transitions. Finally, the last property of the FSM is the actions that get triggered when the FSM is in a given state.

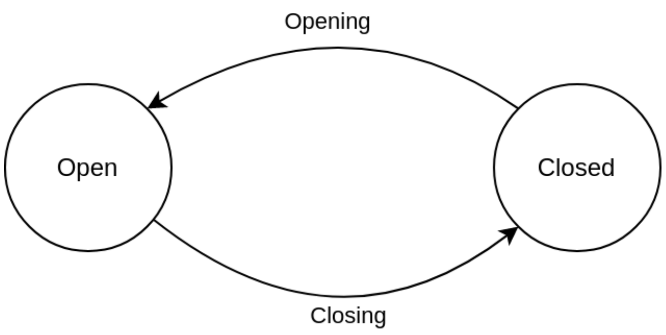

Figure 2.2: Example for a FSM model of a box with a lid.

The example in figure 2.2 is a state machine that models a box's properties. The box has two states, open and closed. The event list will only, in this case, cover two events, open lid and close lid. The same applies to the two transitions opening, and closing.

Table 2.1: FSM property table.

State	Event	Action	Transition
OPEN	Close the lid	Closing the lid	Go to Closed
CLOSED	Open the lid	Opening the lid	Go to Open

Table 2.1 can easily be implemented to code using enum to define the states and a switch/case structure to handle the transitions, omitted from listing 2.2: Helper functions for events and actions.

```
enum State {OPEN,CLOSED}; //Declare the states.
State currentState = CLOSED; //Set the initial state.

switch (currentState) {
    case CLOSED: {
        closeLid(); //Action
        getUserInput(); //Event
        currentState = State.OPEN; //Transition.
        break;
        }
    case OPEN: {
        openLid(); //Action
        getUserInput(); //Event
        currentState = State.CLOSED; //Transition.
        break;
        }
}
```

Listing 2.2: Short implementation of a FSM.

2.1.4 Concurrency - Producer-Consumer Pattern

When designing software that is modular with many tasks that need to get processed, a problem arises that there might be parts that will require more time to complete than others. A way to solve this problem is to run the different parts in multiple threads or to split the program into two processes. Several approaches exist when sharing information between threads or processes, often referred to as inter-thread or inter-process communication. This type of concurrency introduces the issue of ensuring thread safety, implementation of synchronization, and guards prevent the possibility of data corruption.

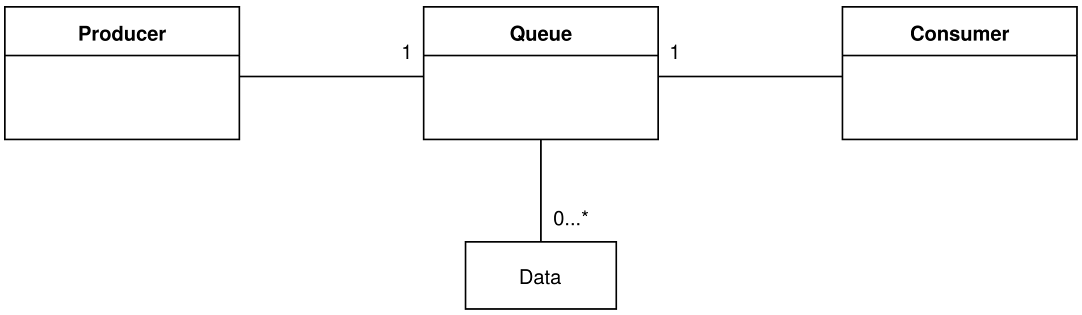

Figure 2.3: Example of the concurrency producer-consumer pattern.

A solution would be to employ the Producer-Consumer pattern implemented as a Single Producer Single Consumer Queue (SPSC) queue. This type of queue enables the ability to share data between one thread and one other thread exclusively [13]. The first one is dedicating to producing the data into the queue and the second thread is the consumer of said data.

2.1.5 Creational - Dependency Injection Pattern

In modular software design one major pitfall is to couple the different module dependencies too strongly together. A solution is to utilize the concept of dependency injection, this allows to abstract the access to for instance data-base access or network services. There are three different approaches to inject the dependents into the main class: constructor, setter function and interface injection [14]. In this thesis the constructor approach is the only one that will be covered and has been implemented.

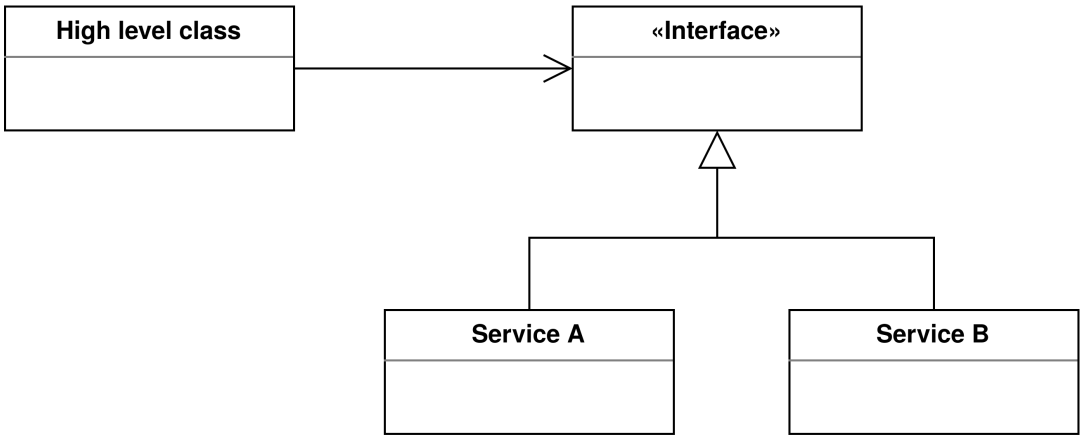

Figure 2.4: Example of the dependency injection pattern.

Constructor Injection

As the name constructor injection implies, the service class gets injected into the top class through its constructor. Utilizing this type of pattern when designing software, top classes can have children of the service class injected upon initialization.

2.2 Network Stack

The term network stack describes the communication from one or more endpoints to another endpoint. In this thesis, the Open Systems Interconnection model (OSI model) will describe the different layers of the network stack used in pCT readout chain. The OSI model stack consists of 7 Layers seen in figure 2.5. When visualizing packets transmitted in the network that traverse downwards in the OSI model layers, the most clear analogy is the box within box model. This model refers to encapsulating packets within packets. User Datagram Protocol (UDP) packet in a Internet Protocol v4 (IPv4) datagram and so forth.

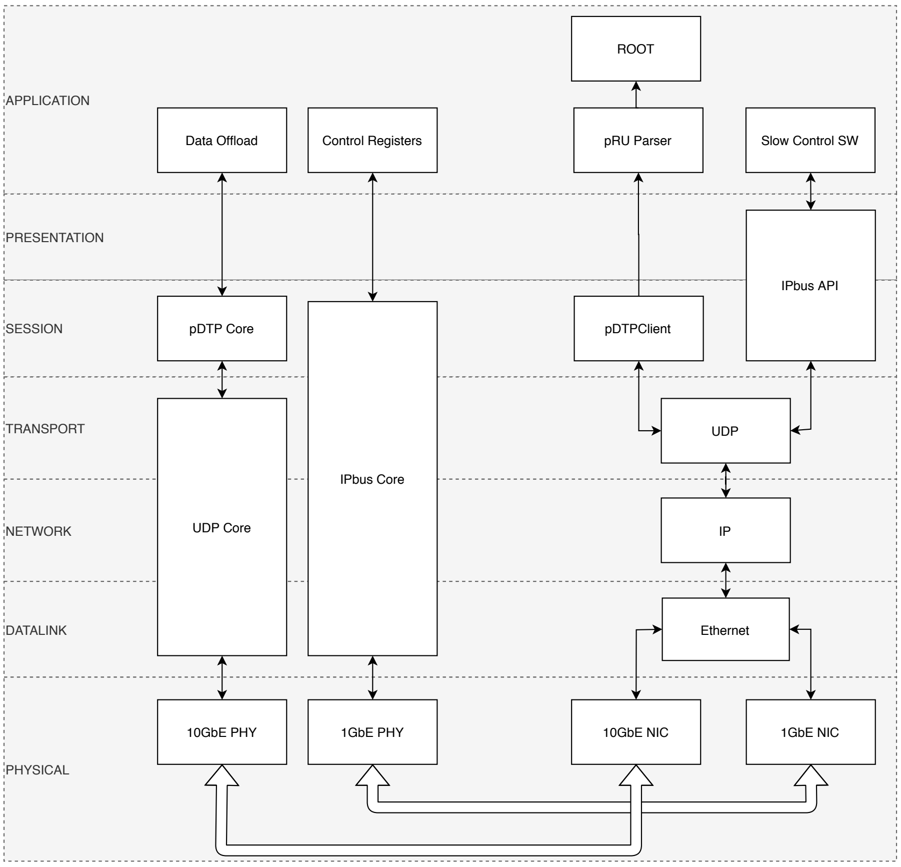

Figure 2.5: pRU and Control network stack

The metrics used in this thesis to describe data transfer are megabits per second and gigabits per second. Furthermore, the primary focus is on the bulk transfer of data instead of network latency or both. As a metric, packets per second are used. The motivation to use this term is to describe the number of packets (transactions) the system has to process each second.

2.2.1 Endianness

Endianness refers to the ordering of bytes in a data word in the memory or a network packet. That holds the position of Least Significant Bit (LSB) and Most Significant Bit (MSB). There are several different orderings, but the two main ones are big-endian and little-endian[15].

Little-endian - CPU Ordering

Most CPU architectures today are little endian or some select few that are bi-endian. Byte 0 is start with the rightmost bits.

Table 2.2: Little endian ordering.

Bit	31 24	23 16	15 8	7 0
Byte	3	2	1	0

Big-endian - Network Ordering

Big-endian is used in several network protocols, to name a few IPv4, TCP and UDP. This same applies to the pDTP protocol that will be introduced in section 2.2.5.

Table 2.3: Big endian ordering.

Bit	31 24	23 16	15 8	7 0
Byte	0	1	2	3

In protocols that uses big-endian byte ordering, the bit ordering can also be reversed. So that bit zero is the rightmost bit.

2.2.2 Link layer - Ethernet

Ethernet is a link layer standard IEEE 802.3 that enables the transmission of data, the standards that are in use today are mostly 100BASE-T and 1000BASE-T. There are also faster standards like 10GBASE-CR. To achieve the high speeds needed to meet the readout system's requirements, the normal Physical Layer Device (PHY) over 1 GbE over Cat6 copper becomes too slow. The decision to meet this requirement was to select a Small Form-factor Pluggable (SFP+) standard that is capable of both 10 (GbE) and 40 GbE(QSFP). This standard allows for copper, also called direct access cable, that provides passive 5-metre and active 15-metre cable length. For longer distances, there is even an active optical transceiver that has a range of 100 metres. One possible issue with the active transceivers is that the pRU is present in a radiation environment. These effects can influence the transceivers by changing the bits that are being sent over the medium. However, the scope of this thesis will not cover this topic.

Ethernet is a frame-based communication standard, the structure of a frame is as follows: Preamble, Destination address, Source address, Length data, and a frame check sequence(CRC). Table 2.4 lists an normal Ethernet frame.

Table 2.4: Ethernet frame

Preamble	Start Frame Delimiter	Destination Address	Source Address	Length	Data Payload	Frame Check Sequence (CRC)
7 bytes	1 byte	6 bytes	6 bytes	2 bytes	46 to 4116 bytes	4 bytes

Jumbo Frames

When utilizing Ethernet standards that provide faster transfer rates than what the 1 GbE does, and the data transmitted increases in volume over the standard frame size of 1538 bytes, the cost of overhead also grows more prominent. So to reduce the number of packet transactions per second, it is beneficial to increase the frame size. The IEEE 802.3 standard specifies this as Jumbo frames, which gives the option to configure NIC to handle frames up to 9000 bytes of payload [16].

Inter frame gap (IFG)

The 802.3 standard introduces the need for an idle gap in the Ethernet transmission. This aids in the recovery of clock signals in receivers, often referred to as an inter-frame gap (IFG). The standard defines the IFG as a 96-bit pause, in 10GbE this results in a 9.6 ns gap between frames. On reception of frames by the receiver, there is an option to decrease the IFG to a 40-bit pause. This can also be expressed as 5 bytes gap in the transmission[16, Table 4–2], calculated like this: $10 \times 10^9 \times 4 \times 10^{-9} \div 8 = 5$.

Ethernet burst traffic

Devices like FPGA have an architecture laid out in a combinatorial way. This enables the firmware to run nearly parallel compared to a more sequential computer system. The firmware can run modules at a faster clock speed than what the Ethernet module has, this in turn means that data into the module will be available faster than the speed that it transmit over the wire. Another issue that can arise is that the receiver, in this case, the computer, can be overwhelmed by the high rate of incoming Ethernet frames—especially accounting for the data moving from hardware to user space through kernel space.

Section 2.2.5 will cover a mechanism to mitigate this issue on both the pRU and on the computer that receives the data transmitted.

Section 2.3.2 will look into what parts of the OS this issue can occur and how to mitigate this effect on the system.

2.2.3 Network Layer - Internet Protocol v4 (IPv4)

Internet Protocol v4 (IPv4) is a protocol that gives each device on the network its unique address. A network interface gets either static by setting it manually, or dynamically from a Dynamic Host Configuration Protocol (DHCP) server.

IPv4s addresses are represented by a 32-bit number, which needs to be unique for each device in the local network. The IPv4-addresses are usually grouped into four groups, like 10.0.0.1 or 192.168.1.1. The pCT system follows a pre-specified addressing scheme, one for readout and one for slow control. Information about this can be found on the pCT wiki page [17].

Table 2.5: IPv4 Datagram

Bit	0	4	8	16	19 31	Oct
	Version	IHL	Types of Service	Total Length		4
	Identification			Flags	Fragment offset	8
	Time to Live		Protocol	Header Checksum		12
	Source Address					16
	Destination Address					20
	Options + Padding					24
	User Data Field					

2.2.4 Transport layer

In the Transport layer, the most used protocols are User Datagram Protocol (UDP) and Transport Control Protocol (TCP). Both of them packet-based network protocols. Sometimes referred to as a best-effort protocol, the UDP protocol has no reliability mechanisms. In comparison, TCP offers reliable connection-based transfer of data between endpoints on the network. Also, the TCP protocol offers congestion control to prevent receivers from being overloaded.

In this thesis, the only transport layer protocol covered is UDP.

User Datagram Protocol (UDP)

The primary motivation to use UDP instead of TCP is that queue/buffer storage required to store any data in case of retransmits on the hardware would increase the cost of the system by several magnitudes. This issue also arises due to the sheer amount of data transmitted from the pRU.

Besides the lack of reliability mechanismes, one of the other shortcomings is that if the UDP packet is larger than the Ethernet frame, the IP protocol will split the IPv4 datagram into smaller pieces. One way to avoid this issue is to use jumbo frames covered in section 2.2.2. Additionally UDP does not have a congestion control mechanism like the one that TCP has [18].

Table 2.6: UDP Packet Header

Bit	0	16 31	Oct
	Source Port Number	Destination Port Number	4
	UDP length	UDP Checksum	8
	User Data Field		

2.2.5 Session layer

proton Data Transfer Protocol (pDTP)

The proton Data Transfer Protocol (pDTP) is a novel protocol developed by Ola Slettevoll Grøttvik for the pCT project [19]. The design philosophy behind the pDTP protocol is to give more control of the data transfer between the FPGA and the host computer. As visualized in figure 2.5, the pDTP packages are encapsulated into UDP packets. Previously described in section 2.2.4 is a best-effort without any commands transferred over the network, which means that the pDTP itself must introduce some of these control and reliability mechanisms.

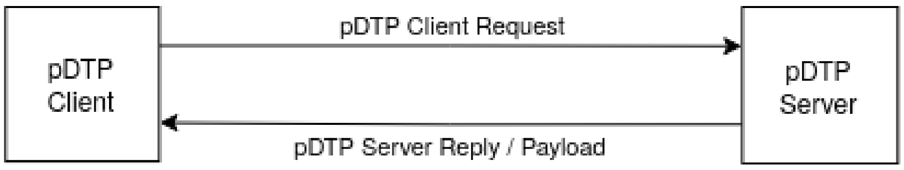

Figure 2.6: pDTP data flow.

Figure 2.6 describes pDTP packages that can contain either of two headers, the client requests and the server replies with payload. The main reason for dividing into two packet types are that the pDTP server only resides as a module in a FPGA and the client is always a computer.

There will be an in-depth explanation of the software design and implementation for a client that handles the pDTP protocol in section 3.1.3.

Client Data Package

The packages sent from the client to the server only contains a header that is 32 bits long. The header specifies a 4-bit Operation code (OpCode), 4-bit flag field, 16-bit command field, and an 8-bit packet size field. The total bytes used for the client header is 8 bytes.

Table 2.7: pDTP Client Packet Header [19][20].

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	IPv4																															
20	160	UDP																															
28	224	pDTP Client Opcode				Flags				pDTP Special Commands																Requested pDTP Packets Size							

The protocol defines eight different client OpCodes, and the software client can use these to instruct the server what kind of instruction it will perform and reply accordingly. When the OpCode is a request for data, it instructs the server to attach data from a buffer.

Table 2.8: pDTP Client OpCodes [19][20].

Opcode	Value	Description
CLIENT_RQR	0x1	A request to the server for a single packet.
CLIENT_RQS	0x2	A request to the server for a stream of between 1 to a maximum of 65535 packets.
CLIENT_RQFS	0x3	A request to the server to initiate a full auto stream, where the server will transmit packages until a CLIENT_ABRT is received from the client.
CLIENT_ERROR	0x4	If a timeout occurs when waiting for a package from the server or a unspecified error.
CLIENT_ACK	0x5	To let the server know that a package has been successfully received by the client.
CLIENT_ABRT	0x6	Interrupts the current operation that the server is performing, i.e. CLIENT_RQFS.
CLIENT_GS	0x7	A request for a status update from the server.
CLIENT_THROTTLE	0x8	Sends a value that throttles the server, in other words makes the server wait a number of clock cycles[1]between sending packages.

The value column in table 2.8 refers to the hexadecimal number is the value set at the start of the header to do the specific command that is requested.

[1]A clock cycle refers to the clock speed of the FPGA that runs the pDTP server module.

Client Throttle Operation code (OpCode)

The Client Throttle feature of the pDTP proposes to solve several issues regarding buffer overflow during the transfer of data on the path from pRU to the computer. The OpCode listed as CLIENT_THROTTLE in table 2.8 lists the term clock cycle, this refers to the clock speed of the FPGA that runs the server.

In the case of the pRU, it can transfer data at a rate of 120MHz×128bit= $15,35$ Gbps from a data buffer to the module that handles the protocol in the firmware. This is greater than the $156,25$MHz ×64bit = 10 Gbps of the Ethernet interface on the pRU. This disproportionality of the clock speeds can cause overwrites of data waiting in the Ethernet buffers when transmission from other internal buffers on the pRU.

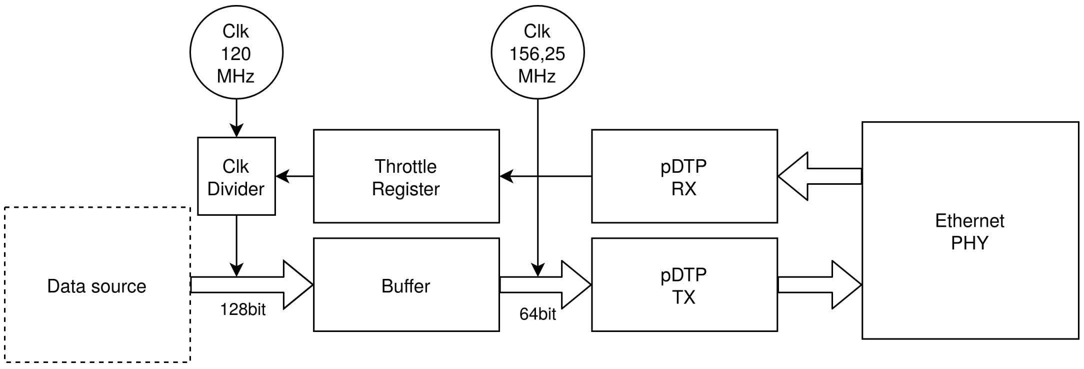

Figure 2.7: pDTP data offload modules in firmware.

The client can create a header with the CLIENT_THROTTLE OpCode with the field WAIT_CYCLES set to the appropriate value. The WAIT_CYCLES value equates to the number of times the 120MHz clock divides down to slow down the data transfer rate into the pDTP firmware module.

Table 2.9: pDTP throttle values and output speed [19][20].

Throttle value	Module Clock [Mhz]	Speed [Gbit/s]
0	120	15,36
1	60	7,68
2	40	5,12
3	30	3,84

Server Data Package

The server packages constructs replies to requests from the client. The package contains a header that is 32 bits, the ABS_TIME field that is 32-bits, and a payload that is from 0 bit up to 32640 bits. So the total package size of 8 bytes up to a maximum of 4088 bytes. The header contains a 4-bit field for the pDTP Server Opcode, a 4-bit field for flags, a 16-bit field for pDTP ID or Buffer Status, and a 8-bit field that indicates the size of the payload.

Table 2.10: pDTP Server Packet Header [19][20].

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	IPv4																															
20	160	UDP																															
28	224	pDTP Server Opcode				Flags				pDTP Packet ID / Buffer Status																Actual pDTP Packet Size							
32	256	ABS_TIME (Server Clock Cycles)																															
36	288	Payload (0 - 255 pRU words)																															

The protocol defines five different server OpCodes, which will help the client identify what kind of information the client has received.

Table 2.11: pDTP Server OpCodes [19][20].

Opcode	Value	Description
SERVER_WRITE	0x0	A singel packet reply to a CLIENT_RQR request.
SERVER_STREAM	0x1	A singel packet in a stream of 1 to 65535 packets that is a reply to a CLIENT_RQS/RQFS.
SERVER_ERROR	0x2	A packet indicating that a error on the server has occurred. This can be in response of one of several errors. The server timing out when waiting for a ACK reply from the client, a error processing a packet from the client, or that there is no data available(Based on what special flags that is set in the request from the client)
SERVER_EOS	0x3	The server has finished transmitting a stream request or that there is no more data available (depending on what special flags have been set).
SERVER_STATUS	0x4	A packet containing a status update from the server. The size of this packet is 128 bit / 16 byte.

Limitations of the proton Data Transfer Protocol (pDTP)

There are some limitations of the pDTP in RQR mode the maximum packet size is limited to 4088. In RQS mode the limitation is the number of packets that can be requested, which is 4093 pRU packets in addition to the issue with RQR that packet size is limited.

Payload - pRU Data Format

Data being read out from the system gets packaged in the pRU Dataformat. One pRU word is 128 bit long, it can contain one of 5 types. DATA_WORD, TAG_HEADER_WORD, TAG_TRAILER, TAG_EMPTY_WORD or a DELIMITER_WORD.

Table 2.12: pRU Data Format.

Field name	WORD_TYPE	RU_ID	STAVE_ID	CHIP_ID	CONTENT
Length [bit]	2	6	4	4	112
Bit placement	127:126	125:120	119:116	115:112	111:0

IPbus Control Protocol (IPBus)

The IPBus protocol is a control system developed for a Large Hadron Collider upgrade at CERN. It utilizes UDP protocol to transfer READ and WRITE commands in hardware registers. In the pCT system, it is used for a host computer to communicate with registries on the pRU. In comparison to the pDTP interfaces, IPv4 addresses on the IPBus interface has the option to be set with DHCP, and it also implements a form of Ping functionality. The IPBus system consists of three parts; one the firmware module, second the ControlHub, and third the micro Hardware Access Library (uHAL). This thesis will only cover the ControlHub and the uHAL since firmware is outside the thesis's scope.

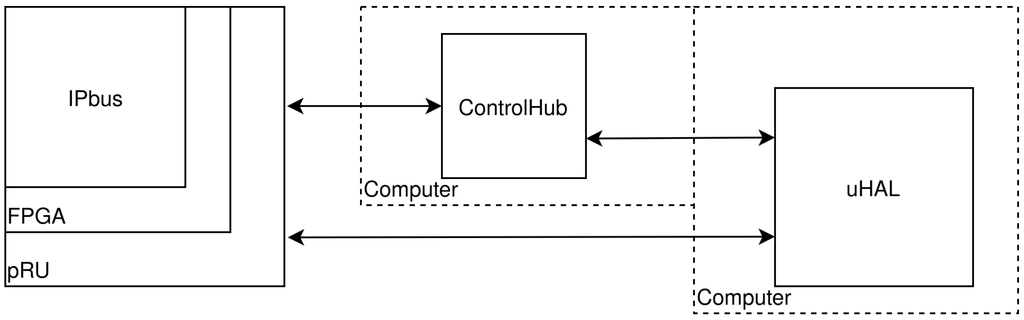

Figure 2.8: IPBus overview.

The ControlHub software implements flow and reliability mechanisms to the IPBus system and the use of the ControlHub one-to-many or many-to-many communication between pRU and host computers.

uHAL is a interface that can be used in either C++ or in Python code [21].

2.3 Linux Operating System (OS)

Ever since its creation in the early 1990s Linux has been a powerful and highly customizable operating system. It comes in several distributions that have both strengths and weaknesses, today Linux is used both for desktop and servers. Linux differs very much compared to proprietary OS like for instance Microsoft Windows, in that all the source code for all the parts that make up the OS is available and can be modified to meet the needs of its users. In this thesis the distribution that will be covered is the CentOS distribution [22].

What happens in an OS can be split into two parts referred to as User space where processes that are running on OS and Kernel Space which mediates the interaction between user space and the hardware that the OS is running on.

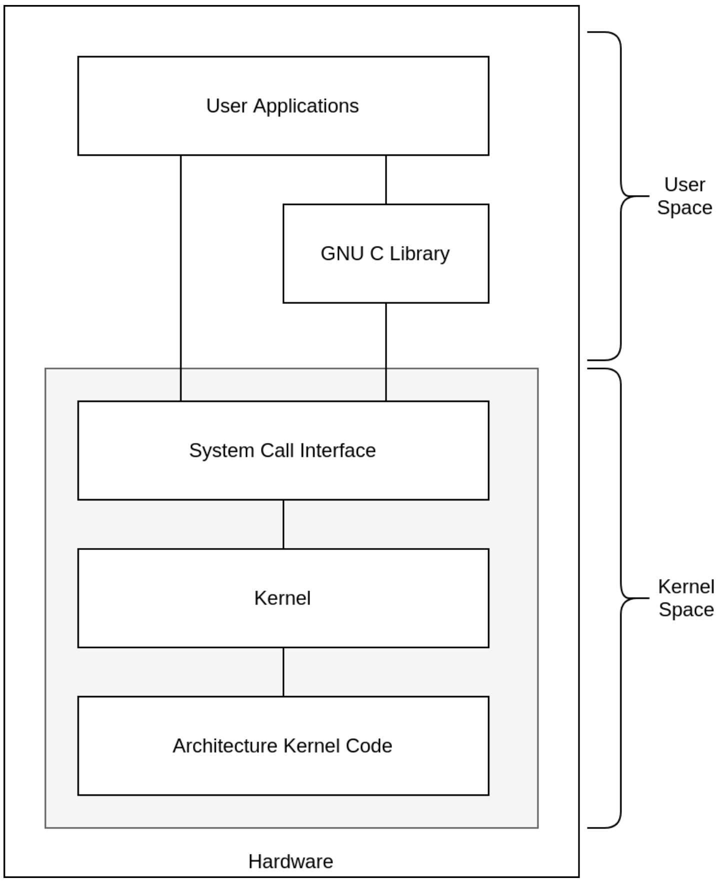

Figure 2.9: Linux layers

2.3.1 Linux Kernel

The kernel is the most central part of most OSs. In Linux the kernel has 4 main functions: memory management, process management, device drivers, system calls and security.

Context Switch and CPU Affinity

In a modern OS like the one that is used in the system, processes and tasks can easily be shifted between cores in order to multitask and balance all the tasks to run on available cores with low utilization.

Interrupt request (IRQ) and IRQ Affinity

At times it can be beneficial for both hardware and software to grab the attention of the CPU. Interrupts are handled through a Advanced Programmable Interrupt Controller. These controllers are cpu architecture specific.

2.3.2 Linux Network Stack

The Linux Network Stack must not be confused with the whole network stack presented earlier in this thesis, the Linux stack can be defined from the hardware drivers up to the user space through the kernel space.

Queuing Disciplines

There exist several different queuing disciplines. The most used in the Linux network stack is the First-In-First-Out (FIFO). When packets arrive in the NIC it is put in a driver FIFO, the data itself is put in socket kernel buffers (sk_buff), while the FIFO only contains descriptors to the sk_buffs. These sk_buffs will be passed through the different layers of the stack until it reaches user space [23].

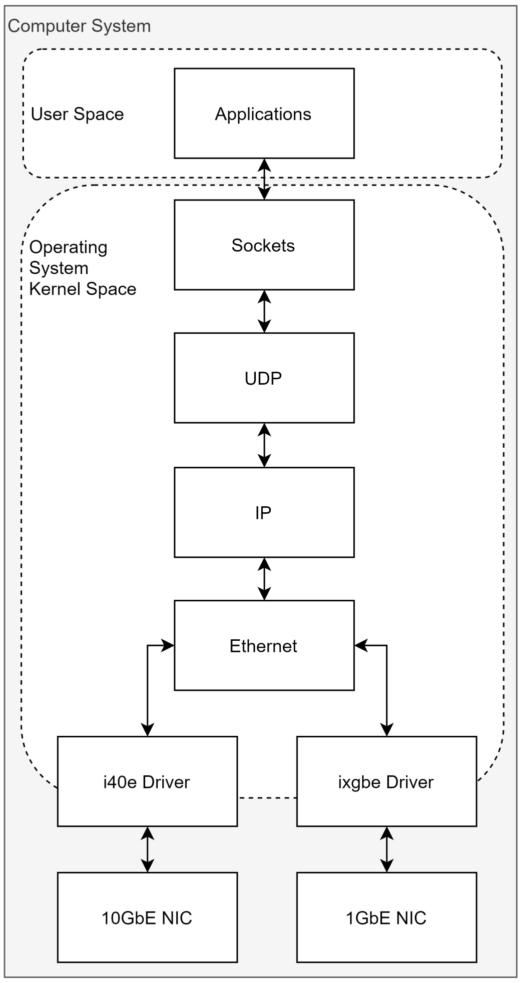

Figure 2.10: Linux network stack.

2.4 Related Work

A large project as the Bergen pCT project is formed during several years of development, both from doctoral and master theses, in addition to the work of researchers and academic staff. In addition, several European institutions have contributed to the project through several disciplines as mechanical and electrical design and production.

This thesis is related to several previous master theses. There is also a doctoral thesis that is directly related to this thesis.

Design of High-Speed Digital Readout System for Use in Proton Computed Tomography [24]

When the project decided to select the ALPIDE as the main component in the detector, Ola Slettevoll Grøttvik designed a readout system and implemented a hardware design for FPGA.

Ethernet-Based Control System and Data Readout for a Proton Computed Tomography Prototype [25]

The thesis work by Karl Emil Sandvik Bohne is the basis for the embedded system with Ethernet support used in the first iteration of the prototype, and the PTB embedded system is a modification of that. Section 3.1.2 will introduce the production test repository, which is also a modification of Bohne's work.

Data Acquisition and Testing Software for a Proton Computed Tomography System [26]

Håkon Andreas Underdal designed and implemented in this thesis a rudimentary software to handle the readout from the first iteration of the prototype.

Scalable Readout for Proton CT [27]

This thesis by Øistein Jelmert Skjolddal was written mostly in parallel with the authors thesis work. It covers the development of a data parser for the pRU and the ALPIDE data formats.

High-Speed Signal and Power Distribution of a Digital Tracking Calorimeter for Proton Computed Tomography [28]

Tea Bodova thesis covered both design of hardware for the pCT detector and a small software contributed to the control of power supplies that were to be used in conjunction with the production test scripts.

Design and Implementation of a High-Speed Readout and Control System for a Digital Tracking Calorimeter for proton CT [19]

Ola Slettevoll Grøttvik has also completed his doctoral degree. Many of the topics in his thesis have laid the foundation for the authors' thesis results. Grøttvik is the designer of the pDTP protocol and the developer of the embedded system on the FPGA.

This embedded system is an integral part of aiding in developing the readout software presented later in this thesis.

Software Design and Architecture in Bergen pCT Project

Researcher Matthias Richter has provided a great deal of input for the architecture of software in the Bergen pCT project. Richter has developed the structure for the pCT-Online repository and provided the data structure and many of the templates used in classes in the repository.

2.5 Methodology

The work that lays the basis for this thesis has been performed at the Microelectronics Laboratory at the Institute for Physics and Technology, University of Bergen. From the start of the project, the work has been both quantitative and qualitative. There has been an ongoing qualitative evaluation of the design and implementation of the software. Likewise, there has been performed quantitative analysis of the performance of the software in connection with the hardware communication.

CHAPTER 3

Design and Implementation

3.1 The proton Readout Unit (pRU) prototype readout chain

3.1.1 Previous pRU Prototype System

In large technical projects, there is a risk of needing to redesign hardware. These changes propagate into changes in the architecture and further into code. In software, the term technical debt is becoming more and more relevant as projects become increasingly complex [29]. The pCT project is no different from a pure software project in the respect that it can contain several ”debts” in both its hardware design and the software that depends on it. The project has gone through many iterations, producing both legacy hardware and code.

The main difference between the current prototype system and the old system is an embedded Real Time Operating System (RTOS) running on a softcore processor within the FPGA that had both the control system and the data offload running over TCP. The main reason to abandon the concept of a Micro Controller Unit (MCU) or a softcore was the slow and unreliable offload speed and the high latency of the control system. The option of a system that runs everything in the FPGA fabric was more favorable. The change in architecture increases the speed of the individual modules. However, this also increases the complexity of the system due to it being harder to develop firmware for FPGA than an embedded RTOS.

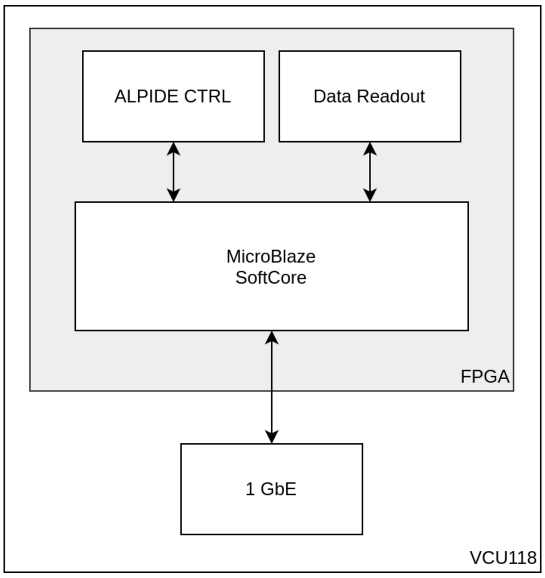

Figure 3.1: Old prototype of control and readout

3.1.2 Current pRU Prototype System

The prototype system can be split into three main parts, the ALPIDE test string, the pRU implemented on a Digilent VCU118 development board, and a host system (Computer).

As introduced in section 1.2.1 the finished detector is divided into 41 layers, one layer containing 12 strings of 9 chips each. For each layer there is going to be a pRU. The pRU is a Printed Circuit Board (PCB), which main component is a Field Programmable Gate Array (FPGA). This FPGAs responsibility is to provide registries and buffers for control and data transfer. The FPGA's main interface for transferring data from the pRU can be configured in one of two configurations, either a 40 Gb/s Quad Small Form-factor Pluggable (QSFP+) that can be split split into 4 x 10 Gb/s Small Form-factor Pluggable (SFP+) or a single 10 Gb/s SFP+.

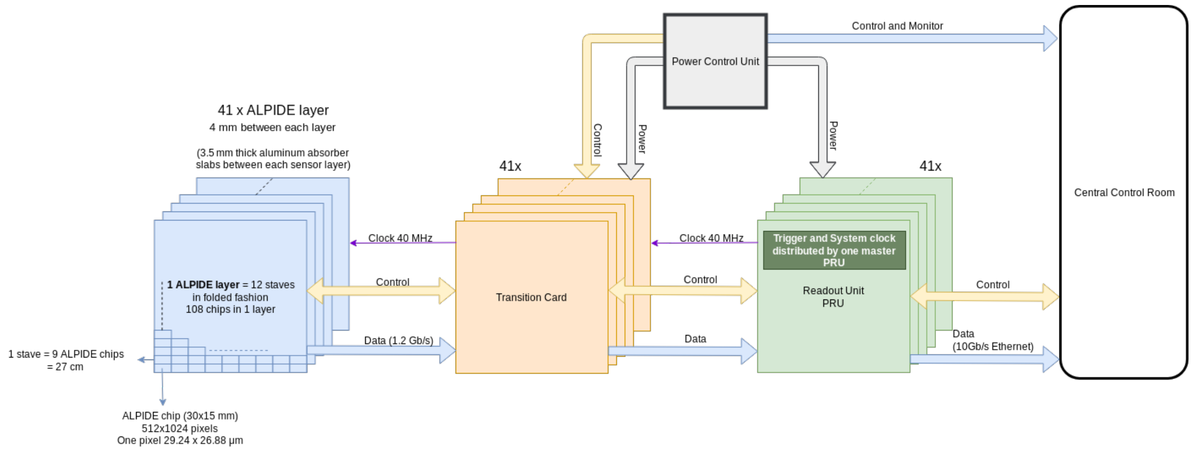
41 x ALPIDE layer
4 mm between each layer
(3.5 mm thick aluminum absorber slabs between each sensor layer)
1 ALPIDE layer = 12 staves in folded fashion 108 chips in 1 layer
1 stave = 9 ALPIDE chips = 27 cm
ALPIDE chip (30x15 mm) 512x1024 pixels One pixel 29.24 x 26.88 µm
Clock 40 MHz
Control
Data (1.2 Gb/s)
41x
Transition Card
Control
Power
Power Control Unit
Control and Monitor
Power
41x
Clock 40 MHz
Control
Data
Trigger and System clock distributed by one master PRU
Readout Unit PRU
Control
Data (10Gb/s Ethernet)
Central Control Room

Figure 3.2: Proton readout chain [28].

The proton Readout Unit (pRU)

The proton Readout Unit (pRU) is being developed on a evaluation card Digilent VCU118, its main component is a Xilinx FPGA.

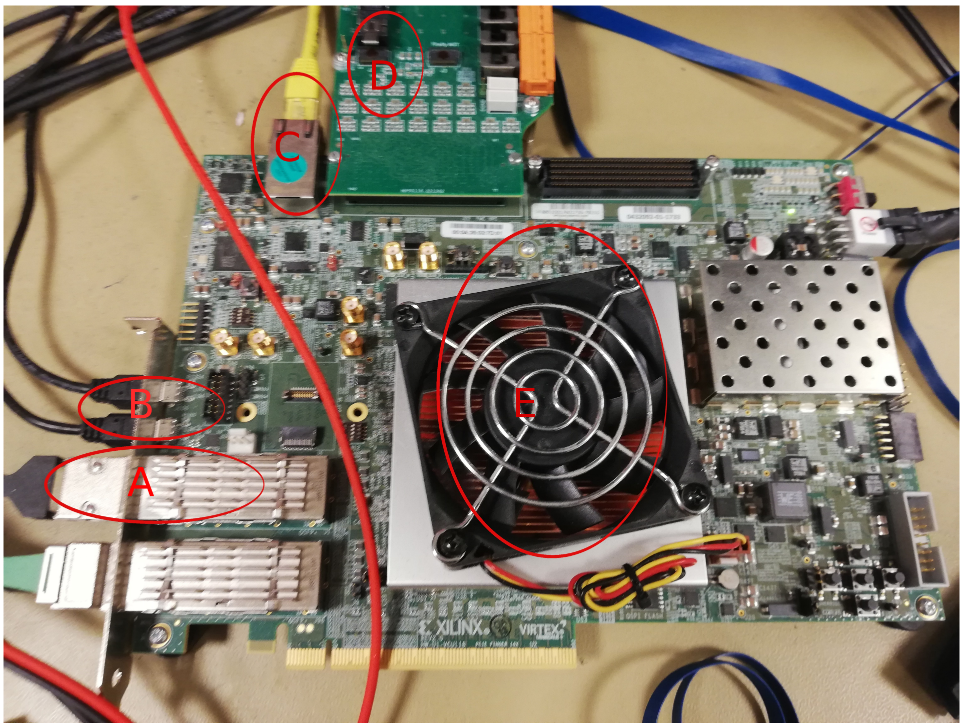

Figure 3.3: VCU118 Development board. A: 10 GbE data offload, B: USB debugger, C: 1 GbE for control, D: Interface for ALPIDE string, E: FPGA.

Since the data offload in the current prototype uses 10 GbE, there is a need to reduce overhead in the data transfers. As introduced in section 2.2.2 the frame size can be increased. The resulting change increases the size from 1538 to 4096 bytes. The reasoning for not increasing it up to the maximum of 9000 bytes is that resources in the FPGA fabric are minimal. A change like this would exceed the budget allocated for this specific module as the buffers required would be too large.

Production Test Box (PTB)

The Production Test Box (PTB) has been designed and built at the Department of Physics and Technology at UiB to aid the manufacturing facility in LTU Ltd in Kharkiv, Ukraine. This facility will handle the bonding process of the ALPIDE-chips to the aluminum plates. The PTB runs in a setup resembling the old prototype system, as it runs a MCU softcore. The box itself has its own PCB board with a separate FPGA module. This box has two different physical interfaces that can communicate with the ALPIDEs, one where individual chips can read or the same string interface as the pRU. The PTBs main interface is a 1 GbE network interface. It also has a USB interface for debugging purposes of the embedded software. The PTB transfers the offload data over TCP rather than UDP like the pRU.

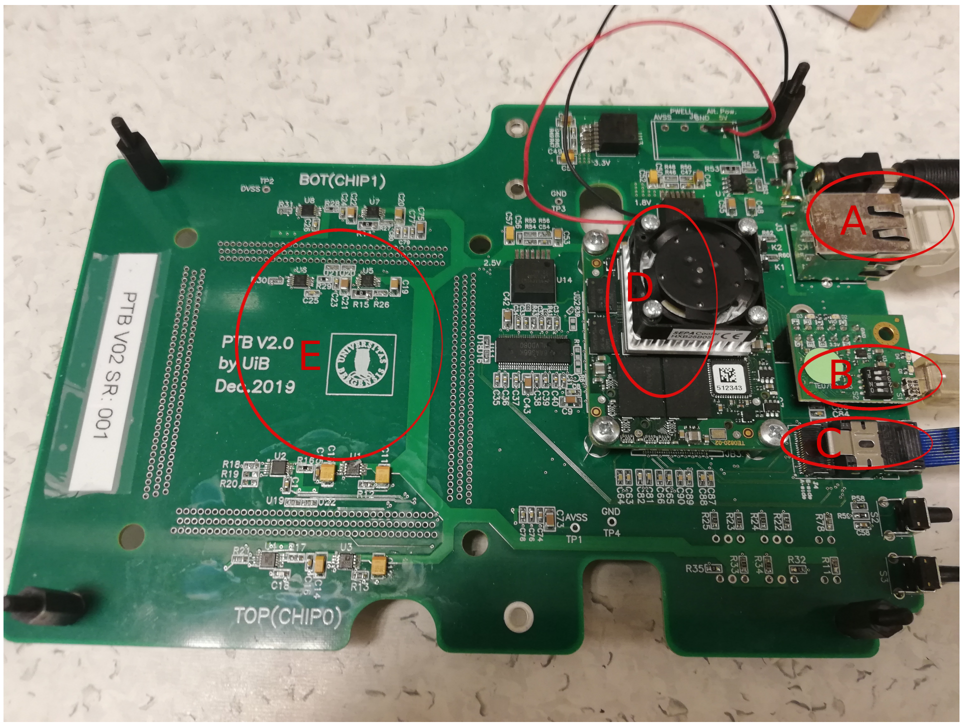

Figure 3.4: PTB. A: 1 GbE data offload and control, B: USB debugger, C: Interface for ALPIDE string, D: FPGA, E: Interface for single ALPIDE (Not mounted).

Production Test Software

The production test software is a repository containing a collection of Python scripts developed to be used together with both the pRU and the PTB hardware to evaluate the quality and performance of the ALPIDE-chips. This software package utilizes the legacy software developed for the old prototype. To accommodate for the different types of hardware architecture used in the pRU and PTB the repository contains a Board Support Package (BSP) folder. The BSP is a collection of scripts that facilitates communication with the specific hardware. For the pRU the pDTP and the readout processor are replacing the data offload parts of the legacy software. In addition, replacing the control part of the legacy software is done by the IPBus package, while the PTB still uses the legacy software.

3.1.3 Computer Test System

The computer is a Lenovo workstation with a Xeon CPU architecture, 64GB Error Correcting Code (ECC) Read Access Memory (RAM). The computer has 3 Ethernet interfaces, two on-board 1 Gigabit Ethernet (GbE) and one Intel X710-DA2 Converged 10GbE card.

Intel X710-DA

This is a network card with two 10GbE SFP+ converged ports, SFP+ which has been described in section 2.2.2. The X710 uses the i40e driver for Linux, the i40e driver provides several different advanced settings that can be tuned to get increased performance, some of these are as following, but not limited to:

- Setting IRQ affinity.
- Interupt moderation.
- Ring size.
- Disabling flow control.
- Queue steering.

If several pRUs are going to be handled by the same computer, it will be beneficial to set up the NIC with queue steering. When using queue steering, the NIC will keep a queue for each of the pRU which will get each own queue. This feature will give more queue length for each pRU and reduce the chance of packets get overwritten by the ring buffer.

Operating System (OS)

On the computer the OS CentOS 7 with kernel 3.10 is being used, this OS is a long time stable operating system.

3.2 Implementing the proton Data Transfer Protocol (pDTP)

3.2.1 Client Configuration

As the complexity of a system grows, more settings are needed to communicate and operate correctly. Examples of this are a IPv4-address, port number, and package size, to name a few. To prevent essential options from being hardcoded into the source code, settings like these can be read in and set in two different ways. One of them is by reading in parameters when calling the executable from the command-line, another more flexible way is to use a configuration file in a standardized format. Adding this functionality can efficiently be done through the Boost Program Options library.

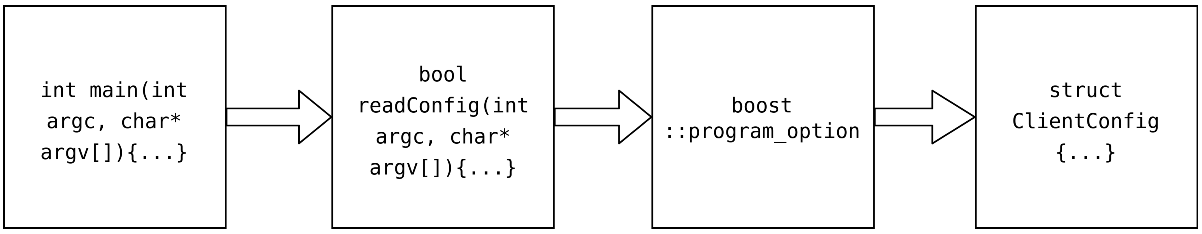
int main(int argc, char* argv[]){...}
bool readConfig(int argc, char* argv[]){...}
boost ::program_option
struct ClientConfig {...}

Figure 3.5: Flow on how to load a configuration from a file.

When executing the program with the file name as a parameter, the file name can be passed to a function that loads the content into a `std::ifstream`.
Using `boost::program_options::variables_map` and `boost::program_options::store` the data will read from the stream into a struct that contains variables for each option in the configuration file.

3.2.2 The User Datagram Protocol (UDP) Client

The source and sink of data from and to the network in a userspace network server or client, is a socket. In this particular case, it is an UDP socket. A socket is either setup as a server, also called a endpoint, or as a client. A socket can be configured with port number, what IPv4 address it will use or time out settings to name a few. When everything has been configured in the socket the next step is to open it for communication.

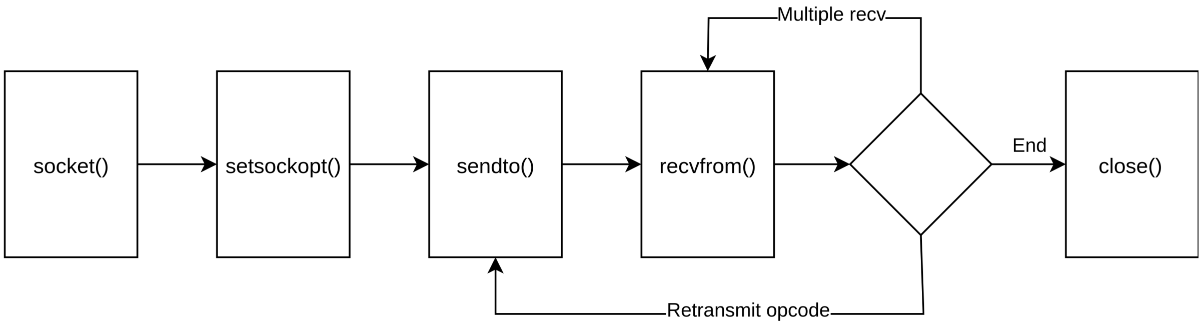

Figure 3.6: UDP Communication Flow in the ComService Class.

This client class is neatly named ComService, separating this part of the code into its own class. Doing this has several advantages, at first it is easier to troubleshoot any errors that happen between the OS and the rest of the code. However, the most crucial feature is however that the designer can employ a technique called Behavior-Driven Development (BDD) testing. BDD entails that you swap out the class for a mock class that inherits from the same base class as the ComService class.

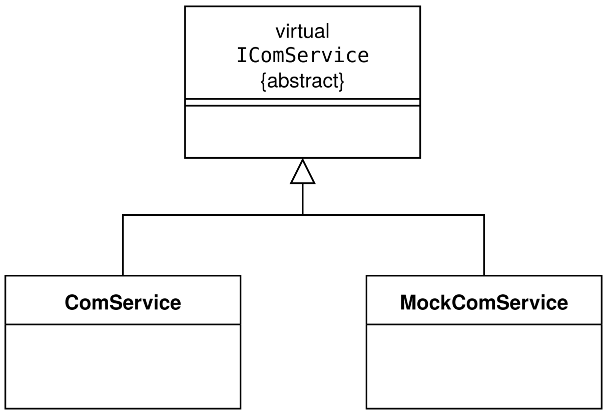

Figure 3.7: Implementation of the classes, ComService and MockComService.

As introduced in section 2.1.5, the dependency-injection pattern, the concept of constructor injection, is used to inject the dependency(ComService) upon creating the pDTP-Client. More about the pDTPClient class in section 3.2.4.

3.2.3 proton Data Transfer Protocol (pDTP) Data Model

When receiving and transmitting data from the UDP-socket, the data is normally written into a C-style array of type char, e.g. `char msg[] = 0x20 , 0xFF , 0xFF , 0xFF;`. When transmitting this array to the pRU, the pRU will reply with a stream of 65534 packets with 4088 bytes each. Rather than using this raw approach, it is more favorable to split off what the header contains from the rest of the payload. Doing this will deal with raw bits contained in a char data type using bit operations and masking to read out the information contained in the header. This approach might invoke unintended behavior when explicitly casting from one type to another.

A more sustainable approach would be to create custom data types to represent the pDTP server request and client replies. Matthias Richter has created a data model and realized it to facilitate this. Implementing this data model for pDTP protocol into a data type is done with a struct class, inside the union declaration it is used to combine each field of the package.

```
union {
  uint32_t raw = 0;
  struct {
    uint32_t flags : 4;
    uint32_t opcode : 4;
    uint32_t nofpackets : 16;
    uint32_t sizeofpacket : 8;
  };
  struct {
    uint32_t unused_msb_thr : 8;
    uint32_t wait_cycles : 24;
  };
  struct {
    uint32_t no_wait : 1;
    uint32_t maxi : 1;
```

```
      uint32_t min_req : 1;
      uint32_t no_ack : 1;
      uint32_t unused_lsb_rr : 28;
    };
   struct {
      uint32_t resend : 1;
      uint32_t timeout : 1;
      uint32_t uninterpretable : 1;
      uint32_t unused_lsb_error : 29;
    };
  };
static_assert(sizeof(pDTPClientRequest) == 4);
```

Listing 3.1: Excerpt of the union in the `pDTPClientRequest` data type

As the last line indicates, assert assures that the struct does not exceed the limit of 32 bits (4 bytes) which is the size of the pDTP client header.

The first field in the union sets all the bits to zero, the numbers behind the variable denotes how many bits each field will use.

```
  std::vector<char> rawreq(sizeof(pDTPClientRequest));
  pDTPClientRequest& request =
    *reinterpret_cast<pDTPClientRequest*>(rawreq.data());
  auto opcode = static_cast<ClientOpcode>(currentOpCode);
```

Listing 3.2: Example on how to convert a pDTP-header.

When converting the data from the raw request to the `pDTPClientRequest` data type, the `reinterpret_cast` conversion is used. This allows for a compile time instruction that the rawreq vector is the type of `pDTPClientRequest`. Afterwards the rawreq vector can be passed off into the ComService member function transmit to be passed on to the socket and finally over the network through the OS.

3.2.4 The pDTPClient Class

The responsibility of this class is to handle the communication over the pDTP protocol. The constructor of the class accepts the config file presented in section 3.2.1 and also the `IComService` presented in section 3.2.2 in this chapter.

To pass the dependency of the ComService object, it is injected into the class through the constructor as a pointer.
`pDTPClient(ClientConfig confp, IComService* coms);`

The primary public member function is `getDataFromQueue()` which returns data that have been read out from the payload of the pDTP packages.

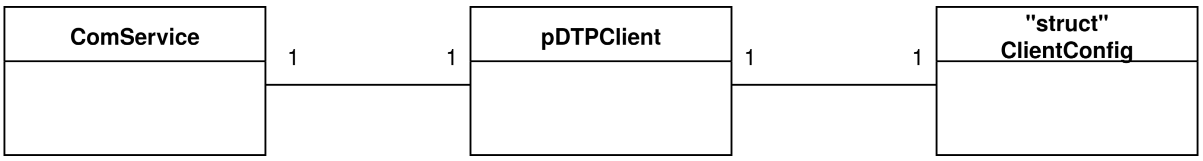

Figure 3.8: Implementation of the pDTPClient class.

Implementation of The pDTP State Machine

As introduced in section 2.1.3, the state machine modeling can be a great tool to design and develop software. To implement the communication between the pRU and the computer, a simple state machine model can be developed from a table into a model and finally constructed into code.

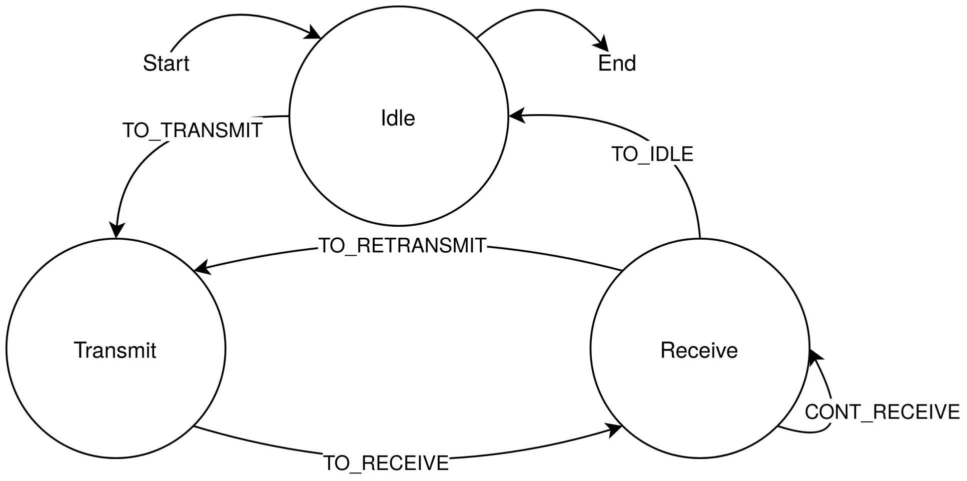

Figure 3.9: Model of the pDTP state machine.

The possible states, transitions and events are best documented in a table as in the example given in table 2.1. As for the pDTPClient class the model was developed from table B.1.

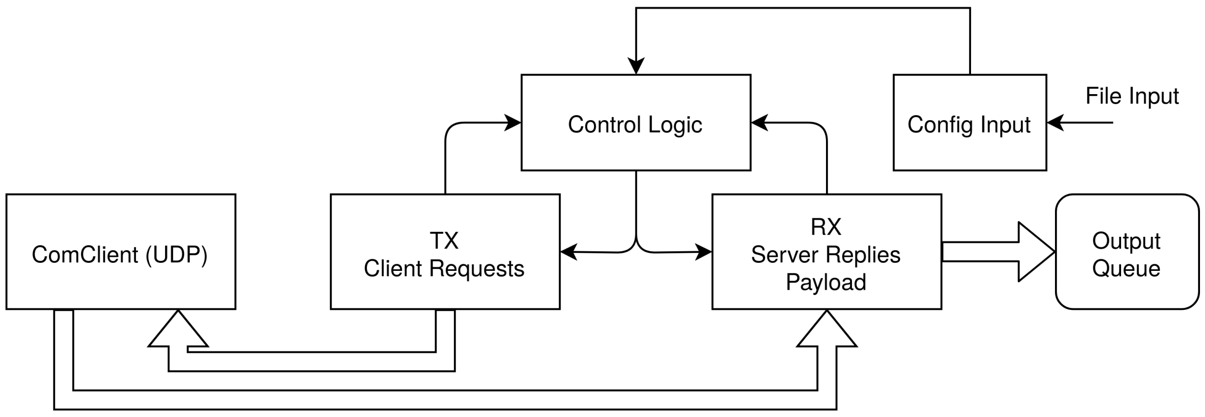

Figure 3.10: Control and data flow in the pDTP Client.

Client Statistics

Sometimes there is a need to measure the throughput of data coming from the pRU system, this can be achieved through different member functions that the pDTPClient class offers. `getMissingPacketErrors()`, `getPacketIdRecved()`, `getBytesPrSec()`, `numberOfElementsInQueue()` can be used to monitor and diagnose problems when the client is running.

Use of Atomic Variables

To ensure that one thread does not access a variable when another writes to it, the programmer can employ the concept of atomic variables from the `<atomic>` header, and this feature has been available since C++11 version. A boolean can be declared atomic by `std::atomic<boolean>`, the result is no undefined behavior from the use of this variable.

3.2.5 Endian Conversion

At times it is beneficial to keep track of what order pDTP packages arrive in, especially during development and debugging. This can be done manually with tools such as WireShark [30], but it is more convenient to implement it in the source code itself. Table 2.10 that lists the pDTP server header, and there is a field called pDTP Packet ID. This 16-bit field will increment each time a packet is sent from the pRU. The field is from bit 8 to and including bit 23 of the header. Due to endianness as introduced in section 2.2.1, the pDTP protocol is different to that of the host systems CPU architecture. Updating the package counter member field in the `pDTPClient` class, it is not a simple extraction of the `packetid` member of the `pDTPServerReply` data-type.

Conversion can easily be done with the Boost Endian library, which offers the ability to convert using byte swapping intrinsics available in the GCC compiler. Implemented intrinsics is a builtin function that maps to CPU instructions. This results in small and fast code when the optimization flag is used when compiling [31].

```
pDTPServerReply& reply;
uint16_t packetID = boost::endian::endian_reverse(uint16_t(reply.
    packetid);
```

Listing 3.3: Example on how to convert endianness.

When `reply.packetid` contains the value `1000 0000`, then after being passed to the `endian_reverse()` function. This function will return and assign the variable `packetID` the value `0000 0001`.

3.3 The Production Test Box (PTB) Software

Both the design and implementation of the PTB server class has been implemented more true to the principles of FSM when compared to the pDTP client class. As the states and events of the FSM are implemented in two switch/case structures. The switch input is based of a enum class.

When compared to the software for the pRU that uses a pDTP client to transfer data, the PTB software uses a standard TCP server to handle the data offload. Here it is PTBs embedded RTOS that acts as a client to connect to the software running on the computer.

Table 3.1: FSM property table for the pTBServer class.

State	Action	Event	Transition
IDLE	none	START	to_starting
STARTING	spawn thread	auto	to_running
RUNNING	read_some() && push_data()	STOP	to_stopping
		!STOP	to_running
STOPPING	socket.close() && thread.join()	auto	to_idle

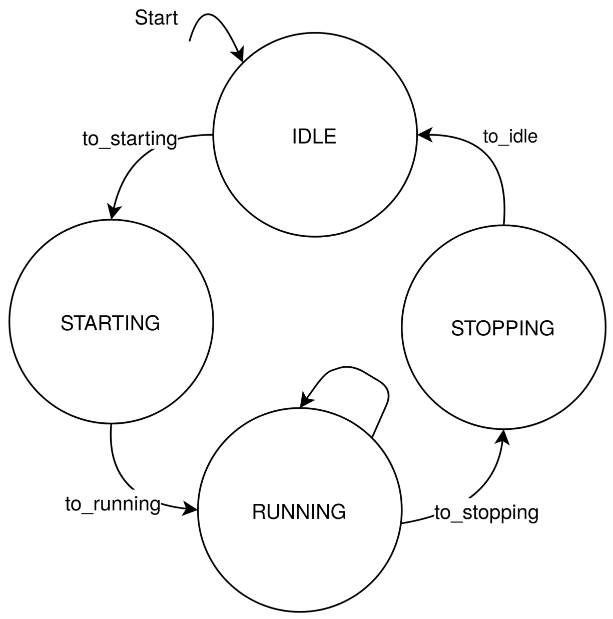

Figure 3.11: Model of the PTB state machine.

```
enum class State {
  IDLE,
  STARTING,
  RUNNING,
  STOPPING,
};

enum struct Command {
  START,
  STOP,
};
```

Listing 3.4: Enum classes with states and events.

Control Commands

With the PTB there are some special considerations to be taken since the embedded system operates both the control and data offload systems. Before preparing the readout software to handle TCP communication, the PTB server needs to open a port on `29070`. The control part of the embedded system needs to receive the value `0x48` on the `49153` port, and this will start an offload thread that connects to the server socket on the computer. The PTB embedded readout system will start to transmit the data available. When communication is finished, the value 0x49 is sent to close the socket and join the thread back to the main thread.

3.4 Readout Session Processor

From the start, the readout software was designed modular with use of the policy based design methodology introduced in section 2.1.2. To tie the process of network readout, parsing and file writing together, the `ReadoutSession` has been designed and implemented with three policies input, forward and output.

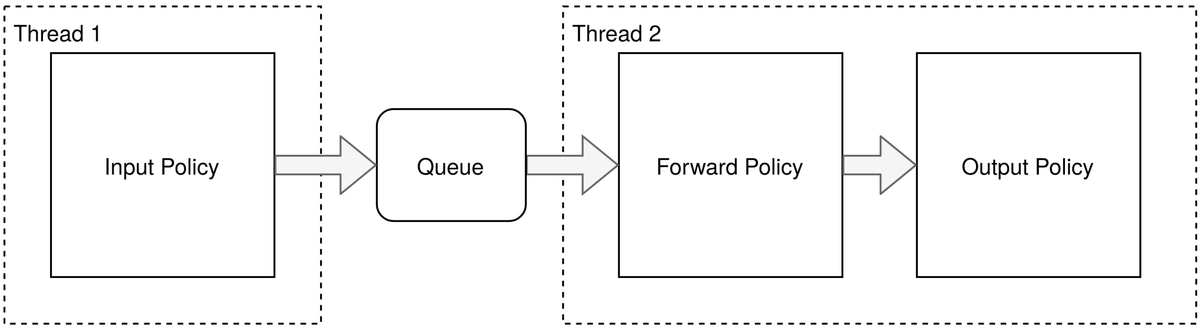

Figure 3.12: Readout Session running in two different threads.

The `ReadoutSession` is declared with `std::function<std::vector<char>>`, this type of function is a class template for a function wrapper. The wrapper accepts functions and lambda expressions, they can be both passed as a copy and as `std::move`.

```
using BufferT = std::vector<char>;
using Processor =
        readout::ReadoutSession<std::function<BufferT()>,
                                std::function<BufferT(BufferT &&)>,
                                std::function<void(BufferT &&)>>;
Processor pipeline(fileReader, forwardPolicy, fileWriter);
```

Listing 3.5: Readout Session Processor

A lambda function can then be declared

```
auto fileReader = [&ifilename, &inputFormat]() -> BufferT {
    std::istream* input = &std::cin;
    std::ifstream inputFile;
    if (ifilename != "-") {
      inputFile.open(ifilename);
      if (!inputFile.is_open()) {
        std::cout << "can not open file '" << ifilename << "' for
   reading" << std::endl;
        exit(1);
      }
      input = &inputFile;
    }
```

Listing 3.6: Readout Session Processor

Policies

Each of the three types of policies can be mixed and matched with any other types of policies. The input policies do not accept any input parameters. Instead, they will acquire data from a file (fileReader) or a protocol handler (pDTPClient or pTBServer). When the data is received, they will return it into a `std::vector`.

Table 3.2: Input Policies

Input Policy	Parameter	Return
fileReader	none	std::vector<string>
pDTPClient	none	std::vector<string>
pTBServer	none	std::vector<string>

The forward policies both accept and pass `std::vector`. The forwardFilter will not change the data in any way and simply hand it over to the following policy. The pRU-Filter, on the other hand, will decode the RU data format before returning the data.

Table 3.3: Forward Policies

Forward Policy	Parameter	Return
forwardFilter	std::vector<string>	std::vector<string>
pRUFilter	std::vector<string>	std::vector<string>

The output policies will only accept data from the preceding policy. The fileWriter will write the data to a binary file, while the rootWriter will create a file in the root format that the root software can read.

Table 3.4: Output Policies

Output Policy	Parameter	Return
fileWriter	std::vector<string>	none
rootWriter	std::vector<string>	none

3.5 Inter-thread Communication

To keep up with the demand by the pRU and the fact that the data processing part of the software is much slower than the high incoming rate. There is a need to parallelize the execution of the receiving end. The solution to this can be achieved by running several threads at the same time. Several issues arise with such a solution, mainly data races and deadlock situations. Data races are when one or more threads read or write upon the same data in memory. One solution that the programmer has at their disposal is to use a mutex lock to prevent this type of scenario, but this again introduces an issue referred to as deadlock, which is when two threads want to access the same shared resource. These mechanics are something that ensures what is referred to as thread safety.

3.5.1 Thread Safe Containers - Single Producer Single Consumer Queue (SPSC)

In the Boost Lockfree library there is an implementation called a `spsc_queue`. This type of queue is a Single Producer Single Consumer Queue (SPSC). This means that there is only only one thread that is allowed to add data to the queue, likewise it is only one thread that is allowed to consume data from the queue.

`boost::lockfree::spsc_queue<T>`[1] requires that the elements type T to have a default constructor and that T is copyable.

```
boost::lockfree::spsc_queue<std::vector<char>> spscQueue{100};

bool addData(std::vector<char> data)
```

[1]T refers to a generic type.

```
{
    return spscQueue.push(data);
}
bool getData(std::vector<char>& data)
{
  return spscQueue.pop(data);
}
```

Listing 3.7: Example on how to use the boost::lockfree::spsc_queue.

`boost::lockfree::spsc_queue` provides several public member functions. `bool push(T const & t)` that is used to enter data into the queue, and `bool pop()` to get data out of the queue. The SPSC queue is implemented as a ring buffer. This kind of buffer inserts(push) the data at the tail end in one thread and retrieves(pop) data at the head of the queue in the other.

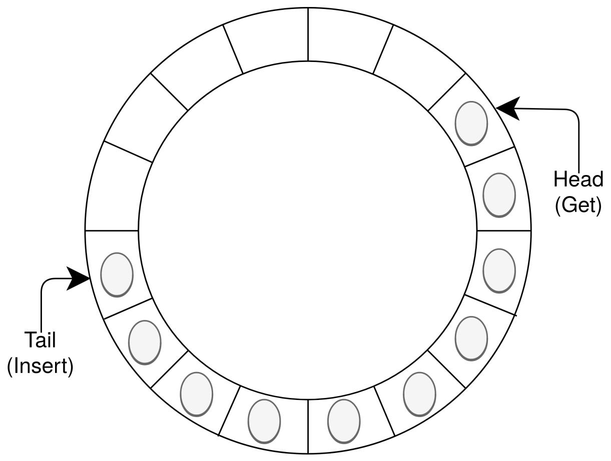

Figure 3.13: Circular Buffer with tail and head.

An advantage of the SPSC queue is that it is lock-free, sometimes also referred to as wait-free, meaning that no mutex or locks are guarding the insertion or retrieval of data. The producer thread can insert data into the queue without waiting to acquire mechanisms that ensure thread safety in other types of queues. The same applies to the consumer thread. To ensure that the transfer of data between the two different threads through the SPSC queue is safe, the programmer must take care to only call `push` and `pop` from their respective threads.

One of the drawbacks is that if the SPSC queue is too small, the number of elements, in respect to the rate the data gets pushed compared to the popped rate, something that might increase the risk of data being over-written by new data, and the head going past the tail end. To prevent the loss of data from occurring from this issue, an option is to increase the queue size so that it holds a sufficient number of elements. Another drawback is that the memory allocated for the queue gets statically reserved at the program execution.

3.6 Inter-process Communication

When designing modular software there is sometimes a need for several processes to communicate with each other. Unlike when several threads of the same program are running in parallel to each other within the same process, the issue can be solved as explained in section 3.5.

As mentioned earlier in this thesis there are several control and tests that runs in Python scripts. Its desirable to stop and empty buffers to make sure that all data has been passed trough the readout software before stopping the Python scripts.

This can be achieved using the Portable Operating System Interface (POSIX) and the Boost ASIO local sockets or sometimes refereed to as domain socket. In contrast to the UDP sockets described in 2.2.4, domain sockets (SOCK_DGRAM) binds to a filename in the local filesystem [32].

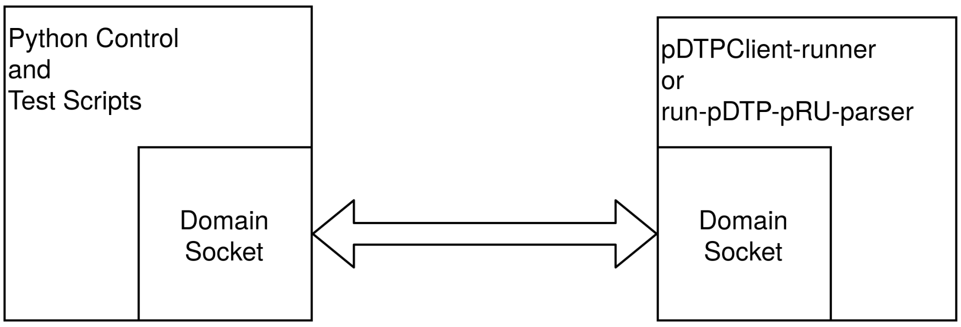

Figure 3.14: Domain socket communication between two programs.

The control software can then use the same socket operations as regular network sockets to transmit and receive data.

```
1  boost::asio::io_context domain_io_context;
2  boost::asio::local::datagram_protocol
3  ::endpoint ep("/tmp/PCTDOMAINSOCK"); //Create endpoint to a local domain socket
4  boost::asio::local::datagram_protocol::socket socket(domain_io_context, ep); //Set context and endpoint, opens socket
5  char buffer[1024];
6  boost::system::error_code ignored_error;
7  socket.receive_from(boost::asio::buffer(buffer), ep);
8  /**
9  Enter actions here
10 */
11 socket.close();
12 ::unlink("/tmp/PCTDOMAINSOCK"); // Remove binding so that if another user uses the application there wont be any lingering files related to the socket.
```

Listing 3.8: Implementation of ASIO domain communication in C++.

The receive_from call on line 7 in listing 3.8 will block the execution in a separate thread. This call will wait for data to be received on the domain socket. Then an appropriate action can be handled trough safe notification to the other threads in the program.

```python
if os.path.exists("/tmp/PCTDOMAINSOCK"):
    client = socket.socket(socket.AF_UNIX, socket.SOCK_DGRAM)
    client.connect("/tmp/PCTDOMAINSOCK")
    print("Ready to send.")
    client.send("END".encode('ASCII'))
else:
    print("Couldn't Connect!")
```

Listing 3.9: Implementation of domain communication in Python.

As seen in listing 3.9 the transmission side in a python script is more straightforward than in C++. Only a check to see if the socket already exists. This code can then be called at the appropriate place to tell the receiver to stop the blocking call.

3.7 Implementing IPbus Control Protocol (IPBus)

In a detector that contains as many individual pixel sensors as the one developed for the Bergen pCT project, there is a need for an extensive DCS system. The main task of the DCS is to configure the detector with optimal settings for each ALPIDE. Each ALPIDE has been classified during the manufacturing process to a specific quality level. The basis for this quality level is how many noisy pixels that the chip contains. Noisy pixels will produce noise in the data generated by the detector. To combat this issue, the ALPIDE can mask out the noisy pixels. Each operation requires several read and writes from the DCS to the pRU.

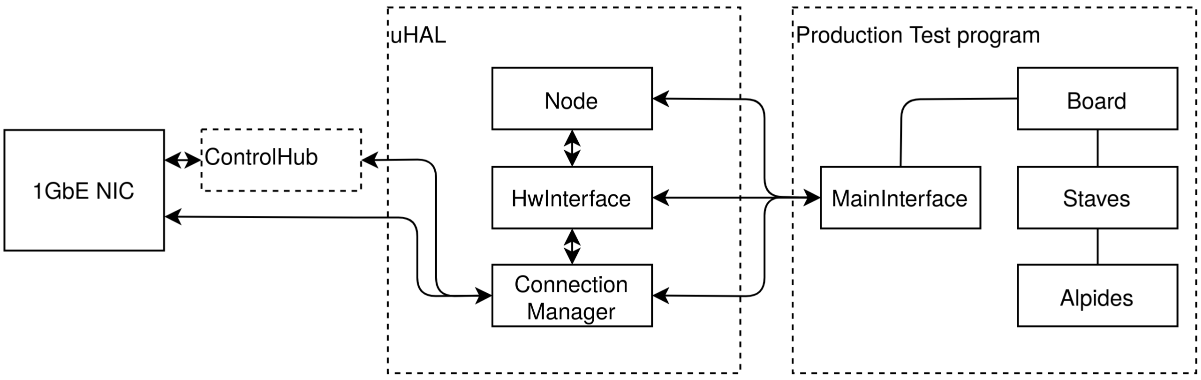

Figure 3.15: Production tests interface to IPBus

Developing a system that meets the requirements of a DCS like this will both be complex and time-consuming. However, as introduced in section 2.2.5 the IPBus package is a tested and proven system that provides the properties to solve the needs of a pCT DCS.

3.7.1 Injecting the Interface Class

Introduced in 2.1.5, the term dependency injection can be employed to make a class more agile by having e.g. a network client injected through the constructor of the class. The board class is the top class that gets initialized when a production test program runs. It depends on a interface class (here interface relates to the class name and not the concept of an Unified Modeling Language (UML) interface).

3.7.2 Modifications To The Board Class

Unfortunately there was a need to modify some of the board class functions that use the interface class and its descendants. This was done to be able to use the MainIPBus class. Due to the legacy control system custom data packages that is created by the board class is properly passed to the interface and then being unpacked in the embedded operating system.

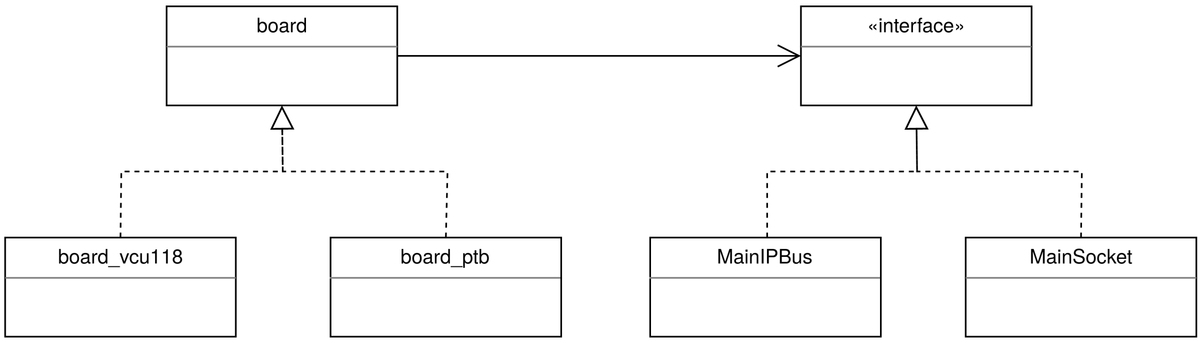

Figure 3.16: Board class with the interface association.

3.7.3 uHAL XML Address Table Files

The control software before IPBus would have its address files generated from the firmware modules automatically into header files like the one in listing 3.10. In the software, when the programmer needs to send a value to the pRU the following line would be used:
`board.write_reg(ALPIDE_CONTROL_H.BASEADDR + START_WRITE_OFFSET, 0x1)`
These functions rely on that the programmer calls the correct address and that the bit-shifting arithmetic is correctly performed.

```
    """ Register: start_write """
    START_WRITE_OFFSET = 0x0
    START_WRITE_RESET = 0x0
```

Listing 3.10: Excerpt from ALPIDE_CONTROL_H

Instead of header files, IPBus organizes the modules and addresses into structured XML files. In the XML file, the module is the top node, register address, and the bit field is also a node. With IPBus, no bit-shifting operations are required to send a command. There are also built-in tags to prevent writing or reading to register addresses marked exclusively for reading or writing. If an illegal operation is performed, an exception is thrown. Custom-made parameters can also be added to a node. Example of an IPBus command: `hw.getNode("alpide_control.start_write").write(0x1)`, after the desired operations have been stated, the `hw.dispatch()` call is needed to be performed before any values can be read by the rest of the software.

```
<node id="alpide_control">
    <node id="start_write" address="0x00000000" permission="w"
   description="..." parameters="reset=0x0;pulse_cycles=1;stall_cycles
   =70"/>
</node>
```

Listing 3.11: Excerpt from alpide_control.xml

3.7.4 Mapping Registry Addresses

The software for the old prototype was designed in a way to use special control functions to read and write to the different registers on the FPGA. To avoid breaking the functionality of the class called board that contains these read and write functions, the MainIPBus interface class needs to handle the addresses of the registers as the hexadecimal address rather than the name given in the address declaration XML file. Designing this so that it will be performed efficiently, upon constructing the mainipbus class, all the addresses get added to an ordered dictionary (hash map). The hexadecimal numeric value becomes the key in the dictionary, and the actual address name becomes the value of the key,value pair. In this way, a look up in the dictionary can be performed at $\mathcal{O}(1)$ speed. Levering the cost of creating the dictionary at creating the object instead of every time the functions read() or write() gets called.

```
manager = uhal.ConnectionManager("file://../bsp/pRU-connections.xml")
self.regDict = collections.OrderedDict()
    self.hw = manager.getDevice(layer)
    regs = self.hw.getNodes()
    for module_node in regs:
        if not("." in module_node):
            regs2 = self.hw.getNode(module_node).getNodes()
            for node in regs2:
                if not("." in node):
                    self.regDict[self.hw.getNode(module_node + "." +
   node).getAddress()] = module_node + "." + node
```

Listing 3.12: Orderd Dictionary for reverse register lookup.

Upon initializing the hardware manager IPBus will add all the modules and their register addresses. The code in listing 3.12 loops through all the addresses in the hardware manager and adds them to the dictionary.

```
def read(self, reg_addr):
    val = self.hw.getNode(self.regDict[reg_addr]).read()
    self.hw.dispatch()
    return int(val)
```

Listing 3.13: Reading a register through IPBus.

3.7.5 Control Sequences

To be able to read and write to the ALPIDEs the control module on the FPGA must be notified that it needs to perform a series of operations. The reason behind this is that the bus on the FPGA that handles the ALPIDEs needs to be stalled so all the commands and values is in the appropriate registers before the bus is released and the pRU-ALPIDE transactions can be performed.

```
    def write_alpide(self, reg_addr, value, chipid, staveid, opcode):
        self.hw.getNode("alpide_control.write_address").write(reg_addr)
        self.hw.getNode("alpide_control.write_data").write(value)
        val = (opcode << 12) + (staveid << 8) + chipid
        self.hw.getNode("alpide_control.write_ctrl").write(val)
        # Initiate the control statemachine
        self.hw.getNode("alpide_control.start_write").write(1)
        self.hw.dispatch()
```

Listing 3.14: Write Alpide Function.

The ALPIDE address and its value is entered to the designated registers, the write opcode is constructed based on the stave and chip id. Finally the control is initiated, and `dispatch()` is called to let IPBus know that it should send all the operations from the computer to the FPGA.

In addition to the `write()` function there is also `read()`, `write_opcode()` and `broadcast()` functions to handle the different aspects of the ALPIDE communication.

3.7.6 uHAL Dummy Hardware - Emulator

With the uHAL software package, there is provided software called dummy hardware. This software emulates a IPBus endpoint so that a developer does not have to have access or be in a lab with physical hardware. The only functionality that the dummy hardware provides is that if a register is written to, the same value can be read back. This dummy hardware software can easily be modified so that a file reader can enter registers and values into the system before any operations are performed on it by the control software.

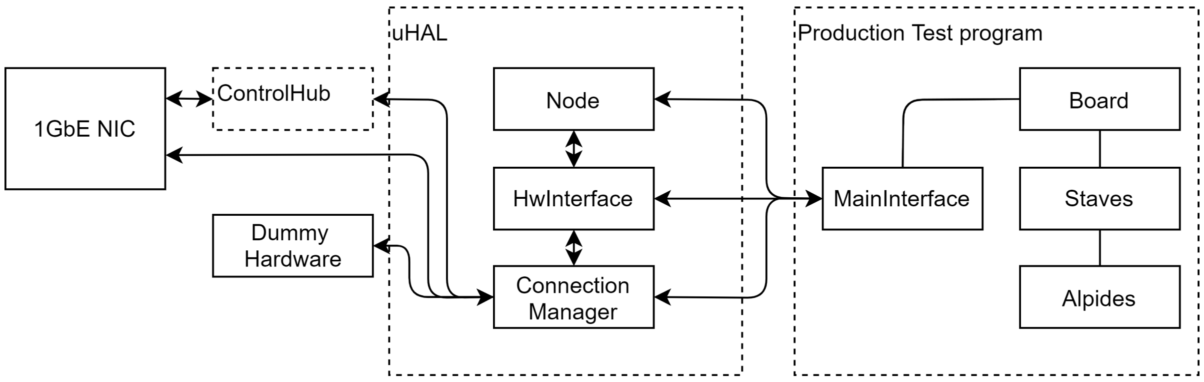

Figure 3.17: IPBus system with the dummy hardware as a module.

The dummy hardware software can easily be added alongside the other instances, both actual hardware, and several dummy hardware instances.

CHAPTER 4

Analysis and Assessment

4.1 Benchmarks of the readout system

4.1.1 Test Firmware

During development and testing of the pDTP Protocol, the VCU118 board needed to be programmed by a special firmware. The difference between this firmware and the regular firmware used for the pRU is that the test firmware only contains the offload module connected to a data generation module. This data generation module starts sending out payload with the value 0x1. For each packet that is transmitted, the value of the payload will increment the value by 1.

4.1.2 Evaluating the pDTP Protocol

To assess the network performance of a system there are several different ways this can be achieved as to measure the maximum processing speed of a system. There exists tools like iperf that can be used to measure the end to end transfer between computers, but due to the custom hardware in the pRU system the iperf tools or any other off the shelf software are not a viable option. A custom firmware was provided which would send data packets at the maximum speed that would be requested and a simple UDP client could be used to transmit the requested pDTP command and receive the number of packets that would be the reply.

Maximum Throughput

The maximum throughput in packets per second for a Ethernet link can be calculated from the following formula:

$$Packets\, per\, second = \frac{Link\, speed}{(Framesize)\, bytes \times 8\, bit/Bytes} \tag{4.1}$$

$$Protocol\, overhead = \frac{Packet\, size - Payload\, size}{Packet\, size} \tag{4.2}$$

$$Protocol\, efficiency = \frac{Payload\, size}{Packet\, size} \tag{4.3}$$

Table 4.1: Ethernet frame sizes in the pCT network stack.

Frame part	Minimum Frame Size	Maximum Frame Size
Inter Frame Gap	5 bytes	5 bytes
MAC Preamble (+ SFD)	8 bytes	8 bytes
MAC Destination Address	6 bytes	6 bytes
MAC Source Address	6 bytes	6 bytes
MAC Type (or length)	2 bytes	2 bytes
Payload (Network PDU)	46 bytes	4124[1]
Check Sequence (CRC)	4 bytes	4 bytes
Total Frame Physical Size	77 bytes	4147 bytes

Theoretical Maximum Transfer Speed pDTP

To obtain the theoretical processing speed of the system it is beneficial to calculate the theoretical maximum transfer speed that can be achieved on a given GbE interface. To do this, we will use the equation 4.1. For a 10 GbE interface this would be for the smallest packet size: $10 \times 10^9/(77 \times 8) = 16233766$ packets per second and for the maximum packet size: $10 \times 10^9/(4147 \times 8) = 3001323$ packets per second. Both these numbers represent full 10Gb/s speed. However, if the overhead from the Interframe Gap, Ethernet frame header, IP header, and UDP header get subtracted from the calculation and leaving only the payload left in the calculation. The transfer speed can be calculated like this: $3001323 \times 4088 \times 8/10^6 = 9858$Mbps, a 98,6% efficiency.

Realistic Maximum Transfer Speed

When considering all the elements mentioned in section 2.2, the numbers presented in the previous section will be unrealistic to obtain without invoking the side effect of package drops. Package drops occur when the system can not keep up with the incoming packet stream. This issue can occur in several layers of the network stack. The main reason the resources received are dropped, is that the next layer in the stack is not ready to consume from the buffer in the layer below before the buffer is filled up. The result is that the system will delete or overwrite the oldest packet to make space for the next packet that is received.

Package Size Range

It is therefore beneficial to define a range of packet sizes from the smallest that the OS can keep up without dropping and up to the maximum size. In turn the pDTP allows for setting the MIN_RQ flag when doing a RQS or RQFS. This means that the pRU will not send any smaller packets than the ones specified in the RQ_PACKET_SIZE field. It will however send packages larger than this size if the transmit buffer of the pRU fills up faster.

[2]IP header: 20 + UDP header: 8 + pDTP header + payload: 4088

Measuring Packets Per Second

When analyzing a network system, it is often beneficial to be able to view the raw packets on-the-wire. Unfortunately doing this will slow down the system a great deal and result in packet drops. Additionally, you only see what the network card relays on to the system. You will only be able to capture from network layer and upwards in the stack (See figure 2.5). Instead a custom UDP client needs to be designed to measure and calculate the correct transfer speed.

4.1.3 Hardware Tuning

There are several settings on the network card that is used in the prototype. The most important ones are the ring buffer size of the card how often the card is allowed to interrupt the processor when a package is received.

```
ethtool -G enp21s0f0  rx 4096
```

Listing 4.1: Command to change the ring buffer on the NIC

Using ethtool with switch -G will change the ring buffers descriptors up to the value of 4096. Ethtool can be used for both configuration and statistics output for the NIC.

4.1.4 Linux OS Tuning

As described in section 2.3.2 there are several different buffers in the stack. Some of these buffers can be increased in size to help mitigate the rate of packets that are incoming to the system. Some examples are the backlog of how many unprocessed packages, the size of how much RAM the UDP layer is allocated.

A list of these commands can be found in appendix F

4.1.5 Execution Tuning

To get the best performance there are some measures that can be taken in order to run a program in the most optimal way. It can be beneficial to pin processes and tasks to specific cores. This is to prevent the system assigning tasks to the same resources. This is introduced in section 4.1.3 when one or more cores handles the interrupts from the network card, whilst the program runs on another dedicated core.

Programs can be set to run on specific cores by specifying the command taskset when calling the executable of the program [33]. Example:

```
taskset -c 2 ./pDTPClient
```

Listing 4.2: How to lock a program to a spesific core

This will lock the program to core number 3 of the CPU (cores are zero indexed). The reason for this is that if the OS tries to schedule the process over to another core, the OS does a context switch [34]. In most cases this is a needed operation due to load balancing. A context switch is a costly operation that essentially stops the execution of the program, the actual cost can be in the tenths of microseconds [35]. A stop like this in execution can thus result in dropped packages, if the stop time exceeds the time it takes for a buffer to overflow.

4.2 Benchmark Results of the pDTP Client

The results in this section are based of three different clients, one plain UDP client, one pDTPClient using `std::vector` and one using `std::array`.

4.2.1 Benchmark UDP Client

To design a UDP client the best approach to get it to perform in the fastest possible way is to use the same programming language that the OS is programmed in. For Linux this is the C programming language, which offers a unrestricted access to resources in the system, at the cost of complexity and the risk of unintended effects. The client only needs to send a single pDTP client packet and receive the number of packets requested in the command, while running a timer at the start of the program and and stopping it when receiving end of stream packet (EOS). The source code for the client can be viewed in appendix C.1.

4.2.2 pDTP client with `std::vector`

This client was the first to be developed, but after several software benchmarks, it became clear that there existed a bottleneck in the software.

```
boost::lockfree::spsc_queue<std::vector<char>> spscQueue{10000};
```

Listing 4.3: Declaration of the SPSC queue with `std::vector`.

Declaring the SPSC this way results in 10 000 empty vectors. The result is that every time a new element is pushed to the queue the system has to reserve memory for the data that enters queue.

4.2.3 pDTP client with `std::array`

A solution to the issue that is related to the `std::vector` implementation is to use a fixed size `std::array` instead.

```
boost::lockfree::spsc_queue<std::array<char, 4096>> spscQueue{10000};
boost::lockfree::spsc_queue<int> spscSizeOfArray{conf.spscQueueSize};
```

Listing 4.4: Declaration of the SPSC queue with `std::array`.

All the memory for the data structure, both the queue and the arrays inside it, are declared when the application is executed. One drawback is that a second SPSC has to be declared and used to keep track of the actual size of the data when it is popped out of the queue by the consumer thread.

4.2.4 Measurements

One pDTP client using std::vector as the primary data structure and finally a pDTP client that uses the std::array instead of std::vector. The data is based on 10 non-concurrent transfers of 65535 packets ranging in size from 2.5KB to 4KB.

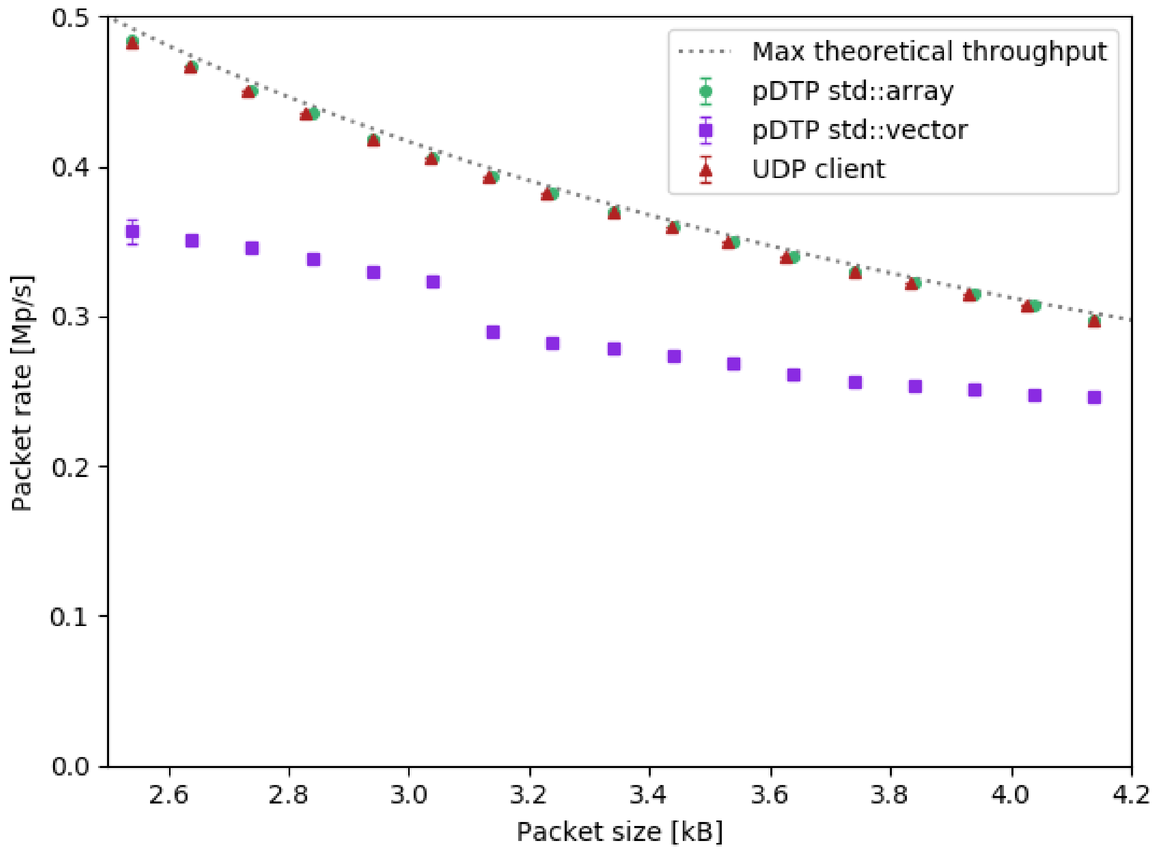

Figure 4.1: Readout packets per second

When evaluating the largest package size with the payload of 4080 bytes, the UDP client receives at a mean of 297648 ± 5.5 pps, while the pDTP client with array has a mean of 297669 ± 33 pps. These clients have respectively 3775 pps and 3754 pps difference of the theoretical max of 301443 pps. The pDTP client that uses std::vector only receives at the speed of 245955 ± 461 pps, this indicates that this client only has a utilization of 80% or 8.027 Gbps at of the max speed of the 10GbE link.

The dissimilarity between the clients that uses std::vector and std::array is that memory used in the first case is dynamically allocated during the run time of the client whilst the client is receiving the data. In the other case the configuration file of the client states how many packages that the SPSC queue shall hold, therefore the memory needed is allocated at start up instead. The draw back of this is that the client is using a large portion of RAM, but with levering this over the much more costly CPU operations during run-time.

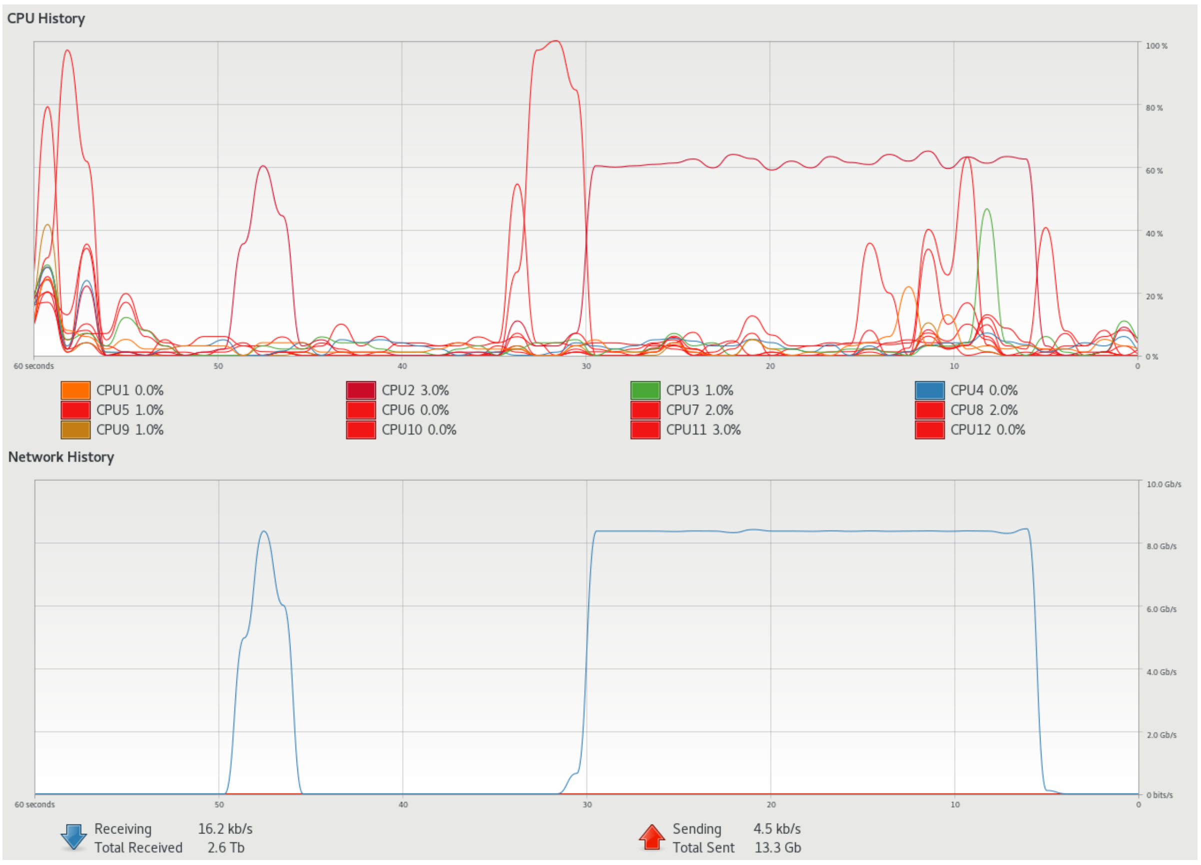

Figure 4.2: Network traffic and CPU load when a data stream is running.

In figure 4.2 there is a clear comparison on how much load the CPU core that has the program pinned to. Even though the software is running at just under 9 Gbps, the CPU load is just over 60%.

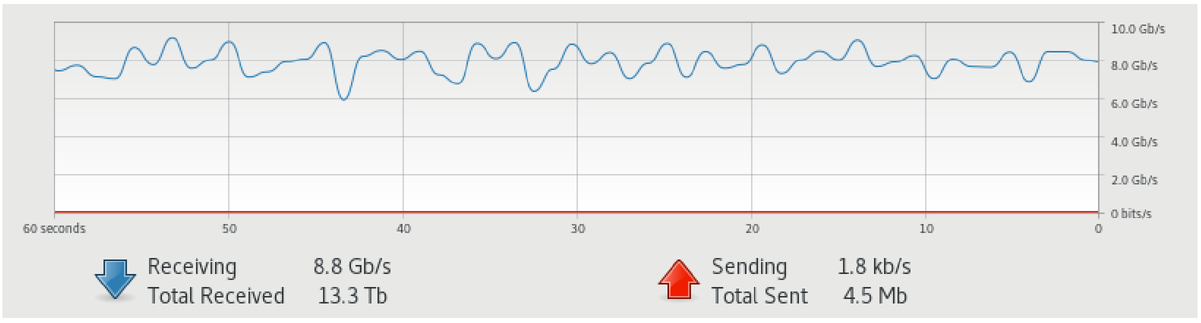

Figure 4.3: Network traffic when a full auto data stream is running.

While running the full stream mode (RQFS) the speed is more variable as seen in figure 4.3.

4.2.5 Profiling the pDTP Client

The perf tool was originally designed to evaluate the performance of modules in the Linux subsystem. When running the perf tool, it creates a call stack for the program that is being profiled. The output from the perf tool can be converted into a flame graph for better visualization. Comparing figure E.1 and figure E.2 in appendix E after first running the benchmark test, the results show that the client software at the time was only capable of between 0.25 Mp/s and 0.4Mp/s. After running perf on the software, the flamegraph shows that the bottleneck is the allocation of memory when using std::vector.

4.3 PTB Analysis

In comparison to the pDTP Client the PTB Server is designed to operate over a 1 GbE interface and over the TCP protocol. Using the same design methodology used when constructing the pDTPClient, the PTB server's performance can hold a steady data rate of incoming packets. The servers decoupling from the rest of the processing chain makes the embedded offload system perform more reliable and reduces any retransmits in the TCP connection.

4.4 Benchmarks of the control system

The primary motivation for a fast control system of the pCT detector is that some of the alpide chips have broken pixels in them. Therefore it is essential to be able to mask these "dead" pixels as fast as possible. An early benchmark with the old prototype system showed that performing 10 000 reads and writes sent from the computer through MCU would take almost 12 minutes while doing the same tasks through IPBus, and the result is just over 11 seconds.

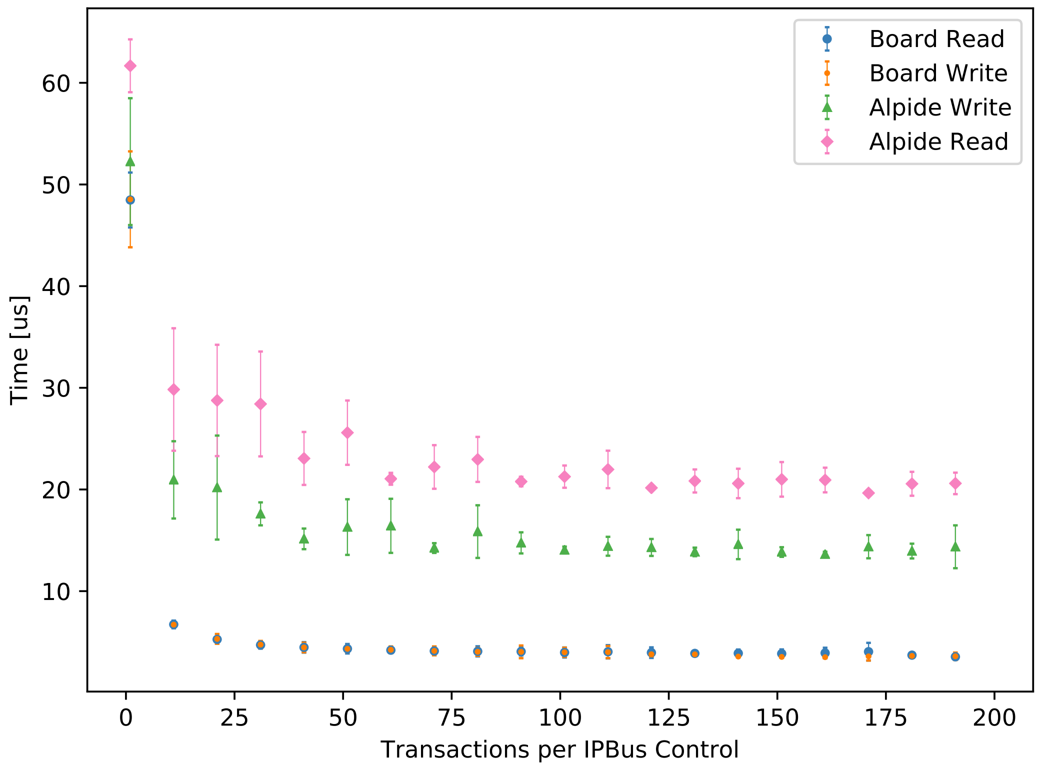

Figure 4.4: DCS benchmark

Figure 4.4 shows a comparison with board R/W and ALPIDE R/W, board are a single IPBus transactions, while ALPIDE are 4 IPBus transactions. The most transactions a regular UDP packet can hold is 175. At this number, the system will trigger a dispatch automatically. A number of transactions lower than 175 the dispatch function need to be called specifically in the code.

4.5 Impact of Thesis Work

The work that laid the basis for this thesis has contributed to the work of several others. When the work started out, the evaluation of 10 GbE and UDP contributed to a a conference paper [8]. Towards the end, when working on the thesis, the work also contributed to a physics paper about pCT [36]. The implementation of the pDTP and the results of the benchmarks have been used in a Ph.D. thesis [19]. The software that has developed is used to read out data from the ALPIDE staves connected to the development board that functions as a pRU. The data collected will be used in a future Ph.D. thesis, and other researchers in the project will also use the software to collect data. Also, several master students in their work will use the pDTPClient and the IPBus control software to continue to develop software for the detector.

CHAPTER 5

Conclusion

5.1 Performance Evaluation

5.1.1 pDTP Client Performance

The findings presented in chapter 4 show that a medium to a high scale computer system running the pDTP client can handle network speeds up to the limit of the 10 GbE NIC. However, it is easy for the pRU with the FPGA based network interface to overwhelm the computer system. This issue can occur if the packets are too small, resulting in a high rate of packets per second on the incoming interface. A solution to help mitigate this issue is to use the built-in mechanisms in the pDTP protocol, i.e. limit the requested package size. Another mechanism is to use the client request throttle opcode listed in the pDTP specification to increase the window between packages transmitted by the pRU.

The benchmarks used are both self-developed software as well as the Linux perf tool. In addition to the previously mentioned pRU hardware and protocol-specific issues, benchmarks have also identified bottlenecks that might occur in the network and on the computer. The most prominent bottleneck in the computer system is in the kernel. Section 4.1.3 shows what network settings to apply to the kernel, and some of these parameters apply a tradeoff that increases latency in the system while leveraging larger buffer sizes for reliability. One of the significant findings is in the client software itself, the data structure that the data queue in the client utilizes from one thread to another.

Nevertheless, considering that the actual readout rate for each pRU layer might be less than 2 Gbps, the client performs well over this estimate and has already been utilized in a test production environment as the primary system for transferring readout data streams. The client proves that it can act as reliable use of the pDTP protocol over UDP.

5.1.2 DCS and IPBus Performance

In section 4.4 the performance of the IPBus was measured to see how many transactions it will take for the uHAL system to trigger a package over the network to the control module. These findings can form the basis for how long it would take to transfer and configure the over 4200 ALPIDEs in the whole detector. The use of the IPBus system reduces the time spent by over a factor of 10. It shows a clear advantage over the previous prototype system that relies on an MCU and an embedded RTOS to perform the control transactions with the ALPIDEs. The major drawback of the old prototype was that it had both the readout functionality and the control system on the same interface, and that the MCU had to control both of the functionality at same time.

5.2 Design Evaluation

Section 3.4 gives an example of how to implement the policy-based design pattern to make the software more modular and adaptable to the requirements of a system like the pCT readout chain.

5.2.1 pDTP Client

The design methodology for the pDTP client introduced in section 3.2.4 has its foundation based on proven design principles used within software engineering. These principles aids the design and construction of the code. The resulting software performs above the simulated estimate for the transfer rates in the system. The utilized software design patterns can aid further optimization of the existing code. Furthermore, it gives the developer the option to add more complex mocking tests or simpler unit tests.

Due to the different speeds of the incoming data rate and the module that further processes the data, the producer-consumer pattern is applied as described in section 3.5 to provide a safe buffer between these modules.

5.2.2 PTB Server

The PTB Server input policy implementation performs better than the previous standalone implementation and the version implemented in Python. The server offers the same decoupled properties as the pDTP client as it can receive data without waiting for the parser or file writer part of the readout chain.

5.2.3 DCS and IPBus Design

Considering the qualitative evaluation of the IPBus system, the development time for the firmware designer to integrate the modules into the FPGA firmware was reasonably short. IPBus facilitates an easy documentation process to integrate the register addresses for the different modules into the IPBus XML address files. Since uHAL provides both support for C++ and Python, it was pretty easy to implement an interface class into the existing production test software package by utilizing software patterns.

The only drawback during the implementation process was a more common problem with an operating system still dependent on Python 2.7. A few re-build processes had to be done to make the uHAL interface built correctly to Python 3.x.

Since IPbus, together with uHAL, is a proven and tested firmware and software package by developers in a CERN project, the overall time and cost-saving achieved by selecting the IPBus system over designing and developing an in-house system can amount to years.

5.3 Summary

This thesis has implemented the custom data transfer protocol pDTP and investigated the problems that have arisen during the testing and evaluation. The software has been tested on a emulator, finally moving on to hardware and being used reliably in a test system to read out actual data from a string of ALPIDE sensors. Finally the data that have been read out has been analyzed and proved correct.

The IPBus system exceeds the expectations, and any initial reservations were proved wrong at an early stage of the evaluation. It rapidly replaced the old control system based on a MCU design. It adds several features that help shorten the development process and aid the programmer by ensuring that correct addresses are read and written. Also, the IPBus system has been used reliably in the test system.

The result of the work that is the foundation for this thesis have been performed in a team setting and the success of the software produced have helped other team members reach their goals. But also without the team the authors own work would not have been successful.

CHAPTER 6

Further Work

6.1 Evaluation of Network Infrastructure

When more pRUs become available either development boards or when the pRUs themselves are produced, it will be beneficial to look closer at what network hardware will be most suitable. Also, what kind of metrics can be measured to establish the most optimal network infrastructure to be used in the pCT detector system.

6.1.1 Switches

The layout of network switch hardware is that there is a front end with the RJ-45 or SFP++ ports, while the switching fabric itself is referred to as a backplane. This backplane has different strengths and drawbacks depending on the model and brand. The most important one is the metrics on how many packets it is able to switch at a given moment. It would be beneficial to do both evaluations of different switches and benchmark tests on physical hardware.

6.1.2 Network card

There are more advanced settings of the Intel X710-DA NIC that can be explored. It would also be beneficial to evaluate another NIC than the Intel X710-DA. There currently exist smart NICs that can be programmed to handle the pDTP protocol and thus offloading this costly operation from userspace to hardware itself.

6.2 Newer Kernel

Due to the restrictions of the old 3.x Linux kernel that was used on the test computer, the development of the pDTP client has been limited to the functionality that this kernel provided. Therefore, it could be beneficial to evaluate a newer kernel version.

6.2.1 Extended Berkeley Packet Filter (eBDF)

A feature since kernel version 4.14 is the eXpress Data Path, which can be shortly explained as Kernel bypass for hardware to go directly to userspace. An example is that eBDF can specify which RX queue on the network card a given port uses. An XDP socket can then read from that specified queue. One limitation is that the i40e driver only supports up to 3Kb frames when relaying to eBDF [37]. Another drawback is that an eBDF script needs to be loaded into the kernel of the host computer. This issue could introduce possible instability in the system.

6.2.2 Zero Copy UDP Socket

Zero Copy is a feature introduced for TCP protocol in Linux kernel version 4.15 with the aim to reduce the amount of data that is being copied when going from one layer of the network stack to the next. In kernel version 5.1 it is also implemented for UDP that the pDTP protocol uses. The utilization of Zero Copy could help the readout process to be more reliable at smaller packet sizes. Benchmark tests would be needed to evaluate this functionality.

6.3 Data Plane Development Kit

The Data Plane Development Kit is a collection of libraries and drivers that helps unload the handling of packages in the kernel and over to processes that run in userspace. The DPDK provides drivers for the X710-DA card used in this thesis. DKDP is also a kernel bypass functionality like the eBDF, except DKDP is not limited in frame size like the regular i40e driver is.

6.4 IPBus

The IPBus DCS needs to be evaluated on a larger scale with a system that has more pRUs with several staves on them. This will give a clearer picture of the performance of the control system.

6.4.1 Control Hub

Further more is would be beneficial to evaluate the Control Hub functionality on a larger scale and measure the if there is a effect to use dedicated hardware to run the Control Hub software.

6.4.2 Convert Detector Control System (DCS) to C++

There is a huge speed advantage porting the existing code base that is used for the DCS to the C++ language. Also by having the entire system being developed in C++ the software would have higher reliability.

Bibliography

[1] *Facts about cancer*. 2018. URL: https://www.kreftregisteret.no/Generelt/Fakta-om-kreft/ (visited on 05/16/2020).

[2] Helse- og omsorgsdepartementet. *Etablering av protonsentre i Bergen og Oslo*. May 2018. URL: https://www.regjeringen.no/no/aktuelt/etablering-av-protonsentre-i-bergen-og-oslo/id2601206/ (visited on 05/16/2020).

[3] M VANHERK. "Errors and margins in radiotherapy". In: *Seminars in Radiation Oncology* 14.1 (Jan. 2004), pp. 52–64. ISSN: 10534296. DOI: 10.1053/j.semradonc.2003.10.003. URL: https://linkinghub.elsevier.com/retrieve/pii/S1053429603000845.

[4] Johannes A. Langendijk, Philippe Lambin, Dirk De Ruysscher, et al. "Selection of patients for radiotherapy with protons aiming at reduction of side effects: The model-based approach". In: *Radiotherapy and Oncology* 107.3 (June 2013), pp. 267–273. ISSN: 01678140. DOI: 10.1016/j.radonc.2013.05.007. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167814013002193.

[5] Cepheiden. *Bragg Peak*. 2010. URL: https://commons.wikimedia.org/wiki/File:Dose_Depth_Curves.svg (visited on 08/29/2020).

[6] Uwe Schneider, Eros Pedroni, and Antony Lomax. "The calibration of CT Hounsfield units for radiotherapy treatment planning". In: *Physics in Medicine and Biology* 41.1 (Jan. 1996), pp. 111–124. ISSN: 0031-9155. DOI: 10.1088/0031-9155/41/1/009. URL: https://iopscience.iop.org/article/10.1088/0031-9155/41/1/009.

[7] Gianluca Aglieri Rinella. "The ALPIDE pixel sensor chip for the upgrade of the ALICE Inner Tracking System". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 845 (Feb. 2017), pp. 583–587. ISSN: 01689002. DOI: 10.1016/j.nima.2016.05.016. URL: https://linkinghub.elsevier.com/retrieve/pii/S0168900216303825.

[8] Ola Slettevoll Groettvik, Johan Alme, Rene Barthel, et al. "Development of Readout Electronics for a Digital Tracking Calorimeter". In: *Proceedings of Topical Workshop on Electronics for Particle Physics — PoS(TWEPP2019)*. Trieste, Italy: Sissa Medialab, Mar. 2020, p. 090. DOI: 10.22323/1.370.0090. URL: https://pos.sissa.it/370/090.

[9] Erich Gamma, Richard Helm, Ralph Johnson, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201633612.

[10] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN: 0201704315.

[11] John Maddock, A. Paul Bristow, Hubert Holin, et al. *Policy Overview - 1.38.0*. 2008. URL: `https://www.boost.org/doc/libs/1%7B%5C_%7D38%7B%5C_%7D0/libs/math/doc/sf%7B%5C_%7Dand%7B%5C_%7Ddist/html/math%7B%5C_%7Dtoolkit/policy/pol%7B%5C_%7Doverview.html` (visited on 02/03/2021).

[12] ISO Central Secretary. *C++ International Standard 2011*. Tech. rep. ISO/IEC 14882:2011. International Organization for Standardization, 2011. URL: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`.

[13] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, et al. "An Efficient Unbounded Lock-Free Queue for Multi-core Systems". In: ed. by C Kaklamanis, T Papatheodorou, and P. G Spirakis. Springer, 2012. DOI: `10.1007/978-3-642-32820-6_65`.

[14] Dhanji R. Prasanna. *Dependency injection: design patterns using Spring and Guice*. USA: Manning Publications Co., 2009. ISBN: 978-1-933988-55-9.

[15] Intel. *Intel® IXP4XX Product Line of Network Processors and IXC1100 Control Plane Processor: Understanding Big Endian and Little Endian Modes*. Tech. rep. Intel, 2003. URL: `https://www.intel.com/content/dam/www/public/us/en/documents/application-notes/ixp4xx-ixc1100-big-endian-little-endian-modes-note.pdf` (visited on 05/08/2021).

[16] "IEEE Standard for Ethernet". In: *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)* (2018), pp. 1–5600.

[17] *pCT Wiki*. URL: `https://wiki.uib.no/pct/index.php/Main_Page` (visited on 06/06/2021).

[18] G Fairhurst L. Eggert and G Shepherd. "UDP Usage Guidelines". In: *Internet RFCs, ISSN 2070-1721* RFC 8085 (2017). URL: `https://tools.ietf.org/pdf/rfc8085.pdf`.

[19] Ola Slettevoll Grøttvik. "Design and Implementation of a High-Speed Readout and Control System for a Digital Tracking Calorimeter for proton CT". PhD thesis. University of Bergen, Feb. 2021. URL: `https://hdl.handle.net/11250/2725567`.

[20] Ola S. Grøttvik. "pCT Data Transfer Protocol". The pDTP specification document. Sept. 2019.

[21] C. Ghabrous Larrea, K. Harder, D. Newbold, et al. "IPbus: a flexible Ethernet-based control system for xTCA hardware". In: *JINST* 10.02 (2015), p. C02019. DOI: `10.1088/1748-0221/10/02/C02019`.

[22] *The CentOS Project*. URL: `https://www.centos.org/` (visited on 06/06/2021).

[23] Brendan Gregg. *BPF Performance Tools (Addison-Wesley Professional Computing Series)*. Addison-Wesley Professional, 2019. ISBN: 9780136554820. URL: `http://www.brendangregg.com/bpf-performance-tools-book.html`.

[24] Ola Slettevoll Grøttvik. "Design of High-Speed Digital Readout System for Use in Proton Computed Tomography". MA thesis. Department of Physics and Technology, 2017, p. 91. URL: `https://hdl.handle.net/1956/16041`.

[25] Karl Emil Sandvik Bohne. “Ethernet-Based Control System and Data Readout for a Proton Computed Tomography Prototype”. MA thesis. Department of Physics and Technology, 2018, p. 80. URL: `https://hdl.handle.net/1956/18466`.

[26] Håkon Andreas Underdal. “Data Acquisition and Testing Software for a Proton Computed Tomography System”. MA thesis. Department of Computing, Mathematics and Physics, 2019, p. 60. URL: `https://hdl.handle.net/1956/20846`.

[27] Øistein Jelmert Skjolddal. “Scalable Readout for Proton CT”. MA thesis. Department of Computing, Mathematics and Physics, 2020, p. 60. URL: `https://hdl.handle.net/1956/24109`.

[28] Tea Bodova. “High-Speed Signal and Power Distribution of a Digital Tracking Calorimeter for Proton Computed Tomography”. MA thesis. Department of Physics and Technology, 2020, p. 82. URL: `https://hdl.handle.net/1956/23535`.

[29] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. “Technical Debt: From Metaphor to Theory and Practice”. In: *IEEE Software* 29 (6 Nov. 2012). ISSN: 0740-7459. DOI: `10.1109/MS.2012.167`.

[30] Wireshark. *Wireshark Go Deep.* URL: `https://www.wireshark.org/`.

[31] Beman Dawes. *Boost.Endian: The Boost Endian Library.* URL: `https://www.boost.org/doc/libs/1_71_0/libs/endian/doc/html/endian.html#overview_introduction` (visited on 05/08/2021).

[32] Christopher M. Kohlhoff. *UNIX Domain Sockets.* URL: `https://www.boost.org/doc/libs/1_76_0/doc/html/boost_asio/overview/posix/local.html` (visited on 05/02/2021).

[33] Kaiwan Billimoria. *Hands-on system programming with Linux : explore Linux system programming interfaces, theory, and practice.* Birmingham, UK: Packt Publishing, 2018. ISBN: 978-1-78899-847-5.

[34] Robert Love. *Linux Kernel Development.* 3rd. Addison-Wesley Professional, 2010, p. 480. ISBN: 9780672329463.

[35] Chuanpeng Li, Chen Ding, and Kai Shen. “Quantifying the cost of context switch”. In: *Proceedings of the 2007 workshop on Experimental computer science - ExpCS '07.* New York, New York, USA: ACM Press, 2007, 2–es. ISBN: 9781595937513. DOI: `10.1145/1281700.1281702`. URL: `http://dl.acm.org/citation.cfm?doid=1281700.1281702`.

[36] Johan Alme, Gergely Gábor Barnaföldi, Rene Barthel, et al. “A High-Granularity Digital Tracking Calorimeter Optimized for Proton CT”. In: *Frontiers in Physics* 8 (2020), p. 460. ISSN: 2296-424X. DOI: `10.3389/fphy.2020.568243`. URL: `https://www.frontiersin.org/article/10.3389/fphy.2020.568243`.

[37] Intel. *i40e Linux* Base Driver for the Intel(R) Ethernet Controller 700 Series.* 2020. URL: `https://downloadmirror.intel.com/24411/eng/README.txt`.

APPENDIX A

Production Test Library

```python
import uhal
import time
import struct
import collections
import logging
from alpide_consts import ALP_REG, OPCODE


class MainIPbus(object):
    """Creates a MainIPBus object

    """
    def __init__(self, layer):
        self.verbose = 0
        manager = uhal.ConnectionManager("file://../bsp/pRU-connections.xml")
        if self.verbose <= 0:
            uhal.disableLogging()
        # Adding all the registers to a dictionary this gives O(1) for numeric value lookup to string address.
        self.regDict = collections.OrderedDict()
        self.hw = manager.getDevice(layer)

        regs = self.hw.getNodes()
        for module_node in regs:
            if not("." in module_node):
                regs2 = self.hw.getNode(module_node).getNodes()
                for node in regs2:
                    if not("." in node):
                        self.regDict[self.hw.getNode(module_node + "." + node).getAddress()] = module_node + "." + node

    def write(self, reg_addr, reg_val):
        """ Send command

        Args:
            reg_addr (int): the address of the registry you want
                            to write to.
            reg_val (int): the value that you want to write to the
                           register.
        """
        if self.verbose > 0:
            print(hex(reg_addr))
        regs = self.hw.getNode()
        self.hw.getNode(self.regDict[reg_addr]).write(reg_val)
        self.hw.dispatch()

    def read(self, reg_addr):
        """Read reg

        Args:
         reg_addr (int): the address of the of the register you want
                        to read.
        Returns:
            int: an int with the value that was read.
        """
        val = self.hw.getNode(self.regDict[reg_addr]).read()
        self.hw.dispatch()
        return int(val)

    def read_alpide(self, reg_addr, chipid, staveid):
        """Reads an ALPIDE register.

        Args:
            reg_addr (int): reg_addr the register address that you want
                            to read from.
            chipid (int): chipid the chip id of the ALPIDE that you
                            want to read from.
            staveid (int): staveid the stave id of the stave that the
                            ALPIDE is located on.

        Returns:
            int: with the value that was held in the ALPIDE register.
        """

        self.hw.getNode("alpide_control.write_address").write(reg_addr)

        opcode = OPCODE.RDOP
        val = (opcode << 12) + (staveid << 8) + chipid
        self.hw.getNode("alpide_control.write_ctrl").write(val)
        # Initiate the control statemachine
        self.hw.getNode("alpide_control.start_read").write(1)
        status = self.hw.getNode("alpide_control.read_status").read()
        data = self.hw.getNode("alpide_control.read_data").read()
        self.hw.dispatch()

        status = int(status)
        data = int(data)
        all_ok = status & 1
        chipid_ok = (status >> 1) & 1
        data_l_ok = (status >> 2) & 1
        data_h_ok = (status >> 3) & 1
        chipid_in = (status >> 4) & 0xFF
        if (not all_ok ) or (not chipid == chipid_in):
            msg = ("read_alp_reg - chip_id: {0} - reg_addr: {1}\n"
                   "Something went wrong!\n"
                   "Status: {2}\n"
                   "All ok: {3}\nChip ID ok: {4}\nData L ok: {5}\n"
                   "Data H ok: {6}\nChip ID: {7}\nData: {8}\n").format(chipid, hex(reg_addr), hex(status),
                   all_ok, chipid_ok, data_l_ok, data_h_ok, chipid_in, hex(data))
            logging.debug(msg)
        return data


    def write_alpide(self, reg_addr, value, chipid, staveid, opcode):
        """Writes to an alpide register

        Args:
            reg_addr (int): reg_addr the address of the register on the
                         ALPIDE that you want to write to.
            value (int): the value that you want to write to
                         the register on the ALPIDE
            chipid (int): the chip id of the ALPIDE that you want to
                         write to.
            chipid (int): staveid the stave id of the stave that
                        the ALPIDE is located on.
            opcode (int): the ALPIDE opcode that you want to perform
        """

        self.hw.getNode("alpide_control.write_address").write(reg_addr)
        self.hw.getNode("alpide_control.write_data").write(value)
        val = (opcode << 12) + (staveid << 8) + chipid
        self.hw.getNode("alpide_control.write_ctrl").write(val)
        # Initiate the control statemachine
        self.hw.getNode("alpide_control.start_write").write(1)
        self.hw.dispatch()

    def write_opcode(self, opcode, chipid, staveid):
        """Writes an opcode to ALPIDE, either unicast or multicast. NOT BROADCAST

        Args:
            opcode (int): the ALPIDE opcode that you want to perform
            chipid (int): the chip id of the ALPIDE that you
                        want to write to.
            staveid (int): the stave id of the stave that the
                            ALPIDE is located on.
        """
        self.hw.getNode("alpide_control.write_address").write(ALP_REG.COMMAND)
        self.hw.getNode("alpide_control.write_data").write(opcode)
        val = (opcode << 12) + (staveid << 8) + chipid
        self.hw.getNode("alpide_control.write_ctrl").write(val)
        # Initiate the control statemachine
        self.hw.getNode("alpide_control.start_write").write(1)
        self.hw.dispatch()

    def broadcast(self, opcode, staveid=0xF):
        """Send a broadcast opcode. Must be 1 Byte opcode

        Args:
            opcode (int): that you want to broadcast to the ALPIDES
        """
        val = (opcode << 12) + (staveid << 8) + 0x0F
        self.hw.getNode("alpide_control.write_ctrl").write(val)
        self.hw.getNode("alpide_control.start_write").write(1)
        self.hw.dispatch()
```

Listing A.1: mainipbus interface class.

APPENDIX B

State Machine

Table B.1: FSM property table for the pDTPClient class.

State	Action	Event			Transition	Description
Idle	Set OP Code	Program start			TO_TRANSMIT	Prepare request.
Transmit	transmit(rawreq)	Transmit == 1			TO_RECEIVE	Succesfull transmit of request.
	transmit(rawreq)	Transmit != 1			TO_IDLE	Error in ComService, ex socket is closed.
Receive	receive(buf)	Recv <0			TO_IDLE	Error in ComService, ex socket returns error
	receive(buf)	Recv == 0	Timeout		CONT_RECEIVE	Read from socket times out, try read again.
			!Timeout	Retry	TO_IDLE	Error.
				!Retry	TO_RETRASMIT	Error, send new request.
	receive(buf)	Recv >0	SERVER_STREAM		CONT_RECEIVE	Data part of a stream recv or readout timeout.
			SERVER_WRITE	Retry	TO_RETRASMIT	Data received succesfull. Poll for more data.
				!Retry	TO_IDLE	Data received succesfull. Stop execution.
			SERVER_EOS	Retry	TO_RETRASMIT	Stream received succesfull. Poll for more data.
				!Retry	TO_IDLE	Stream received succesfull. Stop execution.
			SERVER_ERROR	Retry	TO_RETRASMIT	pRU reports error, ex buffer is empty.
				!Retry	TO_IDLE	pRU reports error, poll again.

APPENDIX C

Source Code

C.1 Source Code for Benchmarks

```
/*
Simple UDP client for pDTP benchmarks
*/
#include <errno.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <unistd.h>
#include <chrono>
#include <cstring>
#include <iostream>
#include <string>

int main(int argc, char *argv[]) {
  int s;
  unsigned short port;
  struct sockaddr_in server;
  struct hostent *hp;
  char buf[4096];

  port = htons(atoi(argv[2]));

  /* Create a datagram socket in the internet domain and use the
   * default protocol (UDP).
   */
  if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    perror("socket()");
    exit(1);
  }
  /* Set up the server name */
  server.sin_family = AF_INET; /* Internet Domain    */
  server.sin_port = port;      /* Server Port        */
  std::string ip = "192.168.2.20";
  hp = gethostbyname(ip.c_str());
  if (hp == 0) perror("Unknown host");

  bcopy((char *)hp->h_addr, (char *)&server.sin_addr, hp->h_length);

  const char msg[] = {0x20, 0xFF, 0xFF, 0xFF};  // Send command no ack,
    request
  /* Send the message in buf to the server */
  unsigned int slen = sizeof(struct sockaddr_in);

  struct timeval tval;
  int rslt;
  int packNum = 65535;

  std::chrono::high_resolution_clock nowTimePoint2;

  auto statSize = 15;
  std::cout << "start\n";
  for (size_t i = 0; i < statSize; i++) {
    auto start = std::chrono::time_point_cast<std::chrono::nanoseconds
   >(
        nowTimePoint2.now());
    if (sendto(s, msg, (sizeof(msg)), 0, (struct sockaddr *)&server,
               sizeof(server)) < 0) {
      perror("sendto()");
      exit(2);
    }

    for (size_t j = 0; j < packNum; j++) {
      recvfrom(s, buf, sizeof(buf), 0, (struct sockaddr *)&server, &
   slen);
    }

    auto stop = std::chrono::time_point_cast<std::chrono::nanoseconds>(
        nowTimePoint2.now());
    auto duration = stop - start;

    std::cout << static_cast<unsigned int>(
                     (packNum * 1000.f * 1000.f * 1000.f) / duration.
   count() *
                     1.f)

              << ',';
  }

  /* Deallocate the socket */
  close(s);
}
```

APPENDIX D

proton Data Transfer Protocol (pDTP) Client UML

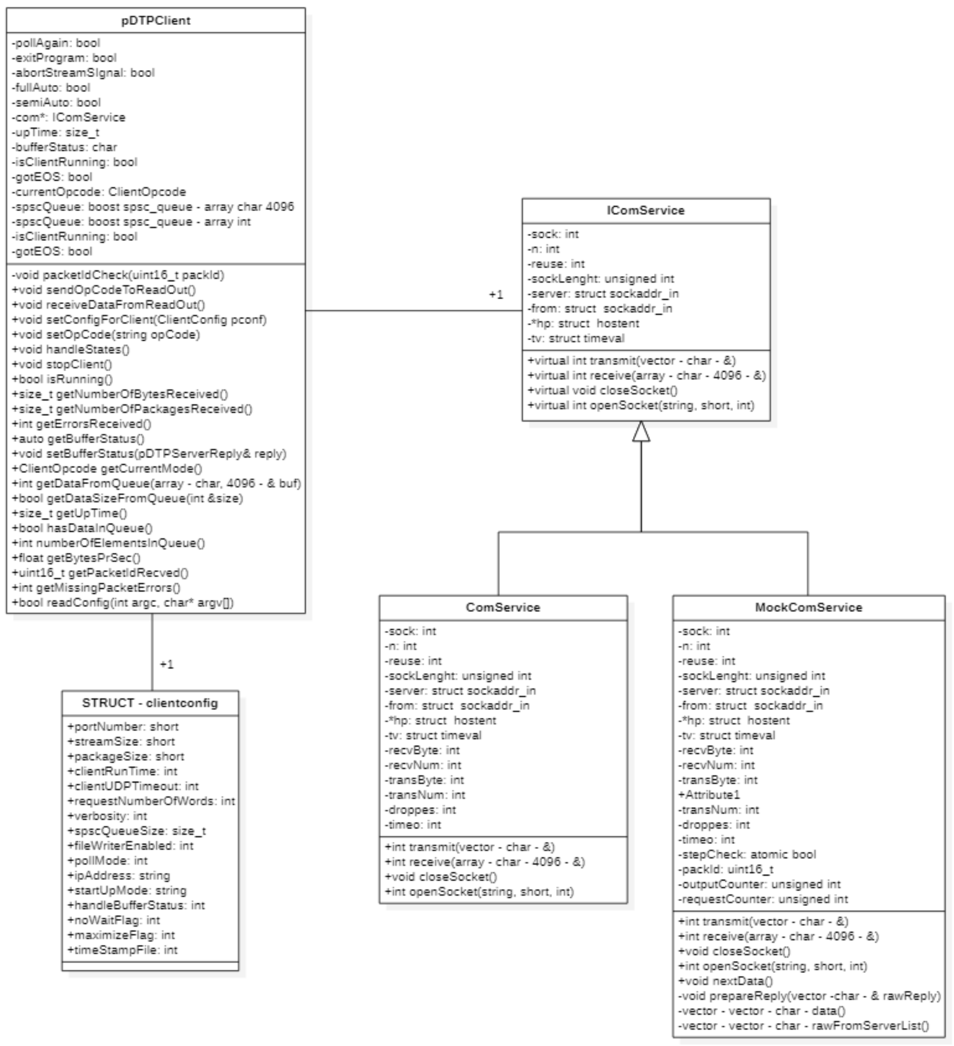
pDTPClient
-pollAgain: bool
-exitProgram: bool
-abortStreamSIgnal: bool
-fullAuto: bool
-semiAuto: bool
-com*: IComService
-upTime: size_t
-bufferStatus: char
-isClientRunning: bool
-gotEOS: bool
-currentOpcode: ClientOpcode
-spscQueue: boost spsc_queue - array char 4096
-spscQueue: boost spsc_queue - array int
-isClientRunning: bool
-gotEOS: bool
-void packetIdCheck(uint16_t packId)
+void sendOpCodeToReadOut()
+void receiveDataFromReadOut()
+void setConfigForClient(ClientConfig pconf)
+void setOpCode(string opCode)
+void handleStates()
+void stopClient()
+bool isRunning()
+size_t getNumberOfBytesReceived()
+size_t getNumberOfPackagesReceived()
+int getErrorsReceived()
+auto getBufferStatus()
+void setBufferStatus(pDTPServerReply& reply)
+ClientOpcode getCurrentMode()
+int getDataFromQueue(array - char, 4096 - & buf)
+bool getDataSizeFromQueue(int &size)
+size_t getUpTime()
+bool hasDataInQueue()
+int numberOfElementsInQueue()
+float getBytesPrSec()
+uint16_t getPacketIdRecved()
+int getMissingPacketErrors()
+bool readConfig(int argc, char* argv[])
+1
STRUCT - clientconfig
+portNumber: short
+streamSize: short
+packageSize: short
+clientRunTime: int
+clientUDPTimeout: int
+requestNumberOfWords: int
+verbosity: int
+spscQueueSize: size_t
+fileWriterEnabled: int
+pollMode: int
+ipAddress: string
+startUpMode: string
+handleBufferStatus: int
+noWaitFlag: int
+maximizeFlag: int
+timeStampFile: int
+1
IComService
-sock: int
-n: int
-reuse: int
-sockLenght: unsigned int
-server: struct sockaddr_in
-from: struct sockaddr_in
-*hp: struct hostent
-tv: struct timeval
+virtual int transmit(vector - char - &)
+virtual int receive(array - char - 4096 - &)
+virtual void closeSocket()
+virtual int openSocket(string, short, int)
ComService
-sock: int
-n: int
-reuse: int
-sockLenght: unsigned int
-server: struct sockaddr_in
-from: struct sockaddr_in
-*hp: struct hostent
-tv: struct timeval
-recvByte: int
-recvNum: int
-transByte: int
-transNum: int
-droppes: int
-timeo: int
+int transmit(vector - char - &)
+int receive(array - char - 4096 - &)
+void closeSocket()
+int openSocket(string, short, int)
MockComService
-sock: int
-n: int
-reuse: int
-sockLenght: unsigned int
-server: struct sockaddr_in
-from: struct sockaddr_in
-*hp: struct hostent
-tv: struct timeval
-recvByte: int
-recvNum: int
-transByte: int
+Attribute1
-transNum: int
-droppes: int
-timeo: int
-stepCheck: atomic bool
-packId: uint16_t
-outputCounter: unsigned int
-requestCounter: unsigned int
+int transmit(vector - char - &)
+int receive(array - char - 4096 - &)
+void closeSocket()
+int openSocket(string, short, int)
+void nextData()
-void prepareReply(vector -char - & rawReply)
-vector - vector - char - data()
-vector - vector - char - rawFromServerList()

Figure D.1: Implementation of the pDTPClient class.

APPENDIX E

Flame Graph Profile

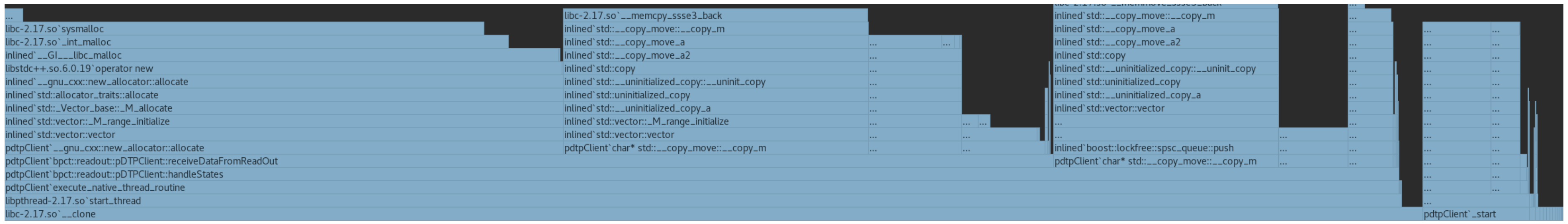

Figure E.1: Flame graph output profiling pDTP Client with `std::array`

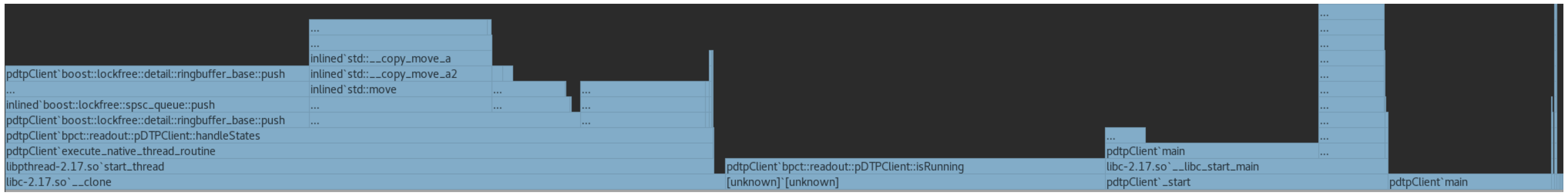

Figure E.2: Flame graph output profiling pDTP Client with `std::array`

APPENDIX F

Tuning commands

```
echo !!!!!!! WARNING THIS SCRIPT MUST BE RUN AS SUPER USER !!!!!!!
echo !!!!!!! WARNING THIS SCRIPT WILL TURN OFF THE FIREWALL !!!!!!!
echo !!!!!!! PLEASE REMOVE THE INTERNET CABLE FROM THE MACHINE !!!!!!!
echo STARTING TUNING SCRIPT

echo STOPING FIREWALL
service firewalld stop

sleep 1

echo SETTING NIC HARDWARE RING BUFFER
ethtool -G enp21s0f0 rx 4096

echo TURNING OFF NIC ADAPTIVE IRQ BALANCING
ethtool -C enp21s0f0 adaptive-rx off

echo TURNING OFF NIC IRQ MODERATION BY SETTING IT TO 0 uSEC
ethtool -C enp21s0f0 rx-usecs 0

echo TURING OFF NIC BUFFER BEFORE IRQ EVENT
ethtool -C enp21s0f0 rx-frames-irq 0

echo TURNING OFF NIC GRO
ethtool -K enp21s0f0 gro off


echo SETTING CORE BUFFER VALUE
sysctl -w net.core.rmem_max=1342177280
sysctl -w net.core.wmem_max=1342177280
sysctl -w net.core.optmem_max=134217

echo SETTING CORE BACKLOG VALUE
sysctl -w net.core.netdev_max_backlog=250000
echo SETTING CORE BUDGET
sysctl -w net.core.netdev_budget=100000

echo SETTING CORE UDP BUFFER SIZE
sysctl net.ipv4.udp_rmem_min=13107200000

#sudo sysctl net.ipv4.udp_rmem_opt=13107200
```

```
#sudo sysctl net.ipv4.udp_rmem_max=13107200

echo SETTTING CPU GOVENOR TO PERFORMANCE
```

Listing F.1: Tuning script for optimum network performance.