



**BERGISCHE
UNIVERSITÄT
WUPPERTAL**

A high data rate readout system for particle detectors
based on FPGA-to-server ethernet connections
and the eXpress Data Path technology

Dissertation zur Erlangung des Grades
Doktor-Ingenieur der Elektrotechnik (Dr.-Ing.)

vorgelegt der
Bergischen Universität Wuppertal
Fakultät für Elektrotechnik, Informationstechnik und Medientechnik

von
Carsten Dülsen

Erstgutachter : *Prof. Dr.-Ing. Dietmar Tutsch*
Zweitgutachter : *Prof. Dr. Wolfgang Wagner*

Tag der mündlichen Prüfung: 07. Mai 2021

Wuppertal, August 16, 2021



Abstract

With the Large Hadron Collider (LHC) at CERN being upgraded to the high luminosity LHC (HL-LHC), the four major experiment (ATLAS, ALICE, CMS and LHCb) use the opportunity to also upgrade their systems. In these so-called Phase II upgrades, the systems of each experiment are reworked or even replaced to enable the recording of a considerably increased event rate. In parallel, the resolution of the tracking subsystems will be increased to reach a better event separation. The ITk Pixel detector of the ATLAS experiment is used as example for this. The increased resolution and the increased number of front-end (FE) chips also leads to a drastically increased data rate, which is necessary to transfer all event information from the detector in a timely manner.

The upgrade is also used for replacing the readout and processing chain outside of the detector volume. Thereby, a unification of the different subsystems as well as a shift towards commercial products and software can be observed. This is due to the immense costs and increasing problems in maintaining the systems over a long time period due to the lack of system experts.

But there are some parts of the processing that still need to be done in hardware (e.g. within a field programmable gate array (FPGA)). Therefore, an interface between the hardware systems and the software running on a remote server needs to be implemented. While the PCIe interface was quite common for this, it limits the bandwidth as each processing card will only have a single PCIe interface. With the server hosting such cards is already busy with forwarding the data via a commercial network, it cannot provide any data processing capability.

Thereby, it would be more efficient to sent directly from the FPGA as this would also allow for better scaling of bandwidth. However, the standard protocols for reliable data transmissions were designed with a software implementation in mind. That's why their implementation in hardware would be rather complicated. Therefore, the question about the necessity of guaranteed data transmission arose.

To answer this question, a network stack was implemented within this thesis to investigate the level of packet drop occurring in the transmission and how this could be reduced to an acceptable level. In this context, an emerging technique named eXpress Data Path (XDP) was evaluated. With its help, the transmission of 5.2 PB (i.e. 2.92×10^{12} packets) within 168 h (i.e. a week) with not a single missing packet was demonstrated.

Contents

Introduction	1
1 Readout systems of detectors at the Large Hadron Collider	5
1.1 The Large Hadron Collider	6
1.2 The Standard Model of elementary particle physics	8
1.3 The ATLAS detector	11
1.4 Upgrade of the LHC and its effect on the ATLAS detector	13
1.5 Upgrade of the detector readout	16
1.6 Plans	19
2 Ethernet based Networks	23
2.1 The OSI model	23
2.2 The Transmission Control Protocol	27
2.3 Alternative Protocols	37
2.3.1 The User Datagram Protocol	37
2.3.2 The QUIC protocol	39
2.3.3 InfiniBand's RDMA over Converged Ethernet	42
2.4 Complexity of Retransmission	43
2.5 Development of the DROP protocol	45
3 Detector readout	55
3.1 The front-end chip	55
3.2 Overview of the readout system	62
3.3 Implication for the front-end interface	69
4 FREDDIE	73
4.1 Path towards the detector	76
4.2 Path from the detector	79
4.3 Network interface	87
4.4 Configuration path	93
4.5 Hardware implementation of the DROP protocol	94
4.6 Data generators	97
5 Data reception	101
5.1 System architecture	101
5.2 Network processing in Linux	104
5.3 eXpress Data Path	107
5.4 System configuration	110

5.5	Data processing	111
6	Tests and measurements	123
6.1	First Phase: Hardware Testing	124
6.2	Second Phase: Transmission characterization	132
6.3	Third Phase: Scaling up	136
6.4	Fourth Phase: Round up	143
7	Summary and Outlook	153
	Acknowledgments	157
	Bibliography	158
	Acronyms	163
	List of Figures	167
	List of Tables	169
A	LHC numbers	173

Introduction

During the last century, the rapid advancement of technology allowed science to gain both broader and deeper insights into nature. Especially the progress in electronics has enabled the extension of possible perspectives to study new phenomena. This is not only due to new sensors and better amplification, but also increased computational power enabling automation. But while technological progress helped to answer many open questions, the broader field of observation produced more and more new ones.

Black holes, which were found as one of the predictions of the theory of general relativity, are an example for this. They mark a situation in which both gravitational as well as quantum effects are in place, but until now it has not been possible to combine the relevant theories into a single one. It already took several decades to prove their existence and the first picture was only presented in 2019 [1]. Also, measurements proved the existence of binary black holes [2].

Related to black holes is the question of the matter distribution in the universe. Measurements show that more matter is needed (e.g. to explain the structure of galaxies) than can be accounted for by observable objects. Despite the name, according to current knowledge, black holes are not the solution to this problem of missing matter. Since missing matter does not interact with electromagnetic radiation, it is called *dark matter*.

A third example is antimatter and its unbalanced occurrence compared to normal matter. While the first antimatter particles were found as by-products of the general searches for new particles, these are generated on purpose nowadays in tiniest amounts (i.e. in the order of ng) for more detailed investigations. This includes measurements on the optical spectrum of an antimatter atom [3].

Such experiments are done in large laboratories as they need a highly complex infrastructure. One of these laboratories is the European Center for Nuclear Research (CERN), located near Geneva on the Franco-Swiss border. It operates a complex chain of particle accelerators with the Large Hadron Collider (LHC) being the final step. With a circumference of 26.7 km and a designed center-of-mass energy of 14 TeV, the LHC is currently the largest and most powerful particle accelerator in the world, being primarily built to collide two opposing proton beams. At the four collision points of the beams, the four major detectors (i.e. ALICE, ATLAS, CMS, LHCb) are located [4]. Notably, ATLAS and CMS played a key role in the discovery of the *Higgs* particle in 2012, which was the last missing particle of the Standard Model of elementary particle physics (SM) [5, 6].

Even if the SM is quite successful, it can still not explain all effects seen. For example, it does not give an answer on what the dark matter is built of.

With hope of changing this situation, the LHC is upgraded in several steps to increase both the maximum energy being released at the collisions as well as the number of collisions occurring within a period. The increase in energy is needed to allow heavier particles to be generated, hoping for new particles to be found. The increase in the number of events is needed to increase the occurrence of rare processes or even find new ones. Both upgrades set higher requirements for the detectors, so these also need to be upgraded in that process.

The experiments use the opportunity and exchange old detector parts to improve their performance. Thereby, they overcome existing limitations in the ability to select desired events. The ATLAS experiment for example plans to increase the first level trigger frequency by a factor of ten to get a broader range of events to select from. With the increased event rate and improved resolution of the detectors, the data rate needed to sent out the event information reaches unprecedented values [7]. Therefore, the entire processing chain is also reworked to take this into account.

In this process, a shift towards commercial hardware and standardized software solutions can be observed. The main reasons for this shift are cost reduction by unification as well as problems in finding qualified man power to develop and maintain the systems. So, more and more processing is done on commodity servers, using standard network protocols for data exchange. This also has the benefit of easier recovery in case of hardware failures as the network traffic can simply be routed to another server which takes over.

With some processing parts still needed to be implemented in hardware, the interface between these hardware instances and the processing software gets a special role as it must fulfill the requirements of both domains. The software side is very flexible, but has a rather coarse timing granularity (e.g. timeout counter given in ms or s). In contrast to this, the hardware is quite fixed (e.g. the optical fibers coming from the detectors are connected to a specific hardware instance), but achieves a very deterministic timing with an accuracy of ns or even better in some parts.

While the usage of the PCIe interface is quite common for interfaces between hardware and software, it requires a second conversion to transfer the data into the network and the processing software. As an alternative, FPGA-based systems offer support for low level network protocols but have issues when coming to the higher level protocols, which were designed for software based systems. With a PCIe card being limited to a single interface and the host PC not doing any processing, the massive increase in required data rate would result in hundreds of servers copying data from PCIe to network. Therefore, it seems worthwhile to investigate how to establish a stable communication between an FPGA and the readout software using standard protocols as far as possible.

Most network protocols are designed to be implemented in software as they are used to connect computers. Therefore, implementing these in an FPGA can be problematic. This is especially true for the TCP protocol, which is the default network protocol used for reliable data transmission. On the other hand, more hardware friendly protocols like UDP do not offer this reliability and would thus require an additional protocol on top to implement it.

This leads to the question, if the requirement of guaranteed data transmission is necessary. If the number of dropped packets could be reduced to an acceptable level, the overhead of the guaranteed data transmission could be avoided, resulting in less complex FPGA designs. Therefore, a test system is implemented in order to examine different approaches for reducing packet losses. In this context, an emerging technique named XDP is evaluated.

Chapter 1 starts with a more detailed description about the context of this thesis, being followed by the theory of communication protocols including a discussion on reliable data transmission in Chapter 2. After this, the data processing chain and the constraints from the surrounding systems are presented in Chapter 3 and how this translates into the field programmable gate array (FPGA) design, especially the network interface (Chapter 4). Chapter 5 then describes the receiving side of the network, i.e. the software (system and user space) running on the server. The measurements done and the corresponding results are discussed in Chapter 6.

Chapter 1

Readout systems of detectors at the Large Hadron Collider

Special accelerator complexes are used to accelerate particles to high energy and let them collide with each other or a fixed target to extend the understanding of modern physics. The results of these collisions are recorded using huge and highly complex detectors. The largest accelerator is the Large Hadron Collider (LHC) at the European Center for Nuclear Research (CERN) near Geneva on the Franco-Swiss border. Figure 1.1 shows an aerial view with the position of the four detectors.



Figure 1.1: Aerial view of the LHC. The interim stages of the accelerator chain as well as the position of the four detectors is also shown [8].

1.1 The Large Hadron Collider

The LHC is a circular accelerator with a circumference of 26.7 km and is located about 100 m below the surface. It was designed to collide two proton beams with a center-of-mass energy of 14 TeV and a peak luminosity of $10^{34} \text{ cm}^{-2} \text{ s}^{-1}$ but is also able to collide lead (Pb) ions for measuring the properties of the quark-gluon plasma [9]. The design luminosity was already reached and even surpassed by a factor of two within Run 2, while the center-of-mass energy only reached 13 TeV. The superconducting magnets of the LHC need more training to withstand the currents needed and therefore it is planned to achieve the 14 TeV in the next run starting in 2022 (see Table 1.1).

The luminosity is a measure of the number of interactions per area and time and is given in the unit of a particle current density. It is multiplied with the cross-section σ_{proc} of a certain process (e.g. only hard scattering collisions are of interest) to get the rate \dot{N}_{proc} of that process in the specific environment:

$$\dot{N}_{proc} = L \sigma_{proc} \quad (1.1)$$

The luminosity can then be calculated by a number of beam parameters:

$$L = \frac{N_1 N_2 f_{col}}{4\pi \sigma_x \sigma_y} \quad (1.2)$$

where N_1 and N_2 are the number of protons per bunch in the corresponding beams, f_{col} is the effective colliding frequency and σ_x and σ_y are the standard derivations of the transverse distribution of protons in the bunches at the collision points. With the transverse distribution being approximately a Gaussian profile, this translates into beam widths in the transverse plane. The nominal number of protons per bunch is 1.15×10^{11} for both beams and the effective colliding frequency is 40 MHz (i.e. collisions are occurring every 25 ns if the bunches are filled) multiplied with the filling ratio (i.e. ≈ 0.788 as 2808 bunches out of 3564 are filled). A complete list of the LHC parameters can be found in the appendix (Figure A.2 on page 175).

Table 1.1: Overview of the LHC runs. The planned numbers for run 3 and 4 are given [10, 11].

run #	duration	center-of-mass energy	luminosity (integrated/peak)
1	2009-2013	7 TeV and 8 TeV	$(30 \text{ fb}^{-1} / 7.7 \times 10^{33} \text{ cm}^{-2} \text{ s}^{-1})$
2	2015-2018	13 TeV	$(190 \text{ fb}^{-1} / 2.0 \times 10^{34} \text{ cm}^{-2} \text{ s}^{-1})$
3	2022-2024	(13 to 14) TeV	$(350 \text{ fb}^{-1} / 2.0 \times 10^{34} \text{ cm}^{-2} \text{ s}^{-1})$
4	2027-	14 TeV	$(4000 \text{ fb}^{-1} / 7.5 \times 10^{34} \text{ cm}^{-2} \text{ s}^{-1})$

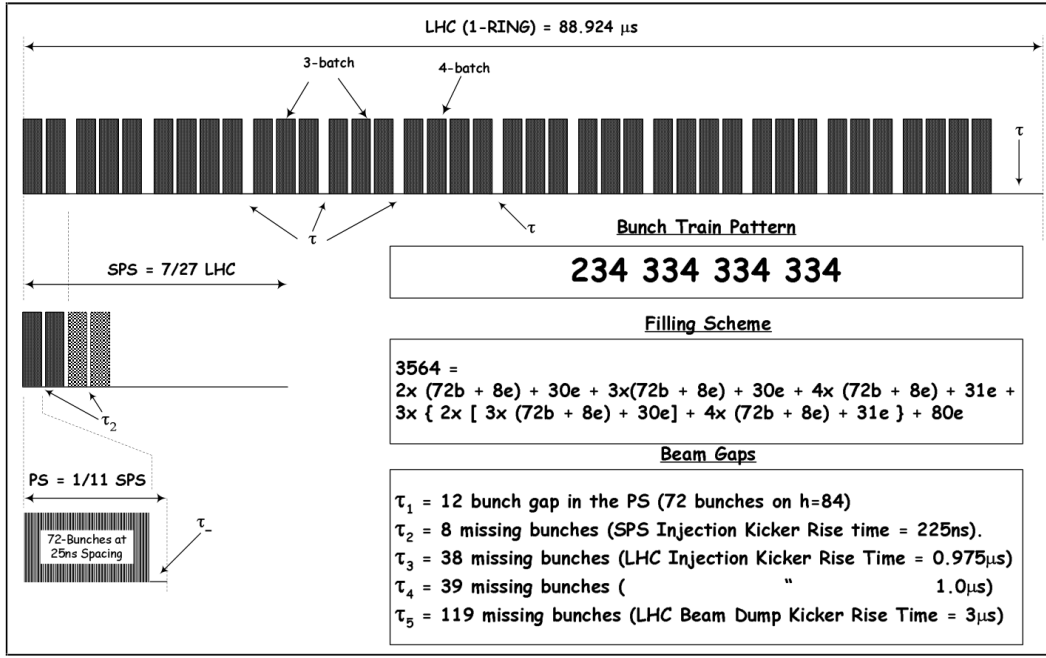


Figure 1.2: Nominal bunch structure of the LHC being introduced in 2003. Not every bunch is filled as gaps are needed in several stages of its construction [12]. This structure was changed over time to adapt for upcoming issues.

There are several ways to increase the luminosity and therefore the event rate for wanted processes. The number of protons per bunch looks like a very effective way as it gets squared in Equation 1.2. But an increased intensity also means a larger current and therefore higher beam induced heating in the superconducting parts. Due to the capacity of the cooling system, this method is very limited.

Increasing the filling factor is only hardly possible as the gaps are needed by several steps within the construction of the beams. Most important factors are the rise times of injector and kicker magnets in each acceleration step. Figure 1.2 shows an example of a beam structure scheme as introduced in 2003. These schemes are modified over time to adapt for changing conditions.

Another way is to reduce the diameter of the beams (i.e. the spatial distribution of protons within the bunches). If the bunches are packed more tightly it is more likely that the protons will collide, resulting in an increased number of collisions, the so-called *pile-up*. Since these additional collisions occur simultaneously within the same bunch crossing, more particles are generated at the same time when increasing the pile-up. Therefore, more tracks through the detector need to be reassembled and it gets more difficult to separate them. However, the individual collisions are not occurring all at the exact same location, but are distributed around the designated interaction point (IP). This can be seen in Figure 1.3, which shows how an event taken in 2017 with a pile-up of 66 looks like.

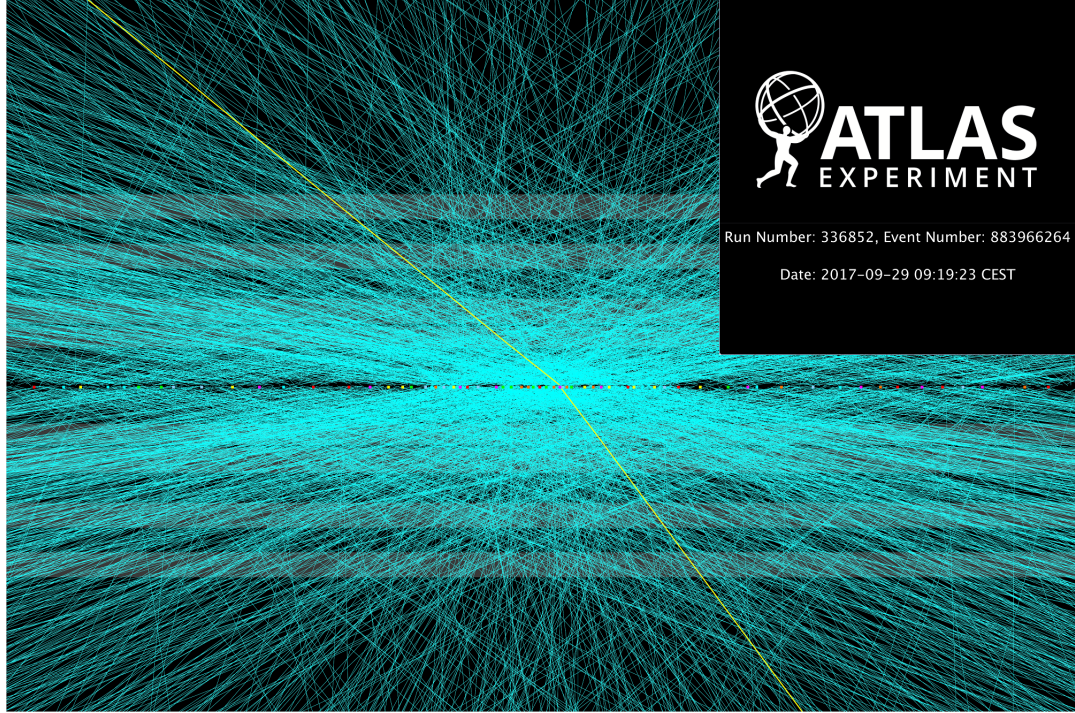


Figure 1.3: An event with a pile-up of 66 as measured and reconstructed at ATLAS on the 29th of September 2017 (Run 336852, Event 883966264). The offset between the single collisions (i.e. the colored spots in the center line) is clearly visible [13].

Besides the instantaneous luminosity, there is also the integrated luminosity as a measure for the amount of events generated (or captured if viewed from a detector perspective):

$$L_{int} = \int L dt \quad (1.3)$$

A high integrated luminosity is needed to have enough data available also for very rare processes. Therefore, it is regarded as one of the major goals to maximize the integrated luminosity during a data taking period.

1.2 The Standard Model of elementary particle physics (SM)

The interactions between two protons in hard-scattering collisions are described by the so-called *SM*. In this model, all particles found until today are sorted into groups with similar properties and are associated to the fundamental forces accordingly (see Figure 1.4):

The quarks are the building blocks of *hadrons* and can only be found (under normal conditions) in these composite particles. At very high energies, they can overcome the bound structures or solitary quarks can be formed as an intermediate state. The quarks carry both a *color* charge and an *electromagnetic* charge. Therefore, they can interact via all fundamental forces.

The leptons are the second group of matter particles with no *color* charge and do therefore not participate in strong interactions. They can be further divided into subgroups by their *electromagnetic* charge. The charged leptons are the *electron*, the *muon* and the *tau*. The neutral leptons are the neutrinos which interact only via the weak force.

The gauge bosons are the force particles. These are the *photon* for the *electromagnetic* force, the *W* and *Z* bosons for the weak interaction and the *gluon* for the strong interaction. They are also called *vector bosons* since they carry a spin of 1 and therefore have a distinguished orientation, in contrast to spin 0 particles which are called *scalars*.

The Higgs boson is the only boson without a spin and is therefore referred to as a *scalar boson*. According to the SM, it gives the particles their mass.

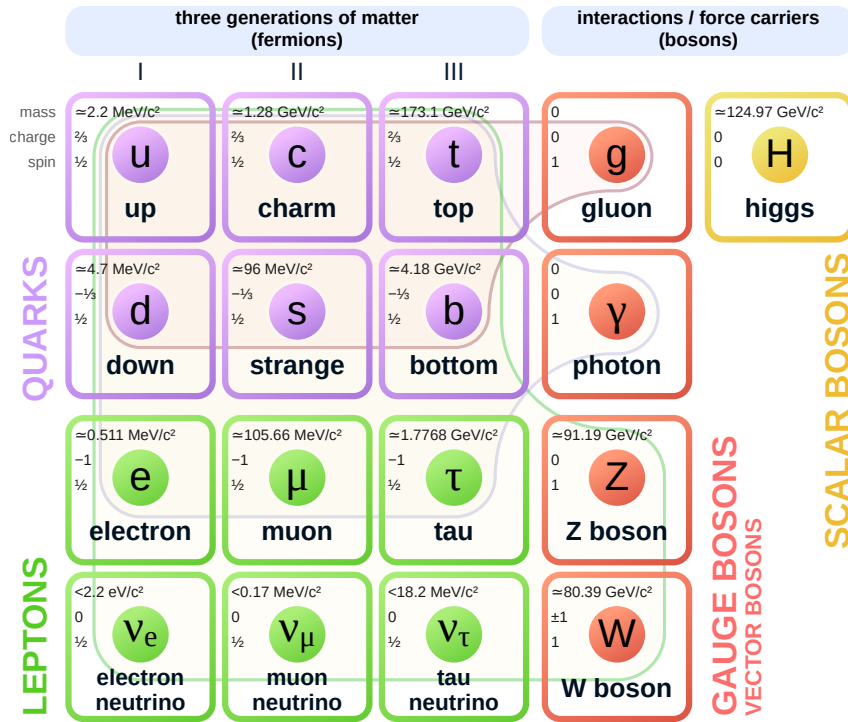


Figure 1.4: Summary of the elementary particles of the standard model [14]. The three generations of a type of particle have the same properties except the mass.

The hadrons are composite particles built-up of quarks. They are composed in a way that the combined *color* charge of the quarks is always zero. This means that they are only sensible to the strong interaction in situations where their internal structure is of importance. The best known hadrons are the *proton* and the *neutron*, which are also the lightest combinations of three quarks.

Only a few particles of the SM are stable enough to be measured directly. These are the so-called *(semi-)stable final state particles*:

- *proton* and *neutron* as hadrons
- *electron* and *muon* as leptons
- *photon* as the only boson

In additions to these, there are also some composite particles (e.g. *pions*, *kaons* or the *Lambda* baryons) with a mean lifetime up to 5×10^{-8} s. With a velocity near to that of light, these particles can live long enough to enter the detector.

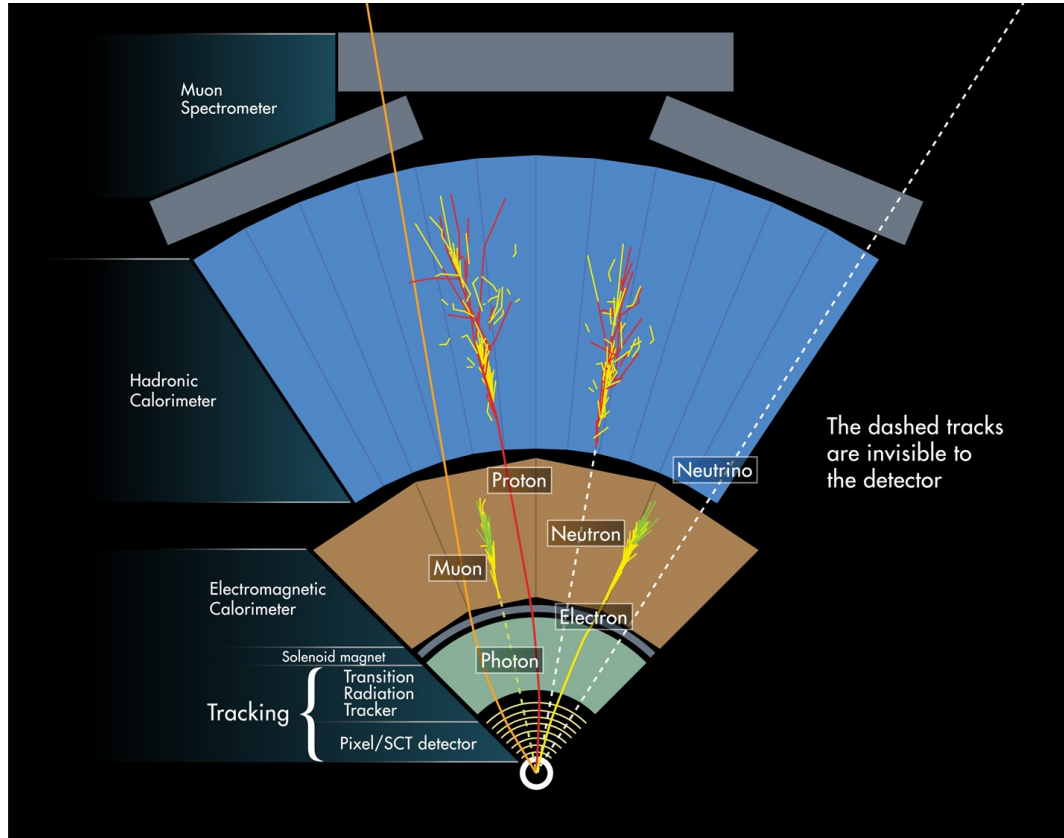


Figure 1.5: How ATLAS detects particles: diagram of particle paths in the detector [15].

Figure 1.5 shows an overview how the tracks and interactions with different detector components look like for the different particles.

1.3 The ATLAS detector

One of the four detectors recording the collisions at LHC is the ATLAS detector which is a general-purpose detector. It has a diameter of 25 m, a length of 44 m and a weight of 7000 t[16]. It is constructed rotation symmetric around the beam pipe as well as symmetrically in the forward and backward direction and uses a right-handed coordinate system with its origin at the nominal IP in the center of the detector. The aim is to measure the (semi-)stable final state particles of the hard-scattering collisions.

Because of the different properties of these particles, different types of detection methods are needed. This is the reason why the ATLAS detector is built-up of different subsystems. In total, the ATLAS detector consists of three mayor detector systems, which can be split further into sub-detectors, and has two magnetic systems [16]. From the IP outwards, these are:

- The first detector around the IP is the inner detector (ID) which is a tracking detector. This means that it delivers information about where a charged particle has passed through the sensor material. The detector material is ionized when charged particles pass through, generating free electric charges which can be measured. The tracking detector is a rather lightweight detector as it tries to minimize the interaction with the measured particles.

The first subdetector the particles are going through is the pixel detector made of silicon pixel sensors. Its innermost layer called *insertable B-layer (IBL)* was inserted in the long shutdown of 2013 and 2014 to improve the resolution of the pixel detector, allowing for more precise measurements of the primary collision points [17].

Around the pixel detector, there are the silicon strip sensors known as *semiconductor tracker (SCT)*, followed further outwards by the *transition radiation tracker (TRT)*.

- Surrounding the ID is a magnet system called the *solenoid* which creates an axial magnetic field of 2 T. This magnetic field forces all charged particles to have a bended path through the ID. The bending can be used to measure the momentum of the particles as well as the sign of their electrical charge.
- Located around the solenoid magnets are the electromagnetic and hadronic calorimeters. The former one is used to measure the energy of electrons and photons. To achieve this, it uses an accordion-shaped structure of lead absorbers filled with liquid argon as active material. The increased density of the lead plates increases the likelihood of electromagnetic interactions while the

liquid argon gets ionized by the secondary particles of these interactions. The electrons and photons are very light particles and therefore easier to stop.

The latter one is for hadronic jets created by protons and neutrons in strong interactions. It uses liquid argon and flat copper plates or scintillating tiles and steel absorbers, depending on the location in the ATLAS detector. The hadronic calorimeter occupies more space as protons and neutrons are harder to stop, but is still unable to stop muons.

Both calorimeters are contributing to the trigger system as a high energy deposition in the calorimeter is needed as an indicator that a heavy particle like the top quark was generated in a collision event.

- Outside the calorimeters is the second magnet system called *toroid*. It is made out of three elements: two end-cap magnets and the barrel toroid made out of 8 coils. They generate a toroidal magnetic field between 0.5 T and 1 T and are used to bend the path of the muons. Since muons produce less bremsstrahlung than electrons, they are not stopped within the detector. The only way to get information about their energy is the measurement of their momentum.
- Interleaved into the toroid magnets are the muon chambers. They detect not only muons generated by collisions but are also able to filter out muons coming from outside sources. This information is also used in the generation of trigger signals.

The systems are labeled in Figure 1.6.

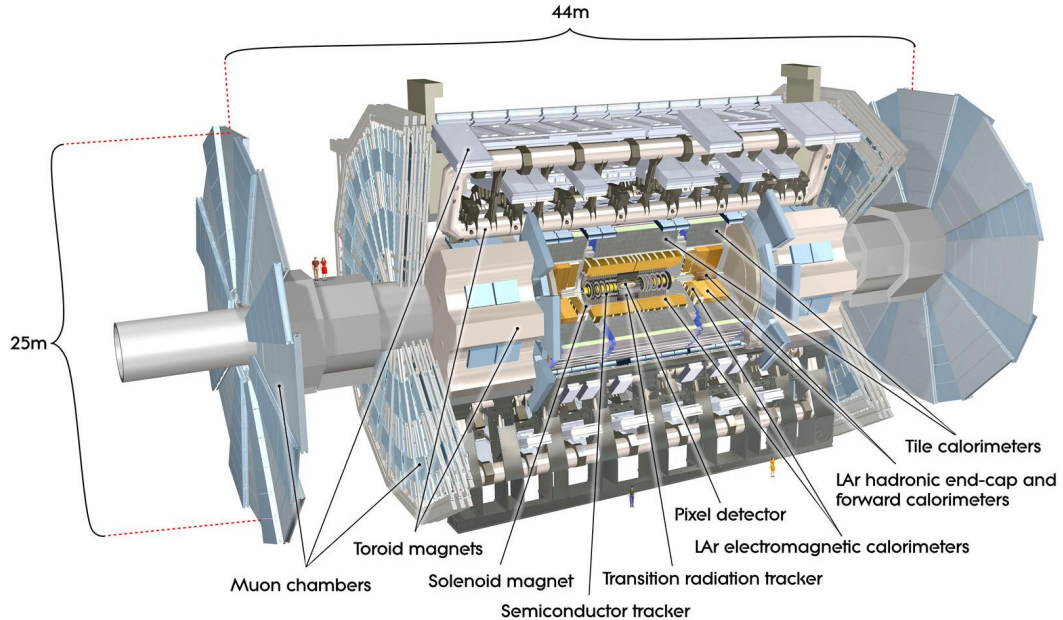


Figure 1.6: A computer-generated overview of the ATLAS detector [16].

1.4 Upgrade of the LHC and its effect on the ATLAS detector

To further increase the luminosity of the LHC, it will be upgraded to the so-called *high luminosity LHC (HL-LHC)* in the third long shutdown (LS3, around 2025-2027, see Figure A.1 in the appendix). This increase is necessary to get more data for rare events in an adequate period. As the LHC will have run for more than six years at the current luminosity level until the upgrade, it would take the same amount of time to double the data if the LHC would not be upgraded.

With the upgrade to the HL-LHC, the detectors also need an upgrade to cope with the increased luminosity. Besides changes in the outer systems, the whole ID will be replaced by the new inner tracker (ITk) in the so-called *Phase II* upgrade [7, 18].

A complete replacement is necessary for several reasons. The first reason is that the old ID already has some known problems and these will get even more severe with more irradiation. At some point, the leakage currents caused by radiation damage will prevent the proper operation of parts of the detector. If the fraction of faulty or dead detector areas gets too large, the detector would get useless. Current predictions say that some parts of the ID might reach that point already before being replaced.

The second reason is the fact, that the current ID could not handle the increased collision rate at the HL-LHC. To keep the same physics performance as the current ID under these conditions, a better resolution is required in the tracker. The number of inelastic proton-proton interactions per bunch crossing will increase to around 200, which is 4 to 5 times more than in the current setup. Figure 1.7 shows how an event with a pile-up of 230 would look like in the new ITk detector.

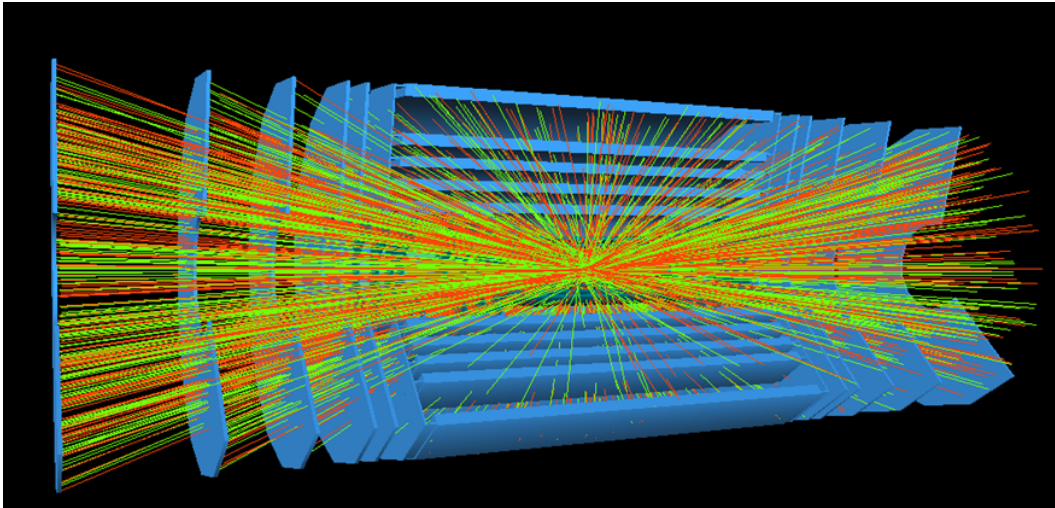


Figure 1.7: A simulated event in the ITk detector with a pile up of 230. In such a scenario, a very good spatial resolution is needed to reconstruct the trajectories of particles [19].

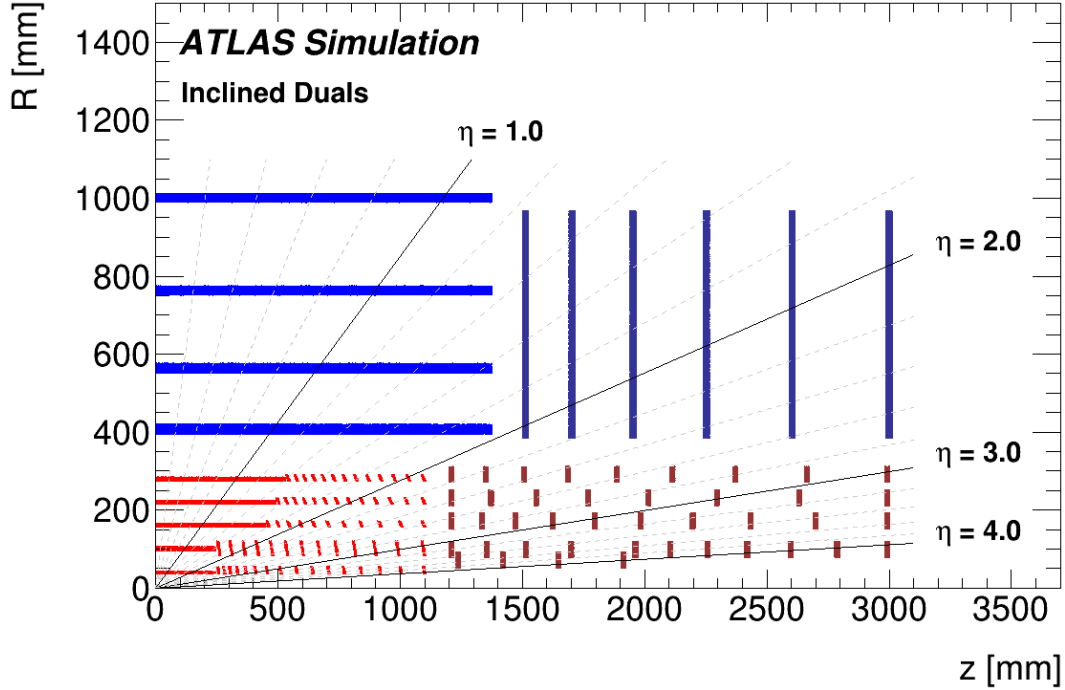


Figure 1.8: Schematic layout of the ITk [7]. One quadrant of active material is shown. The strip detector is shown in blue while the pixel detector is in red. Thereby, the so-called *pseudorapidity* (η) is a measure for the angle between a particles trajectory and the beam axis.

Another reason is the TRT and its basic detection principle not being suited for this high luminosity. Therefore, the replacement of the complete ID by a new tracking detector with a silicon strip and a silicon pixel section, scaled up accordingly, was regarded as being more beneficial as trying to replace each subdetector individually.

This results in five layers for the pixel detector and four layers¹ for the strip detector as shown in Figure 1.8.

In addition to the increased luminosity coming from the collisions, the portion of events which are read out and further analyzed will also be increased. This comes in form of an increase of the trigger rate from 100 kHz to 1 MHz. There is also an optional evolved scenario with a rate of 4 MHz for the first trigger stage, which foresees also the usage of hit information from the ITk in the trigger system [20].

Thus, the new ITk needs to be able to handle bigger events as well as more events being read out in the same time. With the ITk Pixel detector being the closest one to the IP, it needs the best spacial resolution and therefore needs the highest number of channels per cm² of active chip area.

¹These four layers are double-sided in the barrel region, so they generate twice the amount of hit information.

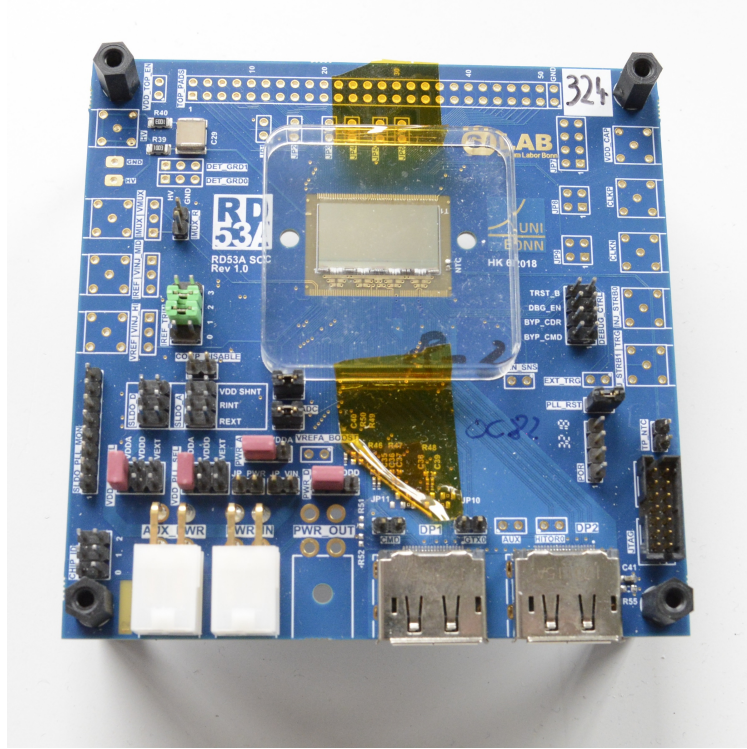


Figure 1.9: First prototype of the new FE chip for the ITk Pixel detector, mounted on a test card. The prototype chip has only half the height with respect to the final one.

New front-end (FE) chip for ITk Pixel

To be able to handle the new requirements a new FE chip for the ITk Pixel detector is designed. The pixel size for this chip is reduced from $250\mu\text{m} \times 50\mu\text{m}$ (as used in IBL) to $50\mu\text{m} \times 50\mu\text{m}$ (i.e. by a factor of five), while the overall chip area is roughly the same. But even with a data rate of 5.12 Gb/s and a sophisticated data compression scheme, the FE chips in the innermost layer are only capable to serve a trigger rate of 1 MHz. A picture of a first prototype of the new FE chip is shown in Figure 1.9.

About 6000 optical fibers will be needed to read out the ITk Pixel detector (the fibers used to send commands not included). With a usable data rate² of 7.68 Gb/s each, the whole detector will generate a total bandwidth on the order of 50 Tb/s. The data rate will be even higher when the data is decompressed and formatted in a more CPU-friendly way.

As a result, the readout and processing chain needs to be replaced entirely by a new one with drastically increased capabilities.

²The signal is sent with 10.24 Gb/s, but only up to 75 % of the bandwidth are used for hit information due to a large overhead caused by forward error correction (FEC).

1.5 Upgrade of the detector readout

When thinking about a complete overhaul of the data processing chain, there is not only the increase in data rate. Operating the ATLAS detector for about 10 years revealed also some other issues which should be addressed.

Problems related to custom solutions

The highly specialized hardware for each subdetector gets an increasing issue with the raising number of processing units. Design and testing of these is a costly task and the numbers needed are not high enough for beneficial effects from mass production. So, reducing the amount of different designs as well as switching to commercial productions instead of custom built hardware would save a lot of money and effort. This also reduces the number of experts needed for maintaining the system.

Additionally, the long time of development and operation of the hardware makes it difficult to find someone able to fix problems occurring with changing operation conditions. The experts who have designed the system have turned most often to other projects or left the experiment completely. On the other hand, new engineers joining the experiments are not familiar with the old hardware (neither the design itself nor the components used). Therefore, there are upgrades for smaller parts of the system from time to time to increase the availability of both experts and processing power. Figure 1.10 shows an example of such an upgrade. The original *Silicon ReadOut Driver (SiROD)* was used by Pixel and SCT from the start of ATLAS. With the introduction of the IBL came an upgraded *ReadOut Driver (ROD)*. As this

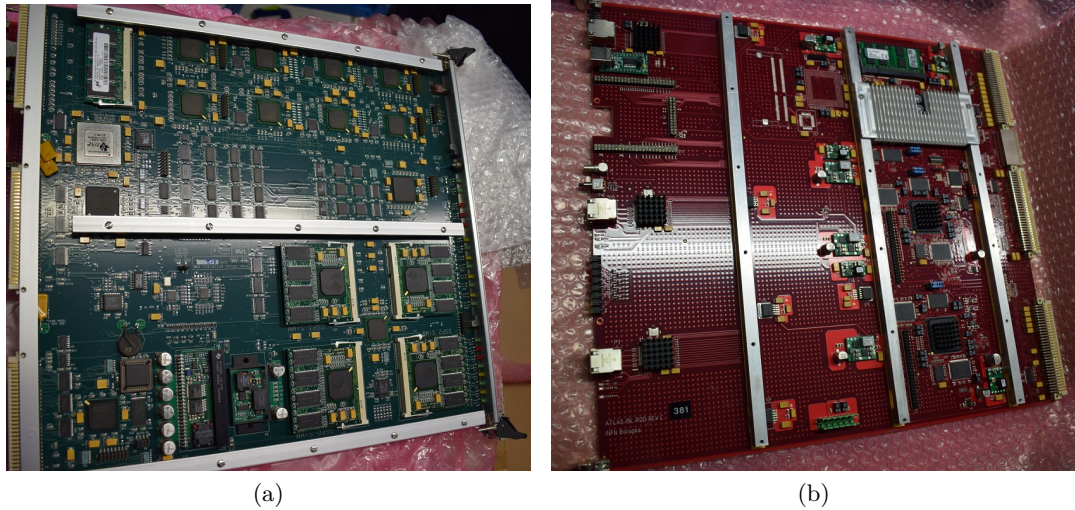


Figure 1.10: a) The original *Silicon ReadOut Driver (SiROD)* used by Pixel and SCT.
b) The upgraded *ReadOut Driver (ROD)* of the IBL, which was later on also used for the remaining Pixel detector.

is able to serve twice the modules with drastically reduced complexity, it was also used to upgrade the readout of the remaining Pixel detector³.

The highly specialized hardware also means custom protocols to be used for communication. General purpose protocols would require either the implementation of features not needed by the hardware, increase the system complexity or to have an incomplete implementation violating the standard. So, new protocols are created by adjusting the standards fitting best for the problem. But using such protocols also requires the opposing component to be able to understand these what can imply the need for special hardware on that side, too.

Reducing the fraction of custom built hardware and protocols will lead to higher usage of software components as these are more flexible. This is also boosted by the drastically increased processing power of modern servers⁴ compared to the ones available at the start of the LHC. The usage of standardized communication protocols enables the use of switched networks what helps in case of failure as a faulty server can be replaced by a spare one just by changing the network address. This saves a lot of event data being lost otherwise by disabling a fraction of the detector or stopping data taking completely if the fraction is too big. The faulty server can be taken care of whenever is time to do so without interrupting the rest of the system.

But the custom built hardware cannot be completely replaced by commercial components as the detector itself will still be a custom design. So, the FE chips still need highly specialized protocols as these protocols need to be minimalistic and error tolerant at the same time. Any inefficiency in the protocol will directly increase the total bandwidth generated by the detector.

In addition to the usage of a highly specialized protocol, there is the need for reading out the FE chips with a fixed latency in the command path. An event to be read out is determined by the time at which the command arrives in the FE chip. So this needs a precision better than 25 ns, which excludes any pure software solution. In order to have at least some flexibility, special programmable logic integrated circuits (ICs) called *field programmable gate arrays (FPGAs)* are used. The modern variants of the FPGAs are also able to handle standardized high-speed protocols, making them the perfect link between the custom and the commercial world.

The Front-End Link eXchange (FELIX) system

A first step to solve these challenges is already made as the first phase⁵ of the ATLAS upgrade is currently being installed. From the readout perspective, the most important change is the introduction of the FELIX system for all upgraded detector parts. It serves as a PC-based gateway linking the custom front-end electronics on one side with the software running on commercial servers on the other side [21].

The FELIX system consists of commercial servers running the FELIX software

³The SCT is still using the SiROD.

⁴This is related to both CPUs and GPUs.

⁵The Phase I upgrade is done for some of the muon systems and the calorimeters, contributing to the trigger decision. The ID and its readout are not touched by this.

and housing the FELIX card (a custom PCIe card). The software receives data and commands like the configuration of the FE chips via network and forwards it through the PCIe bus to the FPGA on the PCIe card. The firmware running on the FPGA takes these commands, adds the line encoding and sends it out towards the FE chips. The FELIX card has also an interface for the Trigger, Timing and Control (TTC) system for timing critical commands like the global trigger for event readout. The data coming back from the FE chips has its line encoding removed and is forwarded via PCIe into the host memory. The FELIX software then sends it over the network to the next processing part (see Figure 1.11). According to the technical design report of the *Phase-I* upgrade, there will be 1324 optical links served by 64 FELIX cards [22].

This approach accounts for the reduction of custom protocols and hardware as the FELIX system will replace the individual systems for each subdetector. FELIX also includes only the timing critical processing and does not touch the actual event data. The event processing is done in software on different servers. The *Phase I* version of the FELIX system still needs to fit into the existing processing infrastructure as only parts of ATLAS are upgraded. It is not designed to take care of the increased trigger rates coming with the *Phase II* upgrades. So, the whole system will be upgraded

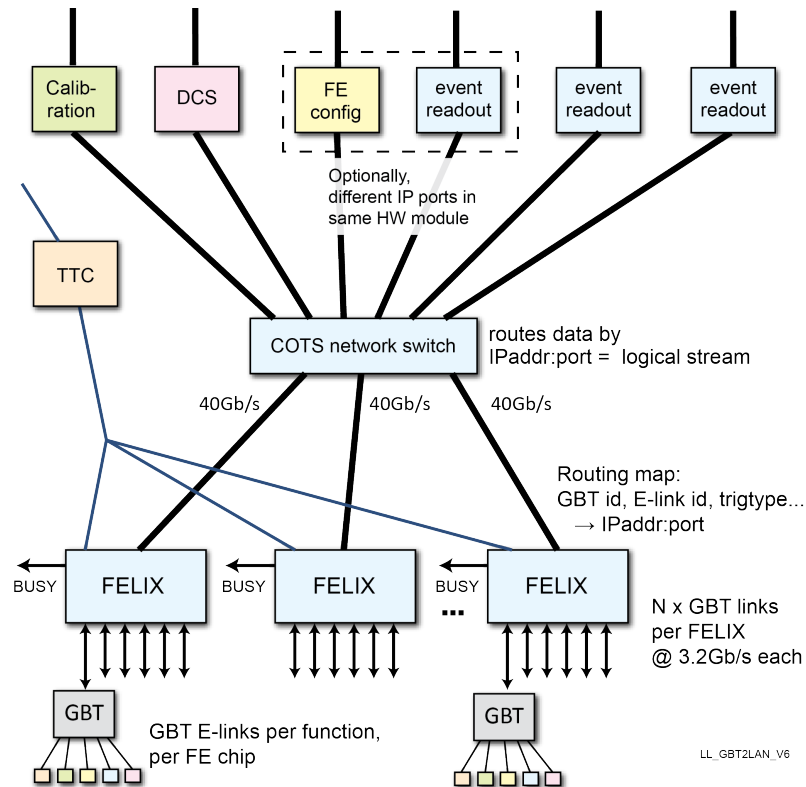


Figure 1.11: Architecture of the Phase-I FELIX system [23].

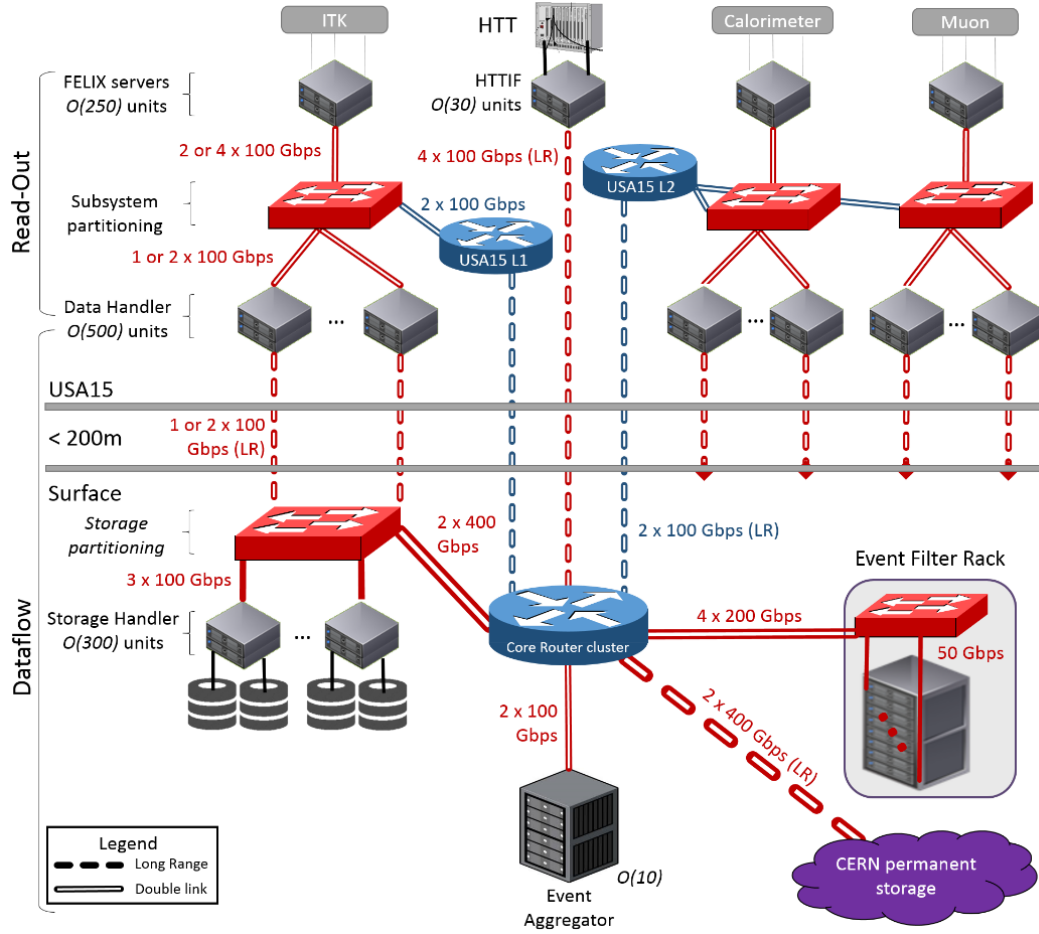


Figure 1.12: Proposed architecture of the ATLAS Phase-II readout network [20].

again.

At this point, all the subdetectors will switch towards the FELIX system as a common readout. As ITk Pixel alone will add around 6000 optical links, the whole system also needs to be scaled up substantially. This is not only in number of FELIX cards, but also includes the network infrastructure and processing system. The proposed network architecture is shown in Figure 1.12.

1.6 Plans

As the FELIX software does not touch the data being transferred, there are also some thoughts to switch to different architectures for the FELIX system. So, the FPGA itself is capable of housing a network interface which could be used to directly connect to the processing servers. The software handling the administrative tasks could run on a CPU being integrated within the FPGA. If the logic to control the routing of the

data can be implemented as firmware, the software can even be placed on a remote server. Such a system design could be realized with a crate architecture to achieve a higher packing density and therefore less space needed in the underground cavern.

But there is an issue in implementing some of the traditional network protocols as these are designed as a software solution. They make use of features being easily available in software, such as large memory blocks used for buffers or timeout counters for each network packet. As an FPGA solution cannot provide these as easy as a software implementation, another path is needed. When searching for the parts using such features most often, these can be identified to be the mechanisms for reliable data transmission quite quickly. So, moving to a crate-like architecture also comes with a strong push towards unreliable data transmission. This means at first only, that there is no guarantee that the packets arrive correctly. It does not include any assumption about the level of packet loss to be expected. There should be no problem as long as the relative data loss is small enough. With the detector readout chain having also other potential sources for data loss, the processing needs to be able to cope with this up to a certain point.

This thesis addresses such a crate-like architecture. It discusses what components are needed to build such a system and how these could be implemented. The designed system uses first prototypes where available as the interfaces for the final system are not fixed yet. Moreover, this thesis will focus on the network part as it is the least specified interface and evaluates the data loss there.

Chapter 2

Ethernet based Networks

There are many different functions needed for successful communication. The information needs to be packed into a message and sent out to the desired medium with the correct modulation scheme. It should also include some address information such that the receiver can determine if this message is meant for it or not. On the other hand, the receiver needs to be listening for new messages, needs to be able to separate a single message and afterwards demodulate and decode the message to get the information. These requirements are true for all kinds of communications independent of which kind the sending and receiving part are and which medium is used.

2.1 The OSI model

The communication between (technical) communication devices is defined by a formalized model grouping the different functionalities. This model is called *Open Systems Interconnection (OSI) model* [24]. It divides the process of communication into layers and defines which functionality should be provided from the underlying layer. There are seven layers defined with their specific functionalities, which are shown in Table 2.1 on the next page.

The layers of the OSI model

The highest layer (layer 7) is the *application layer* which provides the application programming interface (API) towards other application parts. It is also the only layer having any high level means of the object being accessed on the other side of the transmission. All other layers are just getting a chunk of data (i.e. the payload) to be transferred from the upper layer.

Layer 6 is the *presentation layer* taking care of how the information is represented. The data is translated into a common representation to abstract from any host specific elements like different encodings or endianness (i.e. the order of bytes in the memory) used by the operating system. Later on, also features like compression and encryption were added to this layer as these are also some kind of transcoding.

The next layer (layer 5) is the *session layer*. It keeps all information of transfers building a logical group. This includes management of connections (opening/closing as needed, also in parallel) as well as sequences of transfers over the same connection (e.g. negotiation of parameters for encryption or compression).

Table 2.1: Layers of communication according to the OSI model and how a single data unit is called within these.

# of layer	name of layer	function
7	application	high-level interface to the application, including resource sharing and remote access
6	presentation	representation of data, including character encoding, data compression and encryption
5	session	managing continuous exchange of information between two nodes, including parallel connections
4	transport	(reliable) transmission between end points, including segmentation, acknowledgement and multiplexing
3	network	managing of multi-node networks including addressing and routing
2	data link	transmission of data frames between neighboring nodes
1	physical	representation of symbols on the physical medium

The *transport layer* is the 4th layer and provides the end-to-end connections. It is the lowest layer dealing with complete end-to-end transfer as all lower layers may handle only a partial section in case there are more than one. Therefore, it also has to ensure a reliable transfer (if desired) and a way to specify the source and destination at the corresponding end point. This layer is also the highest layer being involved in actual data transfers and thus splits the data into segments on the transmitting side and delivers these in the desired order on the receiving side. This also includes the identification and removal of duplicate segments.

Layer 3 is the *network layer* and is dedicated to finding the addressed destination as well as the best way to reach it. Therefore, it provides an address for each node which should be unique within the same (sub-)network. There are also addresses which identify a group of nodes in which case all of these will get a copy of the packets. The destination is not necessarily the endpoint of the connection, but can also be a routing device connecting two different networks. Another important part is that the packets are transferred independently from each other. The routing of two packets from the same layer 4 connection can be different. This means that these packets can arrive in any order and can both get lost or received multiple times. In case a partial section of the transmitting path supports only smaller packets sizes, the packet will be split up and each part is transmitted as independent unit. The parts need to be assembled into the original packet before being delivered to the layer above it. If a single fragment gets dropped, the whole packet will be regarded as lost.

The *data link layer* (layer 2) is responsible for transferring data between directly linked nodes within the same (sub-)network. It controls the access to the transmission medium and therefore also the data flow as it has to adapt to the available bandwidth

Bit Offset	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	Version			Header Length			Type of Service								Total Length																	
32	Identification																Flags		Fragment Offset													
64	Time To Live								Protocol								Header Checksum															
96	Source address																															
128	Destination address																															
160	Options (0-40 Bytes)																															
192																																

Figure 2.1: Structure of the IPv4 header as defined in Ref. [25]. The most important fields are the source and destination address, the nested protocol and the total length.

of the medium. In case of a shared medium, a method to detect and avoid or at least to recover from collisions is needed (i.e. parallel transmission from two or more nodes at the same time). It also has to detect errors which may have occurred in layer 1.

The lowest layer (layer 1) is the *physical layer* which takes care of transmitting the raw data over the physical medium. It therefore translates (groups of) bits into physical quantities such as electrical voltage. This translation depends on properties like transmission distance, physical data rates, modulation scheme. In addition to this translation, it also defines things like connectors and their pinout to be used or if the transmission is simplex (unidirectional), half duplex bidirectional (with a single direction at a time) or full duplex (both directions at the same time).

Properties and Examples

An important part of the model is that each layer should only access the directly underlying layer and provides services for the one above. If a function from other than these are needed, the layers in between need to provide a transparent interface forwarding the requested data. Also, if a layer does not need the underlying layer to fulfill the requested transfer (e.g. the network layer notices that both endpoints are on the same device, i.e. an internal loopback is done) these will not be involved in that transfer.

The defined functionalities of each layer are implemented in one or more protocols. An example of such a protocol is the *Internet Protocol version 4 (IPv4)* which is defined in Ref. [25] (See Figure 2.1). It implements big parts of the network layer by providing the node addresses, grouping of addresses into subnetworks as well as

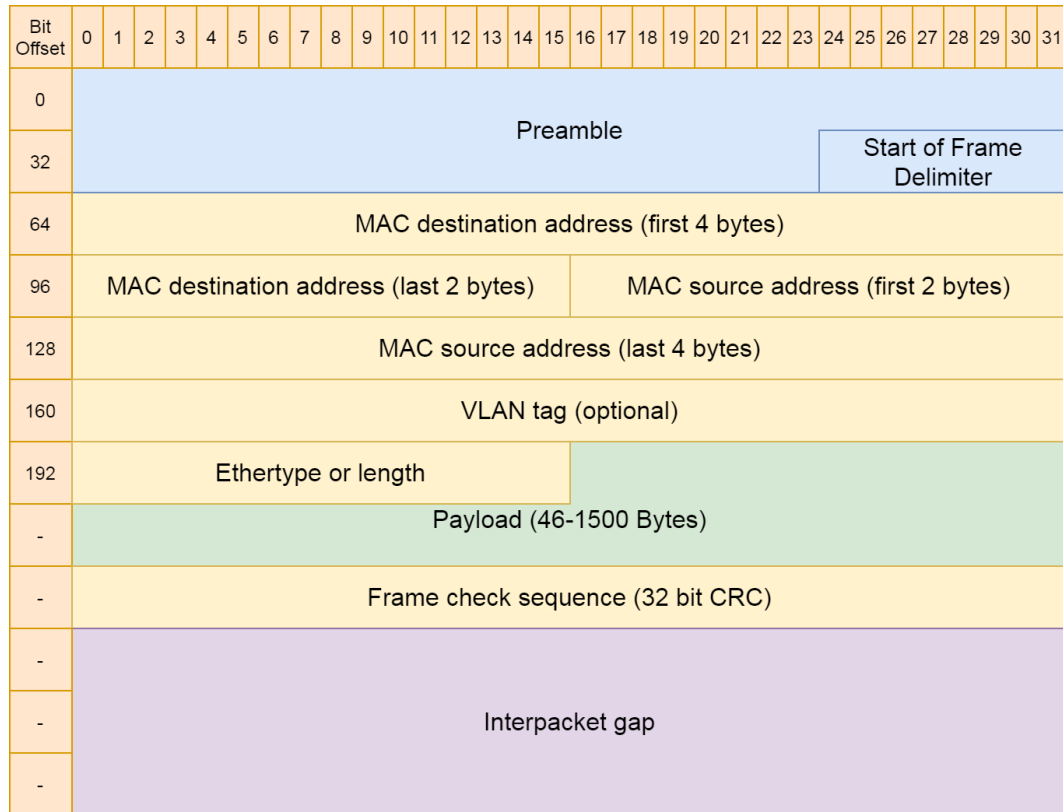


Figure 2.2: Structure of an Ethernet frame as defined in [26]. The yellow fields are put around the payload (green part) and both are forming the layer 2 frame. On layer 1, the Preamble and Start of Frame Delimiter (blue fields) are added to form the transmitted packet. There is gap with a length of at least 12 bytes between two packets, the interpacket gap.

fragmentation of packets¹. Some addresses serve a special purpose like loopback or multi- and broadcast transmission. IPv4 also handles cases where the current packet is too long for an intermediate part of the transmission path.

There are also some protocols serving more than one layer. This is often the case for layer 1 and 2 as the data format to be used in the data link layer often depends on the properties of the physical medium. A typical case for this is the detection of collisions on a shared medium. A collision occurs when two or more nodes try to start a transmission within a small time window (i.e. before the signal from one transmitting node has arrived at another one). To properly detect such a collision at each node the minimal duration (i.e. the minimal length divided by the data rate)

¹A packet might pass several subnetworks on its route to the destination. If the allowed packet size in one of these subnetworks is smaller than the actual packet size, the packet is split into several smaller packets called fragments. Each fragment is then routed individually. At the destination node, the user data of original packet is reassembled again.

must be twice of the propagation time a signal needs from any node to reach any other node. This puts tight limits on the maximum cable length for a given minimal packet length and data rate.

The *Ethernet* protocol is an example for such a protocol serving layer 1 and 2 [26]. It defines a minimal packet length of 72 byte for layer 1. This minimal packet size on layer 1 directly translates into a minimal frame size within layer 2. The structure of these frames is shown in Figure 2.2.

A way to get around the problem of collisions was the introduction of switched networks instead of a shared medium as the network switch decouples these so called *collision domains*. With a single node² per port of the network switch and full duplex operation, it is guaranteed to have no collisions as there is only a single transmitter per wire pair. With increasing data rates this is the only viable solution as the network diameter³ would shrink too much with the fixed frame size of Ethernet.

There are also protocols which are hard to assign to a specific layer as their function touches two neighboring layers. One of these functions is the translation between layer 2 and layer 3 addresses. As layer 3 relies on the underlying layers to do the actual data transfer, it needs to find the corresponding layer 2 address of the next node in the transmission path. In case of Ethernet and IPv4, this translation is taken care of by the address resolution protocol (ARP) [27]. ARP uses broadcast messages to find the layer 2 address of the network node having a requested layer 3 address. These messages include the layer 2 and layer 3 addresses of the requester as well as responder, with the layer 2 address of the responder kept blank within the request. ARP is regarded as belonging to layer 2 as it is sending its messages only to nodes within the same subnetwork (i.e. to direct neighbors).

Within this thesis, if not stated differently, the usage of Ethernet and IPv4 will be assumed as these are the most commonly used protocols in (wired) networking. This includes the usage of ARP for the translation of its addresses.

2.2 The Transmission Control Protocol (TCP)

When it gets to reliable data transfers there is no way around the transport layer as it has to provide the end-to-end communication. The most often used reliable protocol in IPv4-based networks is the Transmission Control Protocol (TCP) as both were developed together.

The TCP was developed together with IPv4 as the protocol to serve the transport layer in the 1970s. It is a connection-orientated protocol and provides reliable and ordered end-to-end data transfer, flow control as well as multiplexing of addresses on the end points. With all the functionalities within a single protocol, there comes also a higher complexity.

²This can also be a network switch with a number of other nodes connected to it.

³As each link has a maximum length due to effects like signal quality or requirements of collision detection, the maximum allowed distance between two endpoints is also given by the number of network switches in between. This distance is called *network diameter*.

Bit Offset	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	Source port																Destination port															
32	Sequence number																															
64	Acknowledgment number																															
96	Data Offset				Reserved				U R G	A C K	P S H	R S T	S Y N	F I N	Window Size																	
128	Checksum																Urgent pointer															
160	Options (if data offset > 5)																															

Figure 2.3: Structure of the TCP header as defined in Ref. [28]. It was extended later on by adding new flags (e.g. for congestion notifications).

The segment header fields

Figure 2.3 shows the TCP header as defined in Ref. [28]. Some additional flags were defined later on which are omitted for now. The same applies to some fields like the urgent flag and pointer as their usage is optional and rarely used. The usually implemented fields within the TCP header are:

The Source port is a 16 bit address and used to specify the application on the sending node.

The Destination port is a 16 bit address and used to specify the application on the receiving node.

The Sequence number is the position of the first byte of data within the data stream. The initial sequence number is transmitted during connection establishment (when the SYN flag is set) and may not be zero. The first byte of data sent has the position of the initial sequence plus one.

The Acknowledgment number is the next sequence number expected by the sender and is only valid if the ACK flag is set. It also acknowledges that all data before this sequence number has arrived.

The Data offset gives the start of the data field within the segment and is given in numbers of 32 bit words. The presence of options is assumed in case of a value larger than 5 as the remaining header occupies 20 bytes. The options field has a maximum size of 40 bytes as the data offset has a width of only 4 bit resulting in a maximum offset of 60 bytes.

The Flags are used to signal special cases or the presence of valid data in some other fields.

Bit Offset	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	Source address																															
32	Destination address																															
64	Zeros								Protocol								TCP length															
96	Source port																Destination port															
128	Sequence number																															
160	Acknowledgment number																															
192	Data Offset		Reserved								U	A	P	R	S	F	Window Size															
											R	C	S	H	T	N																
224	Checksum																Urgent pointer															
256	Options (if data offset > 5)																															
...	Data																															
...																																

Figure 2.4: Depicted is the pseudo header used for the TCP checksum calculation. The purple fields are the IPv4 header fields used for this. The checksum field is set to zero for the calculation.

The ACK flag indicates a valid acknowledgment number.

The RST flag indicates that the current connection is reset.

The SYN flag indicates the establishment of a new connection. Other fields have special meanings if this flag is set.

The FIN flag indicates the last segment sent by the sender. The connection is closed afterwards.

The Window size gives the amount of free space within the receive window (See Figure 2.8) in multiples of *window size units*. This is related to the segment identified by the acknowledgment number and therefore defines the amount of data that the receiver can store beyond that segment. A *window size unit* defaults to be a byte but can be scaled up by using the corresponding option.

The Checksum is used to validate the correctness of the header and data. It uses some additional fields from the IPv4 header to ensure it is correctly delivered to the intended destination. The resulting pseudo header is shown in Figure 2.4.

The Options are used to change the default behavior of a TCP connection. A selection of these are:

The Maximum Segment Size (MSS) option is used to announce the maximum size of a segment the sender is willing to receive. It is usually set in a way that fragmentation in IPv4 is avoided as this could lead to an increased number of retransmissions.

The Window Scaling option is used to change the window size unit by defining the number of bits to left-shift the window size field. Valid values are from 0 to 14. It is only allowed during the establishment of a new connection and both sides must use this option to enable the usage for either direction.

The Selective Acknowledgment option is used to acknowledge blocks of segments which were received correctly but discontinuously. This tells the transmitting side, that only the missing segments need to be retransmitted. The usage of selective acknowledgments needs to be enabled during the establishment of a new connection.

The Connections

The finite state machine (FSM) describing the different states a TCP implementation needs to have is a rather complex one. Figure 2.5 gives a simplified view including only a few less frequent used transitions.

The establishment of a new connection differs between both sides of the communication channel. Before the active side (called *client*) is trying to establish the connection, the other side (the passive one, called *server*) has to be ready for this or the attempt will fail. Thereby, it is independent if the connection is established by the sender (i.e. the client wants to upload some data to the server) or the receiver (i.e. the client wants to download some data from the server) as data can be sent in both direction.

The server is going the passive way and goes in the *listen* state by registering a port number. After this, the server can be reached by the IPv4 address and the TCP port number.

The client is doing the active part by sending a request towards the server. This request is an empty segment with the *SYN* flag set and a random number as the initial sequence number. It will get a random port number which (together with the IPv4 address of the client) identifies the client.

On arrival of a segment with the *SYN* flag set, the server replies an empty segment with the *ACK* flag set and the received sequence number plus one as acknowledgment number to acknowledge the *SYN* from the client. It also sets the *SYN* flag and a random number as sequence number in the reply to start the synchronization in the opposite direction.

When the client receives the response from the server with both flags (i.e. *SYN* and *ACK*) set, it switches into the *established* state. It also sends an acknowledgment

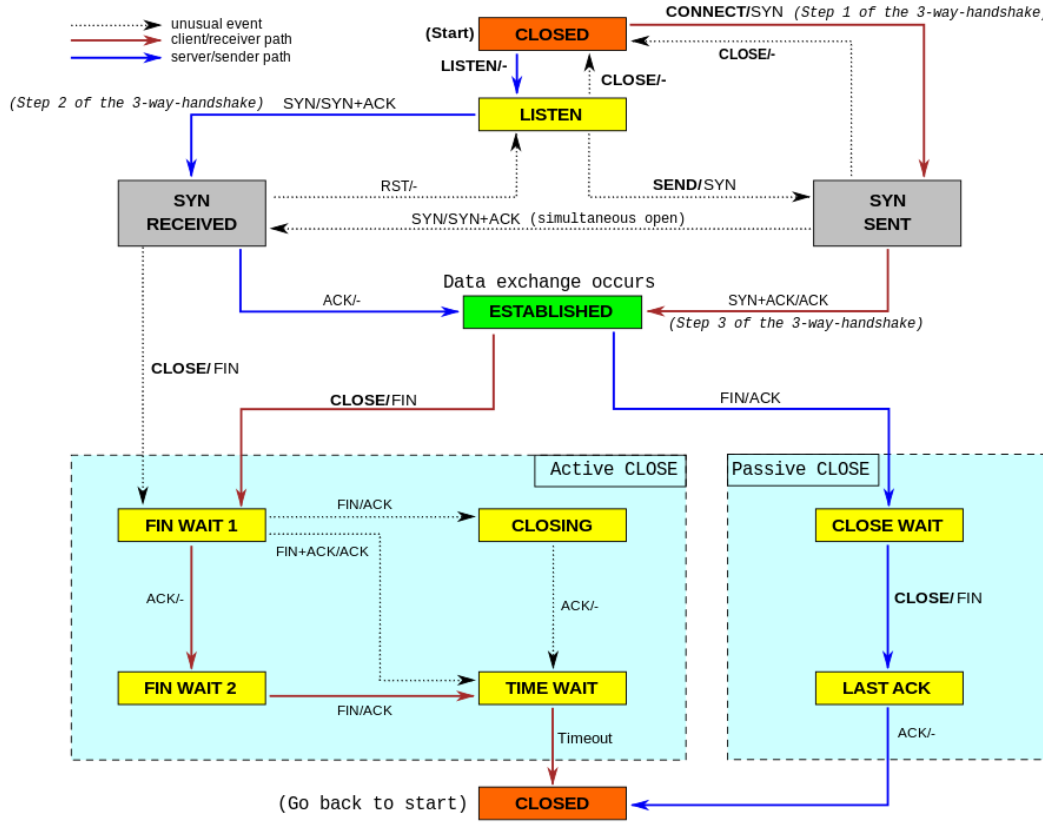


Figure 2.5: Simplified overview of the TCP states [29]. The usual paths of the 3-way-handshakes for server and client are marked. For a more detailed version, see Ref. [30].

back to the server to finish its synchronization. The reply may already include data if the client has something to be sent.

Only a single connection can be established between a given combination of client and server. This means, that a connection is identified by the combination of source and destination addresses of IPv4 and TCP (i.e. source address, source port, destination address, destination port). If an application wants to open more than one connection, it has to instantiate the corresponding number of clients and open a connection for all of them separately.

As Figure 2.6a shows, the establishment of a connection requires at least one reply from the remote node being received before any data can be sent. This delay may not be noticed by the user in normal operation, but it can grow to significant values if the transmission path gets rather long⁴. All timeout counters need to be adjusted according to these delays. It can also produce problems if the transmission path is

⁴There were already some tests made if TCP can be used for communication between space vehicles [31].

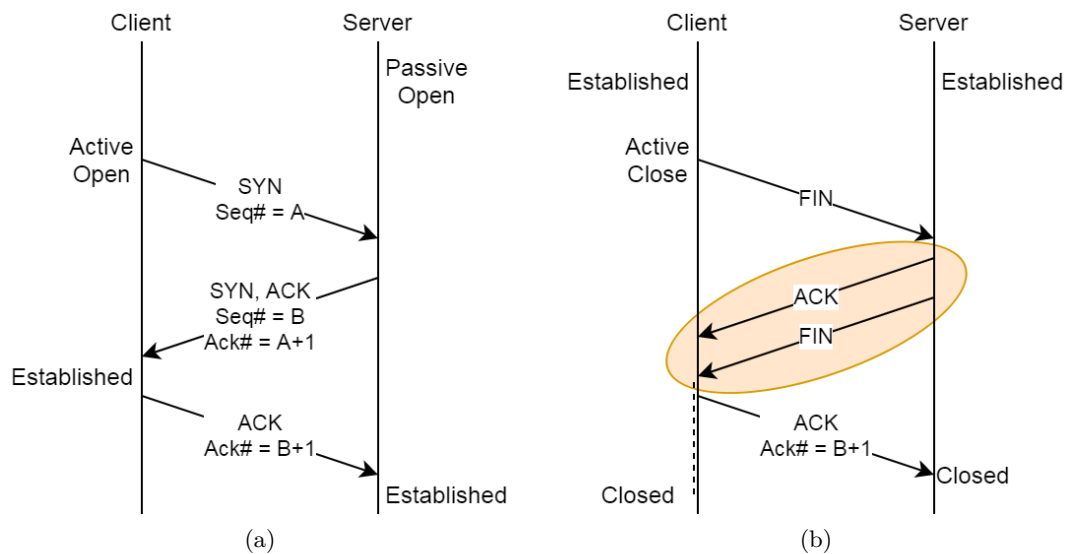


Figure 2.6: a) Establishment of a TCP connection. The client may already send data with the *ACK*.
 b) Termination of a TCP connection started by the client. The *ACK* and *FIN* from the server can also be sent within the same segment.

not stable and the connection needs to be reestablished rather often as this delay will occur each time.

Nearly the same mechanism is used to terminate the connection in a proper way. The two main differences are that the *FIN* flag is used instead of *SYN* and both directions can be terminated independently (i.e. the segment acknowledging the *FIN* flag does not need to carry the *FIN* flag for the reverse part). In the case when only one side terminates its part of the connection (i.e. the connection is still *half-open*), it still needs to read the incoming data and send acknowledgments for this until it receives a segment with the *FIN* flag set. The termination of a connection is shown in Figure 2.6b.

Data (re-)transmission

The payload is sent as a data stream, meaning that each byte has a fixed position in the stream and is delivered in order. This position is given by the sequence number of that byte with the first byte having the initial sequence number (i.e. the one used to establish the connection) plus one. That number is used as identification within the whole protocol. The receiving side is using the sequence number of the next expected byte to acknowledge that all data until this has been received correctly. If a received segment had an error or got lost on its way, the delivery of data to the upper layers is stopped until it has arrived without an error. All segments behind that one (i.e. with a higher sequence number) have to wait for this. If the gap of missing data was

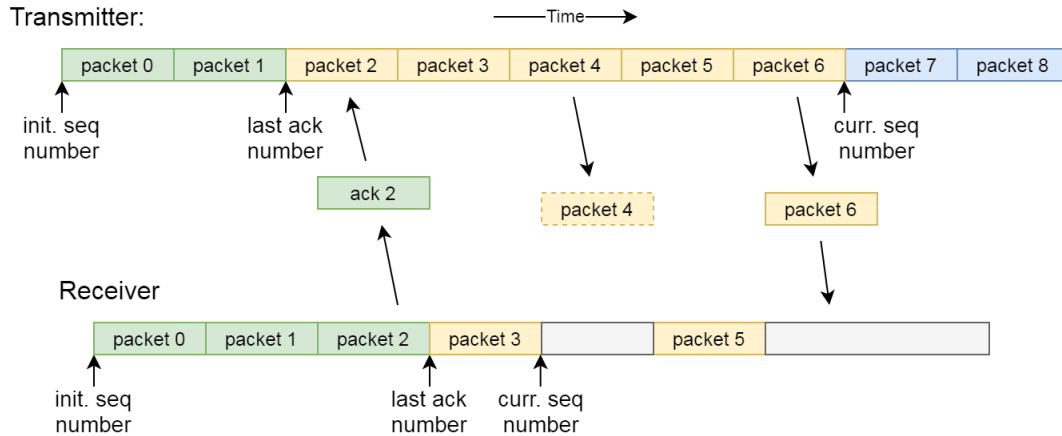


Figure 2.7: An example of how data is transferred using TCP. The transmitter has sent out seven packets and already received the acknowledgments for the first two. The receiver received five packets, with a missing packet in between.

closed, the receiver will again acknowledge all data received without an error until the next missing part by sending the next expected sequence number back.

Figure 2.7 shows an example where nine packets need to be transferred. Seven packets are already sent out with two being acknowledged. The fifth packet got lost in the network and therefore needs to be sent again, leaving a hole in the stream for some time. The receiver got the first four packets and the sixth one. It has recently sent the acknowledgment for the third packet. The sixth and all following packets need to wait for the fifth packets to be received.

There are two ways how an end point can determine if segments need to be retransmitted. The first one is the receipt of duplicate acknowledgments. As the receiver is sending the next expected sequence number, receiving the same acknowledgment number again means it got a segment with a different sequence number. As the underlying layers do not guarantee in order delivery, the segments could just be arriving in swapped order. But getting the same acknowledgment number several times (a threshold of three is used) is a clear indication that something went wrong and the data needs to be sent again. This behavior is called *Fast Retransmit* [32].

The sender may resend only that one segment assuming that it was the only missing one or start to repeat everything from that point. A better solution here is the usage of the selective acknowledgment option. It allows the acknowledgment of out of sequence segments and therefore tells the sender which segments are missing in between. As the options field is limited in size, there is also only a limited number of blocks which can be acknowledged in this way. Such a block is specified by its starting sequence number and the one following right after the block. This option is not mandatory and therefore only used if both sides enable it during the establishment of a new connection.

The acknowledgment number alone is not able to guarantee reliable transmission as the responses can also get lost or the receiving side is not aware of the missing data in case the last segments are dropped. Therefore, there is also a timer started for each segment to trigger a retransmission if that segment was not acknowledged within that period. Some implementations also start to repeat data in case the receiving window (See Figure 2.8) on the other side is closed.

Flow and congestion control

TCP is applying both congestion and flow control mechanisms for not overflowing the network or the receiving side with data it cannot handle. The flow control is mainly implemented by a sliding window called *receive window* which size is defined by the buffer size of the receiver. The buffer on the receiving side has three regions related to the state of the data:

- The first region is the data, that was already acknowledged but not yet delivered to the upper protocol or application. This region will be freed up as soon as the data is delivered.
- The second region contains data, that was already received correctly, but is still not acknowledged. This can be because a segment in between got lost, the segments were received out of order or the receiver wants to acknowledge a bundle of segments and the threshold is not reached yet.
- The third region is the empty one which will be filled up with incoming segments. It defines the amount of data which the receiver can receive before it has to throw data away. The size of this region is sent to the transmitting side in the *window size* field.

The relation between the sequence number and the receiving window (including its regions) is shown in Figure 2.8.

The transmitter calculates from the acknowledgment number and the window size⁵ the maximum sequence number fitting into the receiving buffer. If it reaches this number, it must stop sending additional data and might start retransmission instead. It can start sending new data again as soon as an updated window is reported. A special issue called *silly window syndrome* can occur when the receiver is reporting only a tiny receive window repeatedly as it is processing the data in small chunks. In this case, the transmitter will start sending data and will be stopped shortly after this because the window is closed again.

Another problem occurs when the receiving buffer runs full with all data being acknowledged (i.e. the data is not delivered yet for some reason) as the receiver is not sending acknowledgments anymore (which could update the window status) and the transmitter is unable to send new data which would generate new acknowledgments. To get around that problem, a timer called *persist timer* is started as soon as a

⁵Including the scaling factor, see *Window Scaling* option

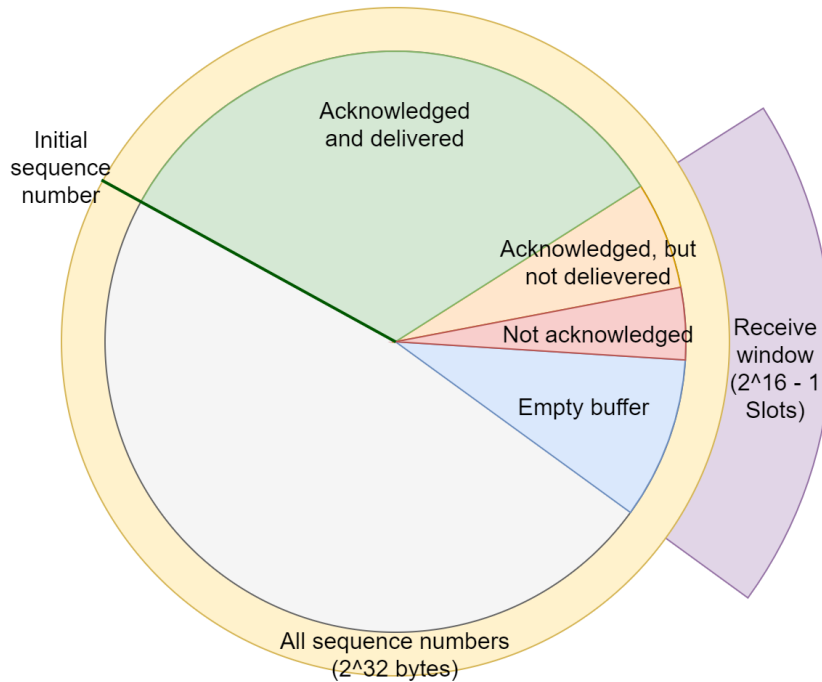


Figure 2.8: Overview of the different states the data can have in TCP. The receive window is holding both undelivered and unacknowledged data.

window size of 0 is reported. The transmitter will send a small amount of data when this timer expires, hoping to get an update with the acknowledgment of this chunk.

The flow control only takes care of the receiving side being unable to cope with the data rate. But the network path between both end points can also be overloaded and thus generate packet loss. Within a managed network with known data rates for each application, this can be avoided by reserving the needed bandwidth for each application. But in an unmanaged network or with unknown bandwidth needs, the maximum reachable data rate needs to be probed. This probing needs to run continuously as the available bandwidth can (and will) change over time. This should be done in a way that does not interfere too much with ongoing traffic. Therefore, TCP also includes a bundle of algorithms called *congestion control*.

As a counterpart to the *receive window*, which is for the receiving side, there is a so called *congestion window* applied at the transmitting side. This window defines the maximum number of bytes being *inflight*, meaning being sent and not acknowledged. To ease calculations, this is done in multiples of the MSS. This window is increased every time a segment is acknowledged and is decreased when a segments got lost and needs to be retransmitted. Different algorithms take care of the size of this window in different situations.

The first algorithm in this area is called *slow start* [32], saying that an end point should not start the transmission with the maximum data rate. It is depicted in

Figure 2.9. The sender will start with a single segment only and wait for the acknowledgment coming back. When this happens, it increases the window by one MSS, sends out the maximum amount of segments (i.e. two) and waits again. With each acknowledgment, the window is increased by an additional MSS resulting in doubling its size after each RTT.

As there is no such thing like an unlimited bandwidth, this exponential growth needs to be stopped at some point. Therefore, a second algorithm called *congestion avoidance* [32] takes over as soon as the *congestion window* reaches a given threshold. It slows the growth down to be linear by decreasing the increment from 1 to 1 divided by the window size⁶. As long as the available bandwidth was not found, the window size needs to grow further. Another reason for this is that the available bandwidth can change in time.

At some point, the network will be overloaded and the packets will be dropped. This is noticed by the sender either by getting duplicated acknowledgments or the retransmit timer running out. In the later case, the sender assumes that all segments got lost and therefore that the available bandwidth has dropped drastically. The threshold between *slow start* and *congestion avoidance* is set to half of the current window size, the new window size is reduced to the initial value of a single MSS and transmission starts again in the *slow start* mode. In case of the duplicate acknowledgments, at least some segments have arrived at the receiving side, so there is at

⁶In multiples of the MSS

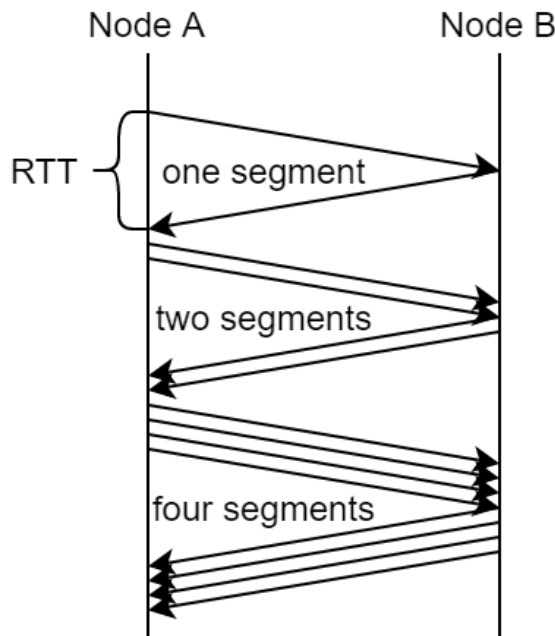


Figure 2.9: Depiction of the slow start algorithm used TCP. The number of segments being inflight is doubling each Round-Trip Time (RTT).

least some bandwidth available for this connection. So, the hard cut from the *slow start* mode is not needed and it is skipped by setting both the new threshold as well as new window size to half of the old value. This is called *fast recovery* [32].

The behavior of increasing the size with each acknowledgment and halving it as soon as a packet got lost is referred to as *Additive Increase/Multiplicative Decrease (AIMD)*.

2.3 Alternative Protocols

There are some cases where the properties of TCP are not fitting the problem to be solved. This is not only due to the complexity which is coming with all the algorithms, there are also cases where a different tradeoff is required. For example, it is in some cases better to get the most current value (data) than getting all values, so sending the new value is better than a retransmission of an old one.

In this section, there are three different solutions presented for different aspects where TCP is not the optimal solution.

2.3.1 The User Datagram Protocol (UDP)

Instead of establishing a connection to only send a small amount of data (e.g. three packets or less) it might be better to send the data directly as TCP would need three network packets to establish the connection and needs to close the connection afterwards again. The User Datagram Protocol (UDP) was developed to have a lightweight protocol for such cases. It is a connectionless protocol without any mechanism for reliable data transmission. This also includes an unordered data delivery and duplicate delivery in case the underlying protocol has duplicated the packet. Like TCP, also UDP assumes that the underlying protocol is IPv4. Figure 2.10 shows the UDP header as defined in Ref. [33].

All fields in the UDP are 16 bit wide and are described next:

The Source port is used to specify the application on the sending node.

The Destination port is used to specify the application on the receiving node.

The Length field is the length over the whole datagram. With the header being 8 byte long, this field must be greater than 8 to hold any payload data.

Bit Offset	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	Source port																Destination port															
32	Length																Checksum															

Figure 2.10: Structure of the UDP header as defined in Ref. [33].

Bit Offset	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	Source address																															
32	Destination address																															
64	Zeros								Protocol								UDP length															
96	Source port																Destination port															
128	Length																Checksum															
...	Data																															
...																																

Figure 2.11: Depicted is the pseudo header used for the UDP checksum calculation. The same IPv4 header fields as for the TCP checksum are used here too. These are marked in purple. Like in TCP, the checksum field is set to zero for the calculation.

The Checksum is used to validate the correctness of the header and data as well as some additional fields from the IPv4 header. The resulting pseudo header is shown in Figure 2.11.

The differences towards TCP

The most important difference with respect to TCP is the lack of reliability. But this is not necessarily a disadvantage as in some situations only the latest value is of interest, so sending an old value again has no benefit. And even if a history of the values should be kept for later analysis, if the update rate is high enough the loss of some intermediate values might be tolerable. For example, when buying stocks, getting the current price is more desirable than getting an old price which got lost before.

Without reliable data transmission, there is also no guarantee that all older packets are arriving. So, there is also no sense in ordered delivery as this would mean, that a newer message is held back until all old values are delivered. But it requires also a way to detect if an older value was received and should be discarded if an old value could be harmful. Even if a protocol on top of UDP implements some kind of reliability, the unordered delivery can be beneficial in case of independent data sets being transferred as the processing of a later set does not need to wait until all earlier sets have arrived.

Being connectionless also means that UDP is stateless, so there is no need in updating any state variables, thereby reducing the overhead. It is also more error

tolerant as it cannot be in a wrong state. Also, in contrast to TCP there are no state changing packets in UDP, so it is harder to completely suppress data transmission.

Another benefit of being connectionless is the possibility of multi- and broadcast messages as the message can be duplicated and sent to different recipients easily. With a connection-based protocol, the sender would have to track the state of each recipient and find a way fitting all of them. So it is easier to have a separate connection for each recipient.

Without connections and reliable data transmission, there is also no need to send data back to the sender. So the sender does not get any information back if the packets have arrived at their destination or if the receiver has some free resources for packet processing. Without these information, there is no way to implement a useful flow or congestion control. A result of this is that UDP should not be used to achieve the maximum bandwidth available. It should rather be used where only a low bandwidth is required.

Typical applications for UDP are applications where the reliable element is already implemented by other means. This can be in case of other layers already ensuring reliability like in tunneling applications (e.g. Virtual Private Network (VPN)). When a reliable protocol like TCP is used within such a tunnel, the tunnel itself does not need to be reliable. If TCP would also be used for the tunnel itself, a lost packet would generate a retransmission on the tunneling level. But the encapsulated TCP connection might also notice the loss as the end point is getting no data until the tunneling connection has received the retransmission and tunneled packets are delivered. This would lead to a retransmission by the encapsulated TCP connection resulting in two retransmissions being triggered for a single packet being dropped. Therefore, such tunneling applications prefer UDP to avoid these duplicate retransmissions. But also applications like Voice over IP (VoIP) prefer UDP where the reliability is realized in the human communication rather than the technical solution. The people talking with each other can handle a short interruption better than a variable delay caused by retransmissions.

Another field of applications using UDP are broad- and multicast applications, where the same information is sent to many destinations at the same time and the information included in later packets supersede the previously received data. These use cases can be compared to terrestrial broadcast services or satellite TV. But this also includes online computer games as the overall status is sent repeatedly, potentially to a lot of destinations. As long as the interruption is not too long, packet loss is not such a great deal and the next arriving packet will have the updated situation.

2.3.2 The QUIC protocol

Initially developed by *Google* as *Quick UDP Internet Connections*, it is not used as acronym anymore. The goal of this protocol is to replace both TCP and plain UDP as transport protocols for binary data (especially Hypertext Transfer Protocol (HTTP) streams) in the internet. It uses UDP as underlying protocol as QUIC itself

is not supported by current operation systems.

Currently, HTTP is running mostly on top of TCP, so a TCP connection needs to be established before any request for a website could be sent. In case of an encrypted connection is wanted, there is also a negotiation step with multiple handshakes before the HTTP request can be sent out. This is causing rather large delays. An attempt to mitigate this was to use a single TCP connection for all objects to be transmitted hoping that the delay is only occurring once. But this is only true if all resources are available on the same server. As modern websites tend to use external scripts for authentication or usage analysis, this is not always possible. QUIC is getting around this as it already includes encryption, so establishing the connection and negotiating the encryption can be done in the same step. It further optimizes this process to reach a 0-RTT handshake with which a data transfer can start right with the second packet. This way, requesting only a single, rather small object from a server can be done with nearly no delay.

Another problem of using only a single TCP connection is the stopped data delivery in case of a packet being lost. This is stopping the delivery of all objects to be transferred, not only the one being currently transferred. QUIC is solving this by separating *data streams* from *connections*. A single connection can handle several streams. In addition to this, it also allows other kind of payload to be sent on a connection. This is necessary as all signaling between both endpoints should be done within the encrypted connection instead of the unencrypted header as TCP is doing. The data to be sent is packed into *frames* of different types to distinguish between data streams and other signals. If a QUIC packet of a type which contains frames⁷ is sent, it must contain at least a single frame, but can both have several frames as well as several types of frames. Furthermore, a packet always contains whole frames i.e. a frame is not allowed to span over two or more packets. In case a packet gets dropped, only the specific frames included in this packet are missing and all other transfers are not affected. Therefore, the number of different streams included within a single packet should be as small as possible to reduce the number of streams being affected if the packet gets dropped. Figure 2.12 shows an example how different streams could be distributed across packets.

Another reason for limiting the number of concurrent streams is that each stream needs some buffering space at the receiving side as the receiver won't have unlimited memory. QUIC thereby supports limits for single streams as well as for a whole connection. In case of such a limit being reached, the sender can notify the receiver that it has more data to be sent but reached the limit and is therefore asking to increase the limit. The receiver can advertise a new (i.e. higher as smaller ones are ignored) limit by sending special frames containing a new absolute offset (in contrast to TCP, which uses the size of free memory left in the buffer). This has the benefit that the protocol processing is not disturbed by multiple updates or if these are received out of order. If the sender is ignoring a limit and keeps sending (like TCP would do to get an update of the receive window) the receiver must close the

⁷Not all of them do.

connection with a flow control error, regardless if a stream limit or the connection limit was violated. This should help the receiver in cases of a malicious sender.

QUIC also introduces several innovations to stabilize the connection between client and server. The most important one might be the change how a connection is identified. While TCP is doing this by the combination of port number and IPv4 address of both server and client, QUIC is introducing an independent connection identifier. This has the benefit that the connection is not lost in case of one of these numbers is changing. As many devices (especially mobile ones) have more than a single network connection nowadays, switching the network interface will also change the IPv4 address seen by the other endpoint and therefore reset the connection. Because of the separate connection identifier is not changing in such a scenario, QUIC is not losing the connection.

Google was also searching a way to reduce packet loss as much as possible. One way is the decoupling of finding the optimal bandwidth to be used from the number of packets being dropped.

Another approach was the introduction of a mechanism to reconstruct lost packets. This can be done by adding redundant information to the data stream and distributing this in the data stream so that a lost packet can be rebuild from the other packets. The problem with this approach is that the amount of additional information and the way it is inserted in the data stream depends on the expected error model. Such a mechanism is called *forward error correction (FEC)*. Sending each packet two or three times would be rather easy, but also results in an increased bandwidth accordingly. Building the parity across ten packets and sending this as an additional packet would only increase the needed bandwidth by 10 %, but only a single packet out of these eleven is allowed to get dropped before data needs to be

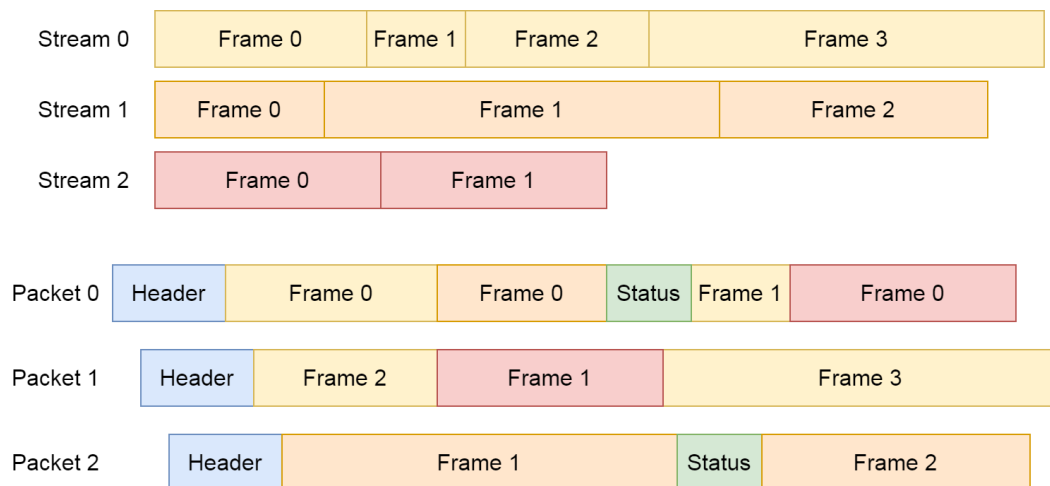


Figure 2.12: Depicted is a possible distribution of three streams on three packets. Arrangements like packet 0 are highly discouraged and should be avoided as long as it is not needed for performance reasons.

retransmitted. So if the expected error model is changing over time, it is hard or even impossible to adapt to this on the fly. Also, if the rate of dropped packets is rather small, a lot of unneeded data is transferred just to rebuild a few missing packets. In the end, *Google* stopped these experiments and removed this feature again.

2.3.3 InfiniBand's RDMA over Converged Ethernet

The InfiniBand Architecture (IBA) specifies a mechanism to connect a processor node with an I/O device or another processor node via a so-called *fabric*. It defines how data and commands can be exchanged between these nodes but not how these commands or the data format should look like. The IBA hardware off-loads most parts of the protocol processing and is able to access the local memory without involving the CPU, thus reducing the CPU overhead of other communication protocols. This also reduces latency as the CPU does not need to stop its current work to copy the data. The IBA is offering a transparent transport of memory content and supports high bandwidth and scalability. It is optimized for nodes with multiple consumers and therefore supports multiple channel adapters per node as well. In the IBA, a transfer is started by creating a *work request* and submitting it to the IBA hardware via the work queues (i.e. send and receive queue). The hardware will execute the transfer and report back with the result using a completion queue.

There are two different kinds of transfers. In case of the first one, the sender specifies the memory area to be transferred and the receiving node, but not the memory location at the receiving side. These are specified by the receiver through special receive instructions. So both endpoints are actively involved in the transfer. The second kind are Remote Direct Memory Access (RDMA) operations where a consumer node is also specifying the remote memory location. In this scenario, the remote node does not need to act actively.

The nodes within the fabric are addressed by none-permanent local identifiers as well as permanent globally unique identifiers given by the vendor. These can be compared to the IPv4 addresses which also may change and the Media Access Control (MAC) addresses of the ethernet protocol. This similarity suggests the replacement of the special IBA fabric with an ethernet based network. With such a replacement, the RDMA transfers can use most likely already existing network infrastructure, which enables the parallel usage of other ethernet based protocols for control and configuration.

The IBA is as flexible as the whole IPv4 protocol stack as it supports both connection-oriented and connectionless communication. But in contrast to the IPv4 protocol stack, the IBA offers the option to enable acknowledged (i.e. reliable) transfers for both. Also, these options are available within the same protocol and can be enabled for individual sets of transfers.

2.4 Complexity of Retransmission

Independent of the protocol being used, reliable communication depends on the ability to repeat parts of the transmission. Even the most efficient FEC algorithms are limited to what set of errors they can correct. There are some scenarios (like broken cables or fibers) where no error correction mechanism can reconstruct the lost packets. In these cases, the only way to get the data to the recipient is to send it again when the problem is solved. But this requires that the data is still available, i.e. all packets sent out also need to be stored in some kind of memory on the transmitter side.

This introduces two constraints which might get problematic. The first and obvious one is the required memory bandwidth to save the data which should be bigger as the network bandwidth for not being the bottle neck. If the data also needs to be read from the memory (e.g. the memory is used to buffer the data stream), the needed memory bandwidth is doubled. In such a scenario, a *DDR4-3200* module with a (theoretical) memory bandwidth of 204.8 Gb/s would barely⁸ serve a 100 Gb/s network interface. Embedded systems for high performance computing (HPC) can have several of such network interfaces which would require a separate *DDR4* memory channel for each of them.

The other constraint is a more subtle one. The data needs to be stored until the recipient signals the correct transmission. This means that there needs to be a communication channel back from the recipient to the transmitter. This sounds like a trivial constraint but there are many popular systems (like TV or radio broadcast services via satellite) without such a response channel. But even if it exists, it takes some time until the response reaches the transmitter, the so-called Round-Trip Time (RTT). As the response channel is most likely also affected by data losses, the acknowledgment might also get lost and needs to be sent again. So, it might take several RTTs until the acknowledgment arrives.

All data sent during this time is still unacknowledged and needs to be kept in the local memory. The amount of memory needed for this can be calculated by the network bandwidth multiplied by the RTT and a safety factor. This can reach values of the order of GB for systems with a combined network bandwidth of several 100 Gb/s. Even if the memory bandwidth and the amount of memory are solvable problems, they increase the space needed, the costs of the overall system and add additional complexity which would not be necessary otherwise.

Besides the hardware related issues, there are also limits at a conceptual level. How to deal with retransmission in broad- or multicast systems where different recipients might miss different parts of the transmission? Are the lost parts sent to all recipients again or only to the ones from which no acknowledgment was received? If the later approach is used, how long should the transmitter wait for acknowledgments?

⁸The memory interface is not as efficient as the network one due to various latencies. It also needs some time to switch between write and read access.

Reason for retransmissions

All this complexity is needed to guarantee the correct delivery in all cases. To reduce the effort needed, it is worth to investigate the reasons for data loss as a cause of retransmissions. These can be sorted into three groups:

- bandwidth probing
- interruption of the communication system
- components being unable to cope with the data rates

In the context of unknown or changing communication conditions, packet loss is a strong indicator that a limit was reached, especially if packets get dropped continuously. As there is no indication for how much of the available bandwidth is used, loosing packets is the only usable one. For this reason, the flow control algorithm of TCP is based on the packet loss rate to probe the available bandwidth as described in Section 2.2.

The only way out of this would be a fully managed network infrastructure where all transfers would have a known, reserved bandwidth available. So each transfer would need to request an empty slot on a predefined route. Furthermore, this would require a central management system for planing and arbitration of the available bandwidth. This could also be organized independently for different sub-networks, but would lead to several of such requests being needed if a transfer is crossing the borders of these sub-networks. So, a managed network is only feasible for rather small networks.

The second group of reasons for packet loss are unplanned interruptions of the network hardware. Examples for this are cables or fibers being accidentally cut by an excavator at a construction site or a router having its power cut-off. All data sent to such a network link will be lost as there is no receiver at the other end for further processing. In a redundant network, the affected links will be shut down eventually and the traffic will be routed around. Retransmissions can recover a transmission by using the redundancy of the network. As the unaffected links will need to take over the traffic, the available bandwidth for each transfer will most probably go down for each one resulting in packets getting dropped.

In case of a network or specific link without redundancy being affected, the data will be completely lost if no local copy is kept. So, reliable data transmission is only working as long as the network has enough redundancy to bypass a broken link. If the redundancy is not given (e.g. the link between the node and the network is affected), even a reliable protocol is not able to transfer the data.

The third group of reasons for packet loss is composed of network components unable to cope with the data rate. This occurs not only at endpoints but can also occur in other components like firewalls. Network links with less bandwidth than needed and not being redundant are included in this group. It mostly happens if the number of packets to be processed in a time interval is reaching the limit. But this

can also happen when the node has other processes running and needs to interleave the tasks. If the node is not switching back to the network processing fast enough, the local buffers can run full and data is getting lost.

So this group is the result of nodes not being configured correctly for the actual tasks. This misconfiguration can be related to both hardware and software. It should be noted, that the node being unable to cope with the actual work to be done might come from an unusual high workload being present. So even if the node is configured correctly for the planned load, it might get into situations where the actual load is exceeding the planned one. In addition to that, finding a working configuration is much more difficult than just planning network capabilities as it highly depends of the kind of traffic.

Retransmissions in small, separate networks

While the first and second group are unavoidable in large area networks like the internet, these can be avoided in small, separated networks. The first group can be dealt with rather easily by the introduction of guaranteed bandwidth for each application (i.e. Quality of Service (QoS)) so that there is no need to probe the bandwidth. Also events related to the second group should occur less often as the network infrastructure can be placed in a better supervised location where it is less likely to be harmed accidentally. Even if such a case is occurring, it would most properly result in an overall interruption of the affected application, so packet loss won't be the issue to worry about.

So all the reasons of packet loss related to the network itself can be taken care of or will be concealed by more urgent issues. But the problems related to the third group are still there. With the network itself being managed much better than it is possible for global ones, the load seen by each network node can be defined and controlled much easier. Therefore, it should be also easier to configure the nodes accordingly.

Assuming an appropriate configuration of all network nodes, it seems that losing a packet could be avoided completely. This consideration gives rise to the question if reliable data transmission is needed at all in such an environment.

2.5 Development of the DROP protocol

As seen in Section 2.3, neither the TCP protocol nor its alternatives completely fulfill the requirements for a lightweight data streaming protocol. The usage of all named protocols except UDP would result in additional complexity as they need some kind of fast memory to guarantee reliability. Only the UDP protocol comes without this feature, but lacks some other important features. Section 2.4 showed that reliability may not be needed, so the solution within this thesis is to go with UDP and add all missing features in a custom protocol on top of it.

Mapping of features to existing protocols

While the connectionless nature of UDP is rather helpful than a problem, the data has still the form of coherent streams with an internal ordering. This means, that the data streams should not be mixed. Furthermore, mechanisms for splitting the streams into portions of data and for sorting these are needed.

To distinguish between the different data streams, some kind of identification is needed. But instead of using an arbitrary number, the method of identification should provide additional information about the source and destination of the stream. Thereby, an identifier for the hardware platform as well as the data source or data sink on this platform should be included, resulting in four identifiers for each stream.

This matches the definition of a stream in the TCP protocol, where the IPv4 address is used to identify the hardware instance⁹ and the TCP port numbers for the entity on the hardware. Therefore, this model is carried over to UDP. With the destination of the data streams being a software running on a server, it is the same as for TCP. The software instance processing the data is given by the destination port of UDP and the IPv4 address of the server it is running on. In the same way, the source of the data stream is represented by the UDP source port for the FE chip the data comes from and the IPv4 address of the instance of the FPGA system.

The splitting of the streams is obviously done by forming UDP packets with the size of a data block corresponding to the UDP length. When adding a custom protocol on top of UDP, the space needed for the protocol header is added to the UDP length and needs to be subtracted again to get the length of the data block.

⁹This might also be virtual hardware in case of virtual servers.

Bit Offset	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	Version			Header Length			Type of Service								Total Length																	
32	Identification																Flags		Fragment Offset													
64	Time To Live								Protocol								Header Checksum															
96	Source address																															
128	Destination address																															
160	Source port																Destination port															
192	Length																Checksum															
224	Flags								Packet identifier																							

Figure 2.13: Shown are the fields of IPv4(gray) and UDP(green) used as well as additional fields added by Data ReadOut Protocol (DROP)(yellow).

Another important feature to be added to UDP is the identification of individual packets. Such an ordering of the packets could be achieved by making use of the *identification* field of IPv4. But making use of this field would require a way to access this field from outside the network stack.

In addition, setting it from the user level may violate the requirements of IPv4 for this field as two different data streams could be interleaved and using the same identifier in consecutive packets. Moreover, there are other protocols like Internet Control Message Protocol (ICMP) running on the platform which also need a valid value for this field.

Therefore, such a mapping would require to split up the value range in segments for each application as the data streams are independent of each other and the order in which their packets are sent is not controlled by a supervisor. But the *identification* field is already rather small with a size of 16 b. With packet rates on the order of millions of packets per second, these segments will overflow within milliseconds, what is also the order of the time a packet needs to cross the network. So, this solution is not suited for the problem.

Figure 2.13 highlights the fields used for the mapping together with the additional field added by the custom protocol.

The DROP protocol as a custom extension

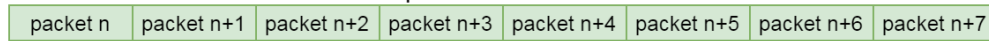
As not all needed features could be mapped to counterparts in the existing protocols, a new protocol is needed. Therefore, the DROP is introduced. The main task of it is the identification of packets. Next to this, it should have some further properties and options.

To design the protocol in a CPU friendly way, it will be built of 32 b words. It should also have a field at the start of the header for some flags which could be used to signal the presence of optional fields. With the identification of packets being the required feature that could not be mapped to existing protocols, it will also be the only mandatory field next to the flags field. Therefore it fills the remaining space of the first 32 b word.

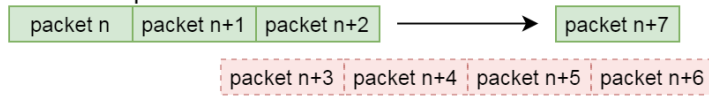
To provide a larger range as the identification field of IPv4 (i.e. 16 b), the identification field within DROP is set to 24 b, leaving 8 b for the flags. This should ensure a long enough period until the same identifier is used again, especially as this field is for a single stream. The identifier should be implemented as a packet counter being increased by one with each packet. This is important for the following reasons:

- If two consecutive packets are received with the identifier of the second packet being increased by one compared to the one of the first packet, it is assumed that no packet was lost in between. It is very unlikely that exactly (a multiple of) 2^{24} packets got lost.
- In the same case as above, it is assumed that no additional packets were inserted into the data stream as it would also need exactly (a multiple of) 2^{24} packets.

Case 1: neither lost nor additional packets



Case 2: lost packets



Case 3: additional packets

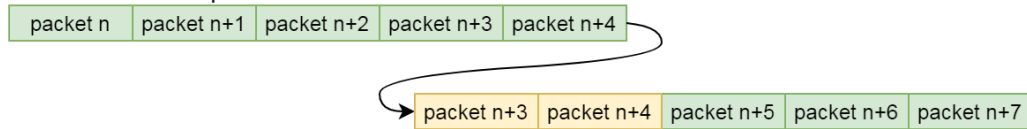


Figure 2.14: Depicted are examples how lost or additional packets are recognized.

- If the identifier of two consecutive packets have a difference larger than one, it is assumed that one or more packets got lost rather than additional packages were inserted. The exact number of lost packets is the difference decreased by one.
- If two consecutive packets are received with the identifier of the second packet being smaller than the one of the first packet increased by one, it is assumed that additional packages were inserted rather than packets got lost.

Figure 2.14 gives examples for the four cases.

With this definition, it is possible to count how many packets are lost both per gap (referred to as a batch of missing packets) and in total. This helps to characterize the network transmission as it can be used to measure the quality of the chosen parameter settings. But this also defines how pattern of typical error scenarios would look like. A flipped bit in the identification field for example would be recognized as a combination of missing and additional packets. The same number of packets would appear as being lost and being added. The order of lost and added packets depends on the direction of the flipping.

As the given definition of the identification field fulfills the main requirements, the header with that single 32 b word and with no flag used is defined as the *default header* configuration. This means, that all other functions have to use a flag (or go into an option if more space is needed).

Optional fields in DROP

Before going through possible options, there are two major problems with options that should be solved first.

The first problem with optional fields is that each one needs a flag to signal its presence and there is only a very limited number of these available (i.e. eight in the default header). If more flags (e.g. for more optional fields) are needed, an option

providing more flags could be added. Such an option must use one of the eight initial flags and should therefore be implemented as one of the first options to not end up in a situation, where a flag needs to be redefined resulting in different versions of the protocol having different meanings for the same flag,

The second problem is the case that one side might not know the meaning of a used flag and therefore also does not know if it signals the presence of additional header fields. If there are such fields, the length of these and therefore the start of the data area is unknown. This gets even worse if multiple flags are used at the same time, especially if only parts of the used options are known and must be found.

A simple but also very strict solution would be to restrict the usage of options to a single option being used within a packet and no data being allowed as soon as an option is used. If a receiver sees a packet with an unknown option being set, it can simply ignore the packet. In this way, the total length of the packet is used by the header and the option.

This solution would forbid the usage of additional flags to signal further optional fields as this would imply at least two options to be used within a single packet. Defining the option for more optional fields as mandatory and therefore being an exception for the *single option* restriction is regarded as not sensible as the needed number of addition flags is completely unknown. This option might never be needed and making it mandatory would only increase the likelihood of needing it, resulting in a self-fulfilling prophecy.

The problem of unknown length could also be solved by defining a standard length for each option. But this would require to implement the options as a list rather than using a flag field and only the presence of the list being signaled by a single flag. Each entry in this list would have the same predefined length, including an option type field stating the format of the current field. The end of the list would either be signaled by a special option type or by a special flag signaling the presence of further optional fields. Both ways would generate additional overhead with the impact of it depending on the size and number of such fields. If a specific option type is unknown to an implementation, it can simply skip that field and check the next one if existing. The benefit of this solution would be that it would count only as a single option for the initial header format and therefore fulfills also the requirements for the former solution. The downsides are that two different ways of handling options would be used and that options needing less space than the defined entry size result in (potential large amounts of) unused data blocks to be transferred.

A third version to get around this problem would be to build the list of used options out of some kind of pointers holding the position at which the corresponding field starts plus an entry for the start of the data field. The type of each option could either be included in the list itself or at the start of each option. In the former case, the data field would be handled as a special option type also signaling the end of the list. In the latter case, the length of the list would either be given before the first entry in the list or marked by a special pointer value (e.g. zero). Figure 2.15 on the next page shows a comparison of the three variants.

It should be noted, that all these organizational options bring additional overhead

More options are made available by...

a) an option providing more flags:

Bit Offset	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	Flags							M F	Packet identifier																							
32	Additional Flags																															

b) a list of equally sized entries:

Bit Offset	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	Flags							M O	Packet identifier																							
32	Type							M O	Option Data																							
64	Type							M O	Option Data																							

c) a list pointing to the option fields:

Bit Offset	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	Flags							M O	Packet identifier																							
32	Type							Offset							Type							Offset										
64	Data							Offset							Type							Offset										
96	Option Data 1																															
128	Option Data 2																															
160																																

Figure 2.15: The different ways to enable additional options. These are: a) an additional field for further flags b) a list of holding the option data directly c) a list of pointers to the start of each option

and restrictions with them. Therefore, as long as no (or not more than a handful) options are defined and regularly used, it does not make any sense to implement one of them.

The first functional option to mention here is obviously for implementing a (partial) reliability in data transmission. The receiver of a data stream would maintain a list of missing packets and sent it to the transmitter periodically, requesting the retransmission of these packets. It is assumed that the packet loss occurs in batches, so the list is built of the packet identifier of the first packet of a gap and the number

Bit Offset	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	Flags						R T	Packet identifier																								
32	Gap length							first missing packet																								
64	Gap length							first missing packet																								
96	Gap length							first missing packet																								
128	(Zero)							last seen packet																								

Figure 2.16: DROP option to report missing packets. The packets are reported as a list of batches. The last entry is the last seen packet identifier allowing to check for missing packets at the end of a data transmission.

of packets being lost. Figure 2.16 shows how this could look like. Since the requests include the identifiers for all missing packets, the loss of a fraction of these requests (the path back is also not reliable) is not problematic as the next request will supersede the former one anyhow. The receiver should nevertheless use a high enough frequency to ensure proper operation even when a few consecutive requests get lost. This does not only include the amount of memory needed to store the packets on the sender side, the retransmission requests also need to arrive before the stream identifier overflows and the requested identifiers refer to a different packet.

The transmitter can use the requests to free up its memory since all not missing packets are regarded as successfully delivered. It could be beneficial to also include the last seen packet identifier in this request in case no packet is reported as missing. Without this information, the transmitter would not be able to clean up its memory as it does not know how old the message is. Including the last seen packet identifier also ensures to detect lost packets at the end of a stream. As the receiver does not know how many packets should be sent, it would not be able to mark these as missing and request their retransmission. It is also important to note, that the requests have to be sent even if no packet was lost (i.e. empty requests).

With this option being enabled on a per stream basis, a mixed operation with reliable streams for slower management streams and best effort streams for the bulk data could be implemented using the same protocol for both. This saves resources by not needing a different protocol to be implemented. If the data rate of the reliable stream is low enough, it might even use local memory within the FPGA instead of needing external one.

As the goal is avoiding retransmissions, there have to be other ways than using the option for reliable transmission. It is better to give the possibility to rebuilt missing packets to the receiver. Such a mechanism is referred to as *FEC*. The transmitter splits the data stream into blocks of defined size (measured in number of packets) and inserts additional information to each of them. This requires that all packets

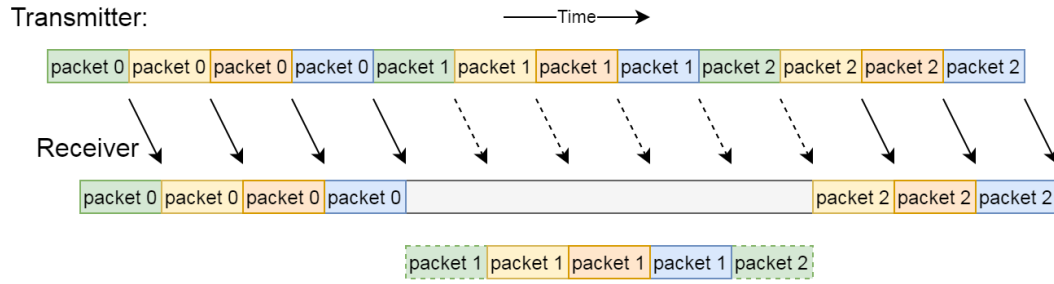


Figure 2.17: Example of how a FEC mechanism could be used to reconstruct missing packets. The data is split into interleaved blocks (depicted by the colors). The packets of the green block cannot be reconstructed as the symbolized mechanism allows only for a single lost packet within each block.

have the same length, so a mechanism to fill up packets is needed in case there is not enough data to be sent. Otherwise, the receiver would most probably fail to stay within its latency boundaries if it needs to reconstruct a packet as the arrival of enough data in that period cannot be guaranteed. But adding padding¹⁰ to the packets is also problematic as the packet size and the amount of data in it is not the same anymore. Therefore, the padding itself should be omitted and only used for the checksum calculation on both sides, which also saves network bandwidth.

The simplest implementation (besides just sending each packet multiple times) would be to calculate the parity across the same bits in all packets and send it as an additional packet. This would allow for a single packet within a block to be rebuild. As it is assumed that packets are lost in batches, the packets of several blocks should be interleaved to enable the reconstruction of consecutive packets. This could also be used to distribute the workload of checksum calculation to independent pipelines.

Figure 2.17 shows how this could look like with four blocks (indicated by the colors) being interleaved. In this example, five consecutive packets are lost and therefore only the three packets in the middle can be reconstructed. The two packets of the green block cannot be reconstructed when the parity method is used. More complex algorithms allow for more packets of a block to be rebuild, but these would also require much more computing power.

The extensions described in this section are for now only seen as possible candidates which could be implemented at some point. The decision about their implementation will depend on the performance of the protocol being seen in the tests done.

¹⁰To ensure the correct alignment of elements with a data structure or a minimal length of such a structure, some additional data elements can be inserted. These parts do normally not hold any useful information. These additional data elements are called *padding*.

Chapter 3

Detector readout

The readout of event data from the front-end (FE) chips is organized in a processing chain with different processing steps at different locations. This is done in order to group these steps by functionality and responsibility. It needs well defined interfaces to get such a distributed system working.

This chapter starts with a more general description of what a FE chip is, taking the first prototype of the new chip for the ITk Pixel detector as an example. Subsequently, an overview of the readout chain is given and the components and interfaces involved are described. The resulting constraints for the first data processing part outside the detector are described in more detail as this part of the readout system also takes care of the translation between proprietary and commercial hardware.

3.1 The front-end (FE) chip

The FE chip together with the sensor and a flex PCB form the modules which are the active components of a detector. Each FE chip has several thousands of readout channels (i.e. pixels in case of a pixel detector) with each channel being connected to a readout cell in the sensor. Figure 3.1 shows a schematic drawing of such a hybrid module.

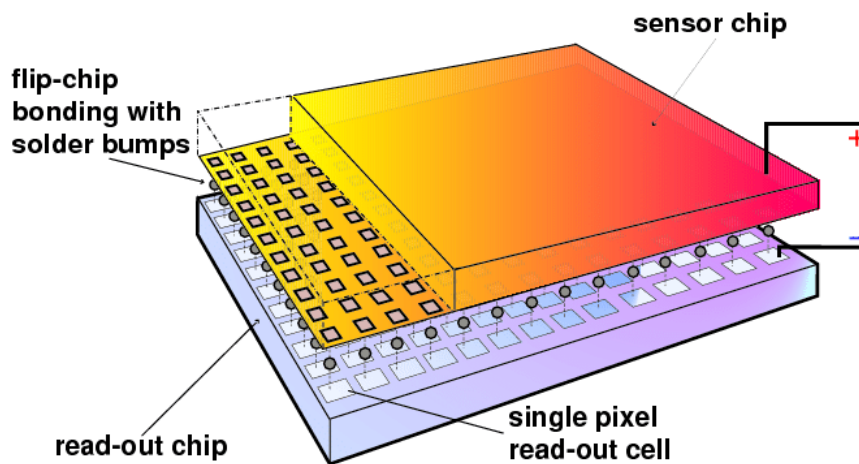


Figure 3.1: Drawing of a silicon pixel sensor with bump-bonded FE chip [34].

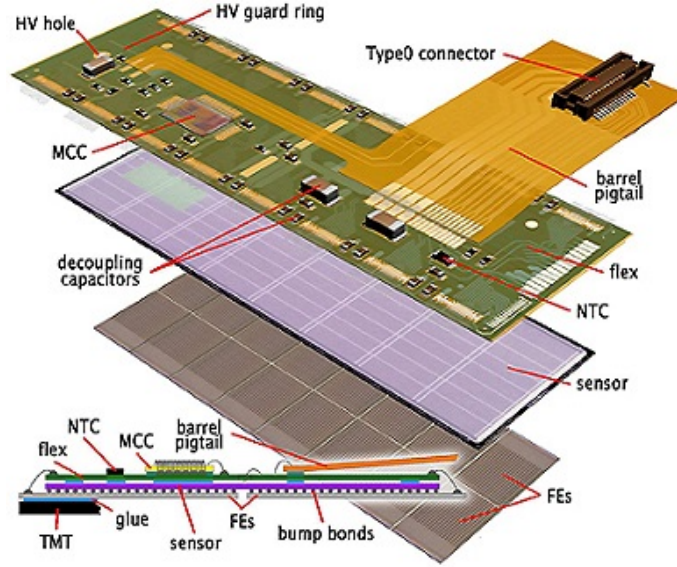


Figure 3.2: Drawing of a module of the current Pixel detector and how it is composed.

There are also research activities to build monolithic modules, where the active FE electronic and the sensor cells are created in the same substrate block. But these are not further covered in this thesis as none of them is used in ATLAS ITk.

A module can contain one or more FE chips while most modules have only a single sensor element. The modules of the current ATLAS Pixel detector for example has 16 FE chips connected to a single sensor. This is depicted in Figure 3.2.

In case of silicon detectors, the sensor forms a diode which gets depleted by a reverse voltage. This voltage can range from as low as under 10 V to values above 1000 V, depending on the thickness and other properties of the sensor. When a charged particle goes through the sensor, it generates free charges and therefore a current. The signal is then fed into the FE chip, amplified, formed and digitized. Figure 3.3 shows a schematic drawing of the processing steps involved.

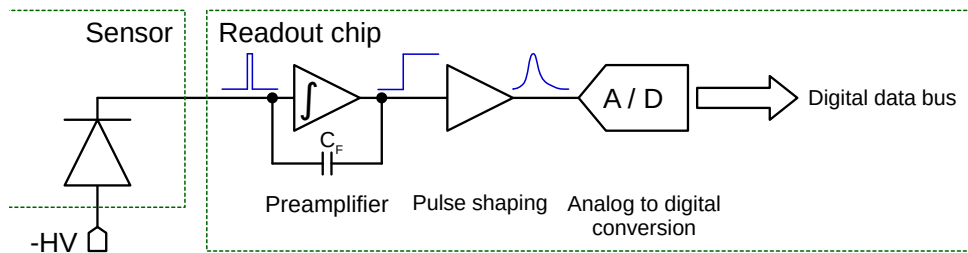


Figure 3.3: Basic blocks of an analog front end electronics for the readout of a silicon sensor. C_F defines the final gain of the amplifier stage. Such an electronic circuit is implemented for each channel of the sensor. Based on Ref. [35].

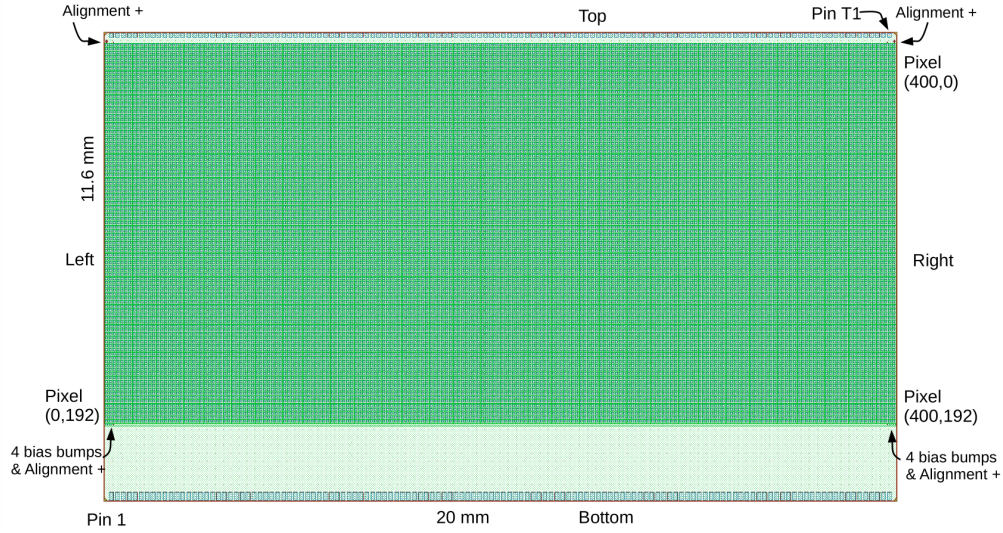


Figure 3.4: Layout of the RD53A chip as presented in Ref. [36]. The size of the pixel matrix (in number of pixel) and of the chip itself (in mm) is depicted. The bright area at the bottom has no pixel cells and holds the global logic elements as well as the power regulators.

In case of the (ITk) Pixel detector, the signal is formed as a triangle with the slope at each side being defined by internal parameters. Afterwards, it is compared to a threshold and the time while the signal is above it is counted in number of clock cycles. The resulting digital value is called *Time over Threshold (ToT)* and is sent out as hit information.

The RD53A FE chip

The ITk Pixel detector will use a completely new FE chip which is developed in co-operation with the CMS collaboration. For this purpose, a new collaboration called *RD53* was founded. This thesis will focus on the first prototype called *RD53A* as it is the only one available [36]. But it will also include an outlook to what will change towards the final version.

The RD53A chip has a size of $20\text{ mm} \times 11.6\text{ mm}$ and is made up of a pixel matrix of $400\text{ pixel} \times 192\text{ pixel}$ with a pixel being $50\text{ }\mu\text{m} \times 50\text{ }\mu\text{m}$ (see Figure 3.4). While built from the same building blocks, there will be two different versions build as the sizes needed for ATLAS and CMS will be different. The final FE chip for ATLAS will double the number of rows in respect to the RD53A chip. The final CMS version will have roughly the same size as the ATLAS one, but with a different aspect ratio.

The pixel matrix is made up of so-called *cores* of $8\text{ pixel} \times 8\text{ pixel}$ with a *core* being subdivided into hit regions of $4\text{ pixel} \times 1\text{ pixel}$. These cores contain all analog and digital components needed to measure the energy deposited by the particles as well as the distributed buffers to store the information until it is read out.

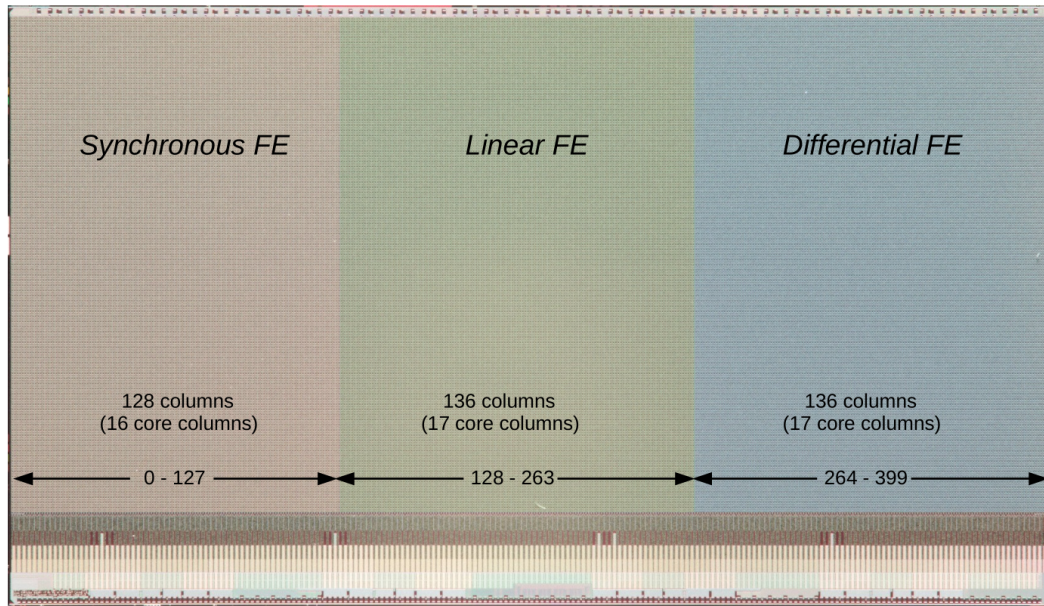


Figure 3.5: The three different flavors of analog front-ends designs are distributed quite evenly [36]. As the number of core columns is not divisible by three, the synchronous design has a core column less.

Being a first prototype, the RD53A chip has three different analog front-end designs called *Linear*, *Differential* and *Synchronous*. All three technologies can be tested in this way. Each flavor occupies roughly a third of the pixel matrix (See Figure 3.5). These are substantially different to allow detailed performance comparisons.

The Linear design is the most common one and uses only a single-ended signal path. The FE chips used in the current Pixel detector and IBL also have linear designs.

The Differential approach tries to minimize the noise introduced by the processing electronic. This is achieved by looking only at the difference between the two data lines used instead of their absolute value.

The Synchronous design uses an *auto-zeroing* feature to adapt for baseline drifts. But this needs to acquire the baseline continuously (every 100 μ s or less).

As all differences between these designs are encapsulated within the core blocks, they can be exchanged with each other. With ATLAS and CMS needing different configurations anyhow¹, they can independently choose the design fitting best to their strategy.

¹The mechanical dimensions were fixed before the chip development started and a redesign of the detector would result in a massive delay. The chip area does not only affect the mechanical structure, but also the data rate needed to read the chip out.

Symbol Name	Encoding	Trigger Pattern	Symbol Name	Encoding	Trigger Pattern
			Trigger_08	0011_1010	T000
Trigger_01	0010_1011	000T	Trigger_09	0011_1100	T00T
Trigger_02	0010_1101	00T0	Trigger_10	0100_1011	T0T0
Trigger_03	0010_1110	00TT	Trigger_11	0100_1101	T0TT
Trigger_04	0011_0011	0T00	Trigger_12	0100_1110	TT00
Trigger_05	0011_0101	0T0T	Trigger_13	0101_0011	TT0T
Trigger_06	0011_0110	0TT0	Trigger_14	0101_0101	TTT0
Trigger_07	0011_1001	0TTT	Trigger_15	0101_0110	TTTT

Figure 3.6: Overview of the trigger commands as given by Ref. [36]. Each trigger commands covers four events with the right most position marking the earliest of them. The trigger pattern *0000* has no command assigned as there is no command sent when no trigger occurs.

Each FE chip needs a way to communicate with the outer world, i.e. to receive commands and to send back the results. To simplify the design and save material in the detector, the data rate for these interfaces is reduced as much as possible. For the RD53A chip, this results in a data rate of 160 Mb/s for the command line and four output lines with up to 1280 Mb/s each. This is already a large increase compared to the FE chip used in IBL, which had only 40 Mb/s for the commands and a single output line with up to 320 Mb/s.

But this still means that the RD53A chip can receive 4 b and send out four times 32 b per bunch crossing, which is still not a lot. To increase the number of symbols understood by the RD53A chip, 16 b (i.e. 4 consecutive bunch crossings) are put together forming two 8 b symbols. In this way, a trigger command can be encoded within a single symbol including a mask for which of the four bunch crossings the FE chip should send out the stored hit information. This is already a big advantage with respect to the currently used FE chips as a trigger command for these occupied five bunch crossings without having any mask. Figure 3.6 shows the list of all trigger symbols with the associated trigger pattern.

The second symbol of a trigger command for the RD53A chip is used for the so-called *trigger tag*, an identifier which is returned together with the event data and therefore used to associate the data with the according collision. In the current system, the FE chips have counters for the event identifiers, which should run synchronous to the counters in the trigger system. Due to radiation effects, a bit in these local counters can flip resulting in a counter mismatch until all instances of these counters are reset synchronously again. If such a bit flip occurs in a trigger tag, only that single tag is affected and thus the number of lost events is limited to these associated with that trigger.

In contrast to the trigger command, the remaining commands are encoded by repeating the command symbol also in the second slot of a 16 b frame. Some of them have additional parameters and therefore need more than one frame. These are called *long commands* and can consist of up to 12 frames. These long commands also have

Command	Encoding	ID/(A)ddress/(D)ata 5-bit Fields						
ECR	2× 0101_1010							
BCR	2× 0101_1001							
Glob. Pulse	2× 0101_1100	ID<3:0>,0	D<3:0>,0					
Cal	2× 0110_0011	ID<3:0>,D15	D<14:10>	D<9:5>	D<4:0>			
WrReg	2× 0110_0110	ID<3:0>,0	A<8:4>	A<3:0>,D<15>	D<14:10>	D<9:5>	D<4:0>	
WrReg	2× 0110_0110	ID<3:0>,1	A<8:4>	A<3:0>,D<15>	D<14:10>	9×(D<9:5>	D<4:0>)	
RdReg	2× 0110_0101	ID<3:0>,0	A<8:4>	A<3:0>,0	00000			
Noop	2× 0110_1001							
Sync	1000_0001_0111_1110							

Figure 3.7: List of commands for the RD53A chip [36]. The *WrReg* command has a short (a single register) and a long (six register values) version distinguished by the bit after the chip ID. The long version should only be used for the virtual portal register.

a separate chip identifier field to address a certain FE chip. This is needed as all FE chips in a module share a common command line, meaning that these are getting the same data stream as input. But this also means, that all other commands are seen as a broadcast to all connected FE chips.

If the commands do not have any parameters, they fit into a single 16 b frame and are therefore called *short commands*. These are high priority commands like resets and can also be sent in the middle of long ones without interrupting their execution. The *No-Op* (i.e. idle) and the *SYNC* commands as well as the trigger commands are also regarded as short commands.

As a frame reaches over four bunch crossings, a FE chip needs a way to align the data streams. This is achieved by the *SYNC* command, which is the only command with more than three consecutive ones or zeros. Also, it consist of two different symbols with the second one being the inverted version of the first one (instead of two identical ones following each other). In this way, the sequence of the *SYNC* command cannot accidentally occur within a valid data stream, making it a distinct alignment marker. Figure 3.7 shows the list of all commands and their parameters.

The trigger commands are the only commands which directly initiate the transmission of data back from the FE chips. Even upon receiving a register read command, the FE chip will not react by sending the requested data immediately. Such a read command only marks a register for readout as soon as the next register frame is sent. If the transmission of these register frames is disabled, the read register command will have no affect at all. With enabled register frames, the read command decides only *what* is read. In the case of no read request being queued in the FE chip, a predefined register is read and its value sent out.

When a trigger command arrives in the FE chip, the associated event is marked for readout. The actual start of the transmission depends on how much data is left in the queue from previous events. The RD53A chip has a simple *first in, first out* strategy implemented, sending out available data as fast as possible. This is possible as the RD53A chip supports only a single trigger level. The next version of the chip

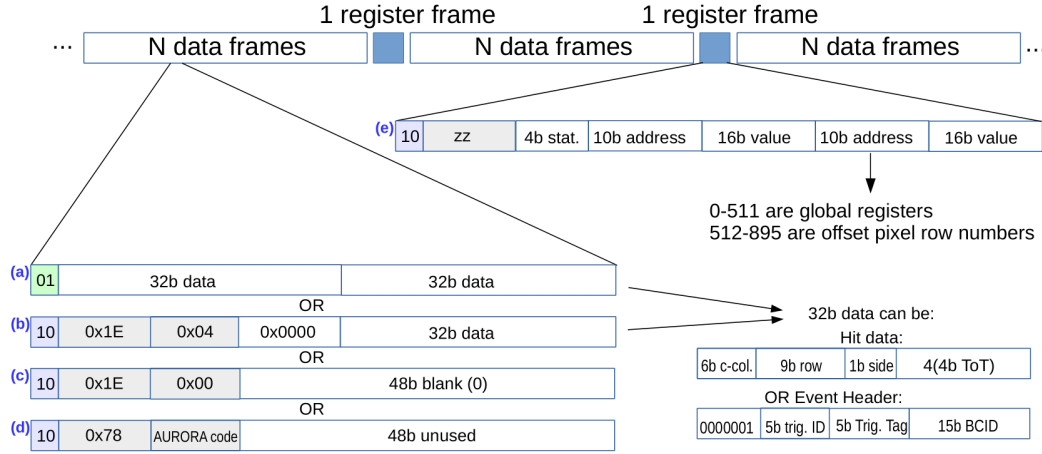


Figure 3.8: The RD53A chip uses four types of Aurora frames [36]. Next to data frames (marked by the bit combination *01* as Aurora header), there are the *End of Packet* frames (starting with *0x1E*) used to signal the end of an event, three kinds of *IDLE* frames with different meanings (starting with *0x78*) and some special frames used for the register readback.

is designed to support two trigger stages with the first trigger marking an event to be kept in the internal memory and the second trigger marking the event for the readout after a fixed latency with respect to the first trigger. If the first trigger does not arrive, the event is deleted to reduce the memory needed. The fixed latency for readout is necessary as events cannot be read out in random order and the readout of the previous event can be triggered until that latency has passed. After that latency, the event data still must wait until all previous events have been read out or deleted due to missing triggers.

Before the data is sent out, it is split into blocks of 64 b and encoded in a protocol called *Aurora 64B/66B* [37]. Therefore, the 64 b blocks are scrambled and a 2 b header is added to distinguish between data and command frames. The header is also needed to find the correct frame alignment on the receiving side. The command frames used by the RD53A chip represent *End of Event*, *IDLE* and some other synchronization symbols. The register frames are also encoded as command frames and are inserted in the output stream periodically after a preprogrammed number of frames. Figure 3.8 shows the output format in detail.

Even if a RD53A chip uses only a small fraction of the output bandwidth, it needs a separate data line for each chip. As this will be the case for many FE chips in the final detector (especially in the outer layers), the final version will have a feature to interleave the data streams from other chips with its own one. In this way, a whole module (i.e. up to four FE chips) can be read out via a single cable instead of four. Next to the reduction of cables needed, this has also the side effect of increasing the space available for routing and other services as well as lowering scattering effects on particles going through.

3.2 Overview of the readout system

The FE chip itself only responds to requests from an outside entity. So a path into the detector (for the requests, the *downlink*) and out of the detector (for the responses, the *uplink*) is needed. Thereby, it is helpful to keep both paths as closely together as possible as the response needs to be matched to the corresponding request. So the paths share most of their components, but also have components which are uniquely assigned to only the *downlink* or the *uplink*.

Figure 3.9 shows an overall picture of the readout structure.

The downlink

In normal operation, i.e. the so-called *data taking* mode, the source of actions is the trigger system, which distributes its information via the TTC interface. These actions are not only triggers, but also resets for bunch crossing and orbit counters used to identify events.

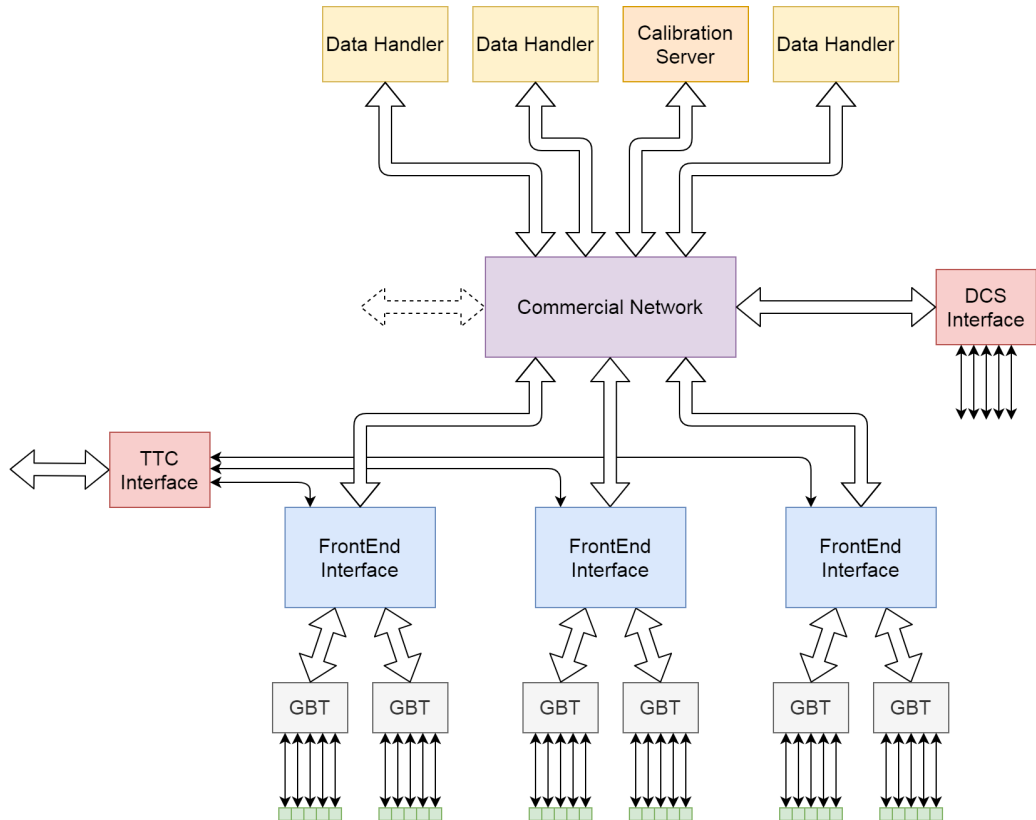


Figure 3.9: Overview of the readout system with the detector elements in green at the bottom and the servers with readout software at the top. The triggers for event readout come via the Trigger, Timing and Control (TTC) interface.

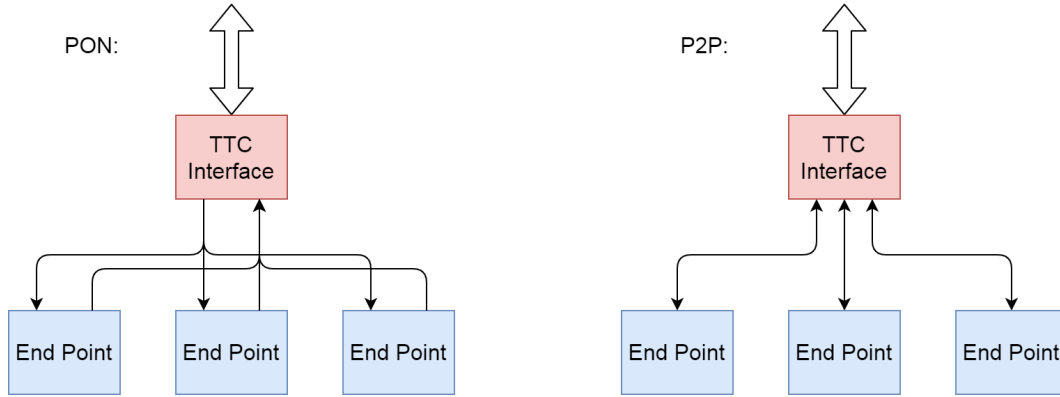


Figure 3.10: Depicted are the two topology variants under discussion for the distribution of the TTC signals. The PON version needs less optical components while the P2P version has higher capacity for individual signals.

It is not yet decided if the TTC interface will use a passive optical network (PON) or peer-to-peer (P2P) connections (See Figure 3.10). While PONs are easier to implement (only a single transmitter and receiver is needed at the TTC interface), they have the problem of shared bandwidth. This is no big issue in the downlink direction as the trigger and reset information are the same for all detector parts. But in the opposite direction (i.e. the uplink), the end points are served in a round robin manner resulting in each end point being able to send updates less often.

As these updates also contain a trigger veto signal in case the endpoint cannot handle any additional events, a lower frequency results either in requiring larger reserve capacities to buffer all events until the next update (i.e. the next possibility to assert the veto) or in the veto being set more often as it could not be guaranteed to not lose any data. When the veto is set at any point, a lower update frequency results also in the veto being set for a longer period until it can be cleared again.

With the usage of P2P connections, any end point can send an update whenever it is needed without waiting until it is its turn again. The downside is that a transmitter and receiver is needed for each end point, increasing both costs and space needed.

The end points are called *FE interfaces* within this thesis and allow access to the detector. They collect all information to be sent to the FE chips, interleave these streams and encode the resulting stream. The FE interfaces also need to generate the trigger tags if the TTC system does not already provide these. To increase the efficiency, several of these streams are encapsulated in a proprietary protocol called *GBT*² and sent out via an optical fiber. Figure 3.11 on the next page shows an overview of the GBT architectures.

The GBT protocol defines a frame to be sent for each bunch crossing, with a stream occupying an amount of bits according to the bit rate of the stream divided by the

²There are two versions of it: the older *GBTx* and the new *lpGBT*. As the differences do not matter for now, both are grouped in the term *GBT*.

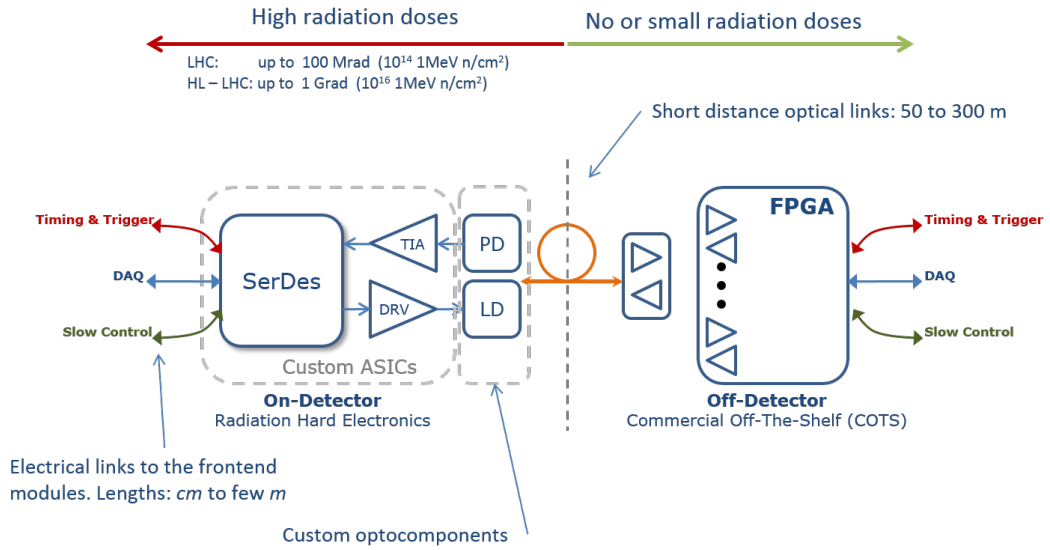


Figure 3.11: Architecture of a GBT link [38]. It contains a custom, radiation hard application specific integrated circuit (ASIC) within the experiment, a bidirectional optical transmission path and an FPGA with specialized firmware blocks at a radiation safe distance.

bunch crossing frequency (i.e. 40 MHz). So, a stream for a RD53A chip will occupy 4 b as it needs a data rate of 160 Mb/s. The GBT protocol is decoded at the other end of the fiber by a dedicated ASIC, after being converted back from an optical to an electrical signal. This ASIC extracts the streams and transmits each one on a separate port. From that point on, the streams are distributed via thin cables until they reach their destination (i.e. the FE chip) where the commands are received and interpreted. In case of the ITk Pixel detector, these cables will have a length of up to 6 m as there is not enough space to place the GBT chip (and the remaining components around it) within the detector volume.

The uplink

The results of the requests go back the same route that was used for the downlink. First as an electrical signal on thin cables until they get to a GBT ASIC again. It does not need to be the exact same chip as a single GBT chip can serve more FE chips on the downlink path than on the return path. This is especially true as a single command line is connected to several FE chips while the return lines can be separated (See Figure 3.12). But this is fine as long as it is in the same group of GBT chips as the associated fibers are organized in bundles, too, and a bundle is served by the same FE interface.

At that interface, the GBT protocol is decoded again, leaving the transmission protocol of the FE chip (i.e. Aurora in case of the RD53A chip). It would be possible

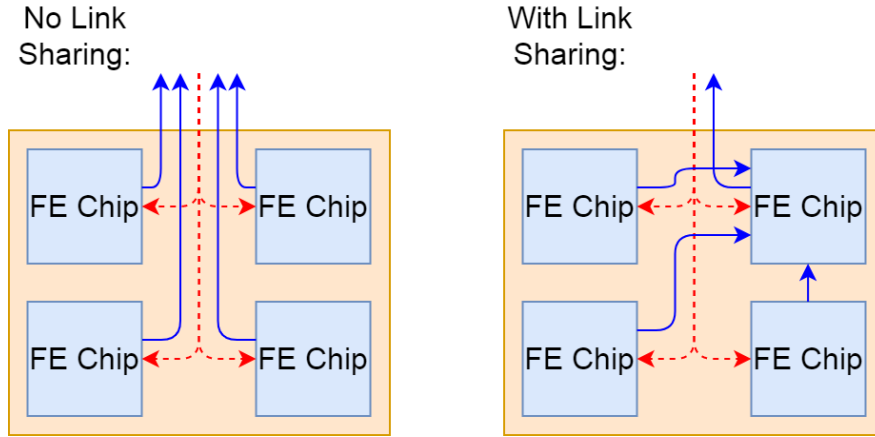


Figure 3.12: While all FE chips of a module share the same downlink, the uplink was planned to be separated for each FE chip (i.e. no uplink sharing). The capability of uplink sharing was added later on to reduce the number of cables in sections with low bandwidth requirements.

to forward everything through the network to a commercial server and do the remaining decoding in software, but this would mean to forward the whole data stream coming out of the GBT decoder. This would include not only all the overhead of the remaining protocols, but also the idle symbols and potentially unused links. Another downside is that the transmission protocols are designed to be easily decodable in hardware, but this is not necessarily true for a software implementation as well.

So, the FE interface should not be too transparent and forward everything to software. Instead, it should contain some data processing as long as it can be integrated without any bigger effort (both in terms of man power and hardware resources). A good example for this is the decoding of the transmission protocol (i.e. Aurora in case of the RD53A chip) and the stripping of sideband information (like idle symbols). This also helps to route the encapsulated data to the desired destination as the type of information can be accessed (e.g. the register frames can be separated from the event information in case of RD53A chip). For example, the RD53A chip is able to measure environmental values³ like its temperature. Such data are forwarded to the detector control system (DCS) for monitoring.

The event data is sent to the servers called *Data Handlers* where the software collects and combines the event fragments from a larger detector region. The software gets the trigger information from the FE interface (both the counter values used by ATLAS as well as the trigger tags if used by the FE chips) and compares them to the values included in the event data. If these values do not match, an error is reported and the corresponding event might not be used for further processing. When the fragments from all FE chips (which the data handler instance is in charge of) are

³It has an ADC combined with an analog multiplexer to measure the values given by different sensors.

combined, the data is sent to the next stage where the data from the whole ATLAS detector is used to reconstruct the event and the final decision is made if the event is kept (i.e. written to tape) or not.

In addition to the normal path for the event data, a partial copy is sent to the calibration servers to monitor data transmission errors and data quality. This is needed to check whether the modules perform as expected or whether any action is needed.

Calibration

A different but not less important operation mode is the calibration of the detector. It ensures that the detector delivers a constant performance as the continuous bombardment with particles changes its behavior over time. Other radiation effects like trapped charges or defects in the substrate of the ASICs affect the behavior (e.g. the voltage needed to deplete the sensor will increase with radiation dose) and need to be compensated.

The calibration is done by a number of tunings, whereby a tuning consists of several scans with the adaptation of a set of parameters in between. The scans differ in their behaviors, but all of them measure a specific parameter by sending specific trigger sequences to the FE chip and generating a histogram of the parameter distribution as a result. As the FE chips have too many pixels to trigger all of them at the same time, the matrix is split up in a defined pattern called *mask step* (See Figure 3.13) and each region is triggered individually. When all triggers have been sent to such a region, the mask is shifted and the next group of triggers is sent. This repeats until the whole pixel matrix is covered.

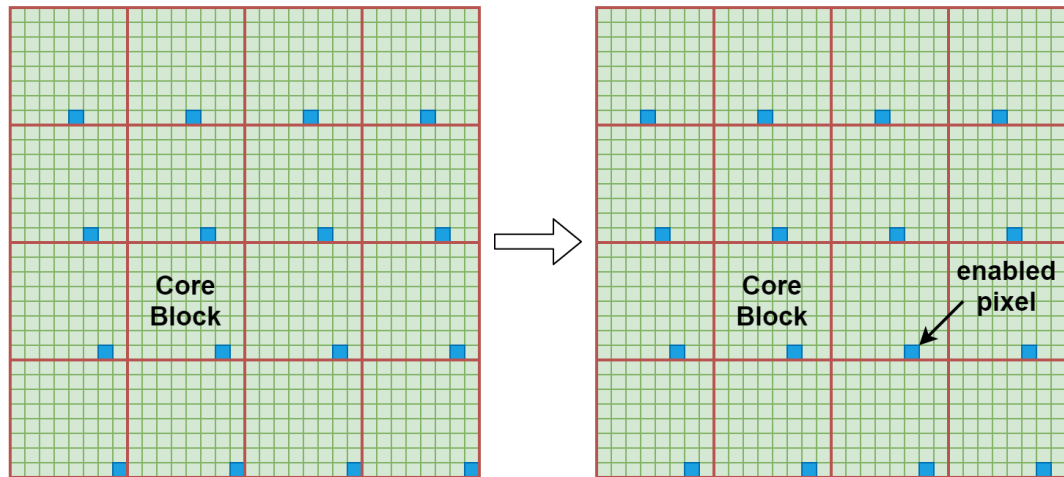


Figure 3.13: Schematic view on mask steps for RD53A chip. A single pixel per core block is active at a given time. After all triggers for a mask are processed, the mask is shifted and the next triggers are sent again.

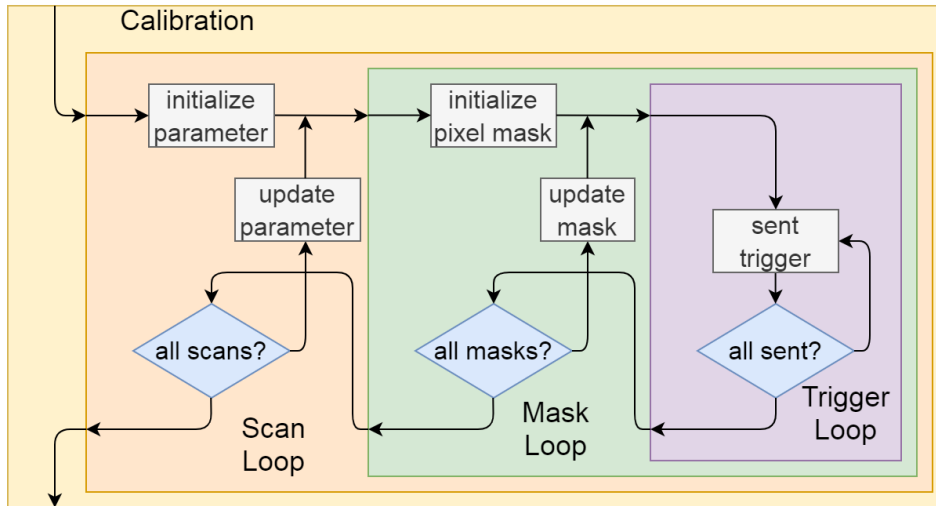


Figure 3.14: Schematic view on the nested loops used for detector calibration. Some scan types are build of several nested loops, which are not shown here.

The simplest scan is the *digital* scan, which only activates a test signal within the pixel logic to directly overwrite the hit marker. It does not make any use of the analog part of the pixel cells and thus is useful to check if the readout logic itself is working. The digital scan is also used to check if the communication with the FE chip is working at all. With each active pixel generating a hit, the number of hits coming back as the result of a group of triggers is well known and lost data is easily detected.

When the digital scan is working, an *analog* scan can be performed where a defined charge is injected into the sensor. If a charge around the threshold of the discriminator within the pixel cell is injected, the noise within the pixel cell (e.g. electromagnetic noise, charged particles like atmospheric muons going through, ..) will cause the pixel cells to generate a hit for some triggers.

As the analog scan is the most basic operation including the analog part, it is used to build up other scans and tunings. So for example, a series of analog scans with different injection charges can be used to determine the individual threshold of each pixel cell and is therefore called a *threshold* scan. On the other hand, a series of analog scans with the same injections can be used to tune the global threshold offset. This is done by interleaving analog scans with the adaptation of the global threshold until roughly half the pixels generate a hit for a given charge. Figure 3.14 shows a generalized view of the scan algorithm.

A very special variant of scans is the noise scan where only triggers are sent without any injection. The events coming back will only include hits generated by noise or due to other problems (e.g. a permanent connection of the signal to the supply voltage). So, this can be used to check which pixels of a FE chip should be disabled permanently as they will not deliver useful physics data.

The calibration of a (part of the) detector is done in a stand-alone operation as each detector has different requirements and procedures for it. But this means that a different trigger source is needed, which is controlled by the calibration routine. In addition, the data coming back from the detector needs to be processed (i.e. the hits need to be histogrammed). As the data handlers used for data taking are not meant for such a task (they lack the processing power needed), there are some special servers running the calibration software.

For a small number of FE chips, the calibration servers can generate the triggers in software and sent them to the FE interface via network. But if a bigger chunk of the detector should be calibrated, it is more efficient to include a simple trigger generator within the TTC or FE interface. The former option would allow bigger detector parts to be calibrated synchronously, while the latter one enables smaller test setups in laboratories to also use it. In the end, it is most likely that both options will be realized.

Another reason to have dedicated calibration servers is that these can do additional monitoring and error checking during data taking operation. One option would be to collect all hits from a set of FE chips and do some kind of passive noise scan. As the hits from physics are sparsely distributed (especially in the outer layers), a faulty pixel can be detected rather easily. A different possibility is to check the data streams coming from the detector for bit errors. If these are too numerous, the settings for the corresponding link should be adjusted or the FE chip must be disabled until the problem can be fixed.

Continuous Reconfiguration

A special requirement for the ITk Pixel detector is the continuous reconfiguration of the FE chips. Due to radiation effects, the configuration of a FE chip can change over time. Charged particles can ionize the material potentially leading to memory cells flipping their content. In theory, it is possible to implement countermeasures against bit flips like triplicating the logic with a majority voter afterwards. But this would take too much area on the chip and would only reduce the chance of these as the voter would go for the wrong result if two out of three instances are hit simultaneously.

The expected rate of these effects is on the order of a bit flip every few seconds within a single FE chip. This means, the whole configuration of all FE chips needs to be rewritten as often as possible (i.e. at least once a second) to fix the flipped bits. As the write commands can be interleaved with short commands like trigger or sync frames, these can be used to rewrite the configuration whenever there is unused bandwidth. But this requires a low latency access to the configuration at the point, where such short commands are inserted into the data stream. In this regard, the best solution would be to store it in the FE interface, but this needs a compromise between memory space, bandwidth and latency.

To simplify this operation, the configuration is stored as encoded bit stream ready to be sent to the FE chips. But this increases the size to 629 kB, compared to 158 kB for the pure register values [39]. As the tuned values for each pixel are included

within the configuration, each FE chip needs its own bit stream. So the size of a single bit stream needs to be multiplied by the number of FE chips served and thus can reach several 100 MB.

But the memory space is not the only requirement here as the data needs to be sent to the detector. So the memory bandwidth must also match the one of the links toward the detectors. Such a downlink can serve up to eight modules, resulting in a combined bandwidth requirement of 1280 Mb/s per downlink fiber.

3.3 Implication for the front-end (FE) interface

The FE interface itself functions as a gateway between the detector and the commercial systems. Therefore, it has a very specific set of requirements to fulfill to make this connection working. This section will go into more details about what exactly needs to be done.

The FE interface can be built in different ways by choosing different solutions for the diverse problems to be solved. This results in varying numbers of components involved. However, there is a single component which stays the same for all of them: an FPGA. Its mixture of parallel processing with real-time capability, support for a wide range of protocols and still being programmable while being regarded as hardware is the perfect solution at this point. Therefore, the FE interface will be an electronic assembly with one or more FPGAs.

In addition to that, the FE interface will have three groups of interfaces:

- A set of interfaces towards the servers for communication with the servers doing the data processing.
- A port for the connection with the TTC interface, this might be a bidirectional, optical link, but should also support other variants for laboratory operation.
- The links towards the detector, which are grouped in bundles of optical fibers.

The type of interface towards the servers determines a first limit on how many detector modules can be served by a single FE interface unit as the interface towards the servers needs to forward all data coming from the detector. Some kinds of these interfaces will work only with a single instance, which can be split up logically within the FE interface. For example, a PCIe card will most likely have only a single PCIe connector and therefore a hard limit on the usable bandwidth. As PCIe is not suited for communication with other computers, it would also require additional hardware for translation into a different interface.

Other interfaces towards the servers are more flexible as they have smaller connectors with less impact on the printed circuit board (PCB) design. They make usage of serial protocols and high data rates on these single lines. In addition to that, many of these interface have variants with multiple instances being packed into a combined connector. For example, optical interfaces can be placed in high density and can therefore be scaled up as needed.

The number of FE chips being served by a FE interface is defined by the ratio of the data rates of the interface towards the servers and the detector uplinks. This is because of the uplinks are more limited due to the higher bandwidth than the data streams towards the detector (i.e. the downlinks) and the latter one are also shared by several FE chips.

As the interface towards the servers runs much faster than a downlink channel, the latter one can be easily flooded resulting in data getting lost. On the other hand, sending a new chunk of data just in time when the previous data packet ends is not possible as software cannot time packets with such a precision. Therefore, some buffers are needed to hold data until it is sent out. To enable independent data transfers for each link, a separate buffer is needed for each link instead⁴ of a big global one. Because of the available memory being limited, these might be rather small ones holding only a few chunks. In this case, using a PCIe card in a computer has the big advantage of the host memory being directly accessible with quite low latency allowing pre-buffering. For all other types of interfaces towards the servers, this problem has to be solved differently.

The same problem occurs for the continuous reconfiguration, where the configuration of each FE chip needs to be stored (if it is not sent repeatedly by the software). The internal memory of the FPGAs is too small for devices in the desired price range. There are devices with internal High Bandwidth Memory (HBM) stacks offering several GB of memory, but these are the high-end class versions with an corresponding price tag. Therefore, some external memory would be the best choice to this problem. This would be in addition to any memory used to buffer the uplink data.

In a crate based system, as considered in this theses, there are only two viable solutions. The first one aiming for a setup with smaller FPGAs (i.e. many smaller cards for better modularity) would place some memory components on the FPGA card. As the number of links will be limited, also the number of required memory components is quite low for each card.

If a higher number of links per card is desired and therefore a larger FPGA is needed, the increasing number of memory components is getting a problem, in terms of space and number of general purpose input/output (GPIO) pins. In that case, it makes sense to employ larger FPGAs including HBM memory to get around the external components.

⁴Or at least in addition to a global one.

Chapter 4

FRontEnD Data InterfacE (FREDDIE)

The FRontEnD Data InterfacE (FREDDIE) is an implementation of the FE interface described in the previous chapter. As most components of the final system are still missing, it uses a simplified setup with the following properties:

- FREDDIE is implemented on a *ZCU102* evaluation board by *Xilinx*, providing a *Zynq UltraScale+* FPGA, two FPGA Mezzanine Card (FMC) ports and four small form-factor pluggable plus (SFP+) connectors. The FMC ports are used for special interface cards for the FE chips. The SFP+ connectors can be used for a separate 10 Gb/s Ethernet connection each or combined to a single 40 Gb/s Ethernet interface.
- The FE chips are single RD53A chips placed on a single chip card (SCC), which uses a *DisplayPort* connector for the command and data lines. These are directly connected to the FMC interface cards via commercial *DisplayPort* cables. So, there is no conversion into optical signals as well as no usage of the *GBT* protocol.
- The interface towards the servers is implemented as a 40 Gb/s commercial network using the UDP protocol on top of IPv4.
- There is no TTC interface implemented as such a system can be emulated within the FPGA on the *ZCU102* evaluation board.

The basic design of FREDDIE foresees a wrapper for each major function block (See Figure 4.1) on the next page. In addition to these, there is the *application wrapper* grouping all processing blocks except the network stack to enable a fixed interface independent on how the data transmission is organized. Also, the device specific IO blocks are separated in a separate wrapper to ease the implementation on different platforms. So, the external interface of the application wrapper has only three parameters:

- the bus width of the interface towards the servers which is also used for the internal buses
- the number of data generators included
- the number of supported FE chips

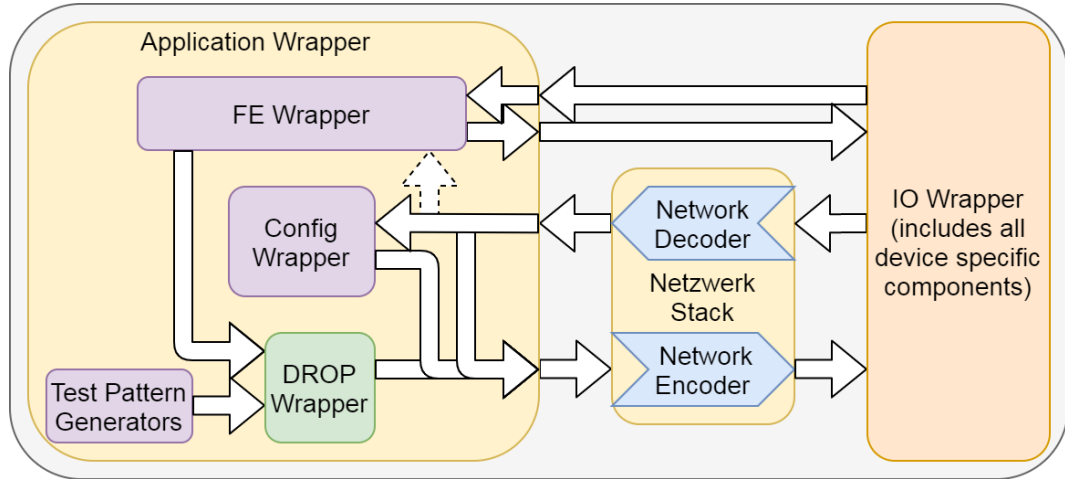


Figure 4.1: Organization of the different firmware blocks with the FREDDIE system. Besides the network stack and IO specific components, all other functional blocks are grouped within the application wrapper.

The application wrapper is shown in more details in Figure 4.2. Included in the application wrapper are the *FE wrapper* including both the downlink and the uplink path, the *configuration wrapper* also used to fill the broadcast memory, the *DROP wrapper* for adding needed information to communication with the processing side and data generators to test and characterize the communication. The different parts are described in the following sections in more detail.

FREDDIE uses mainly the Advanced eXtensible Interface (AXI) in version 4 as format for the data signals, which is part of the Advanced Microcontroller Bus Architecture (AMBA) [40]. More precisely, it uses the continuous aligned *AXI4-Stream* version. This means, that all bus cycles with valid data (except for the last one) must have all bytes filled, but there might be pause cycles (i.e. without any valid data) in between. A result of this strict rule is, that all firmware modules adding or removing data to/from a transfer must realign the data as long as the data being added or removed is not an integer multiple of the bus width.

The *AXI4-Stream* interface has several components with all of them being optional except for the valid signal. The used components are:

TVALID is set by the master and indicates that all other signals have valid values and should be transferred. A transfer takes place when both TVALID and TREADY are asserted. Once asserted, TVALID must stay set until the transfer was executed.

TREADY is set by the slave and indicates that the slave can accept a transfer in the current cycle. A transfer takes place when both TVALID and TREADY are asserted. In contrast to TVALID, TREADY may get deasserted again once a transfer was executed.

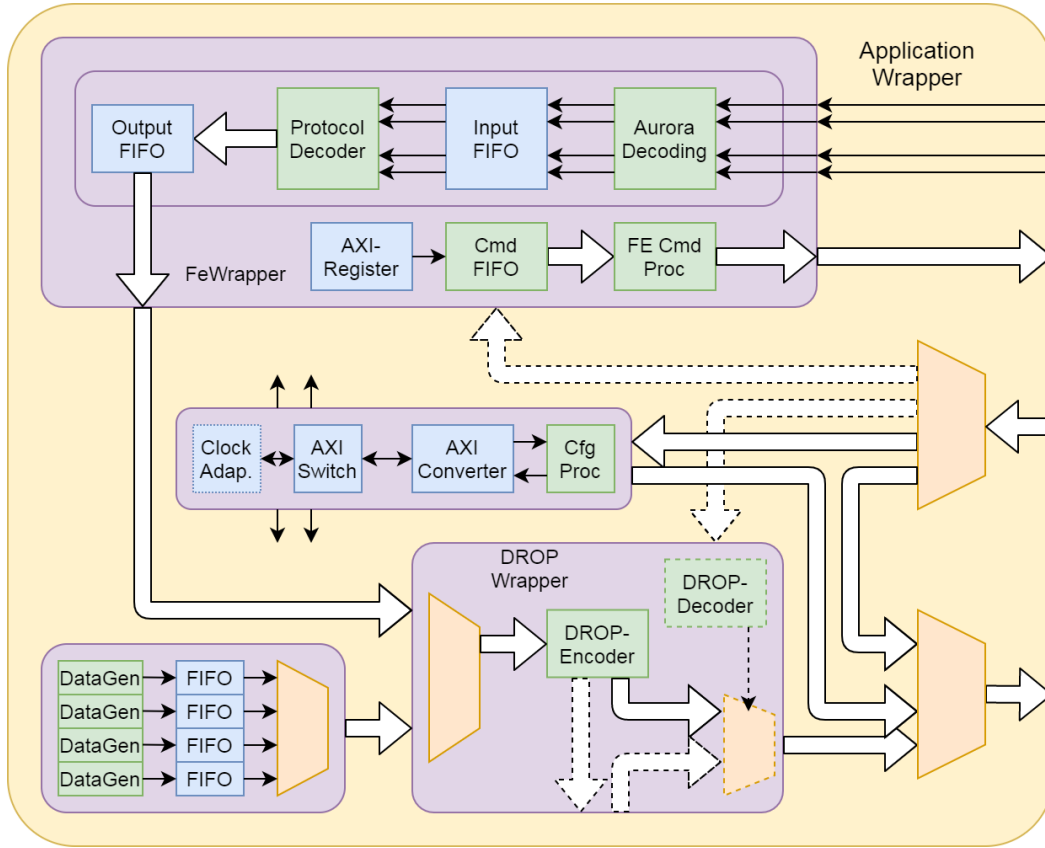


Figure 4.2: Overview of the application wrapper including all parts but the network stack and the device specific IO components.

TDATA [(8n-1):0] is the primary data payload that should be transferred from the master to the slave. The width of the data payload is an integer number of bytes.

TKEEP [(n-1):0] is the byte qualifier indicating whether the content of the associated byte of TDATA is processed. As continuous aligned streams are used, it must be fully asserted in all but the last transfer.

TLAST indicates if the current transfer is the last one for the current block of data being transferred. The next transfer after this is the start of a new block of data.

TDEST [(d-1):0] provides routing information for the data stream.

TUSER [(u-1):0] is the user defined sideband information that can be transmitted alongside the data stream.

In addition to these, there are the *TSTRB* and *TID* fields but these are not used.

It should be noted that different parts of FREDDIE use different combinations of these signals as the requirements are different. For example, the receive path of the network stack does not use the *TREADY* signal as the data needs to be processed in time or is lost otherwise. There is no memory between the network interface and the network stack that could store the data until the network stack is ready. However, the transmitting path needs the *TREADY* signal as the network stack runs faster than the network interface and needs to be stopped regularly. As the overall design aims at forwarding the data from the detector towards the processing servers, the majority of the modules uses the same bus structure (including the bus width) of the network transmit path.

4.1 Path towards the detector

On the transmit side towards the detector, there are several tasks to be done. Besides the transcoding of the TTC signals and the insertion of data arriving through the interface towards the servers, basic communication tasks like the generation of synchronization words in regular intervals as well as filling empty frames with *idle* frames must be executed. In addition to this, some helper tasks like the configuration of the FMC interface card or TTC emulator are also needed.

Protocol encoding

As the RD53A chip uses the recovered clock from the data stream to clock all internal logic, it needs to synchronize the internal phase locked loop (PLL) to it. To assist in this, the FE interface should send a data stream with a regular pattern. This is done by sending *idle* frames.

After locking the internal PLL, the RD53A chip needs to find the alignment of the 16 b frames in the data stream by looking for synchronization frames. As the FE interface does not know, when the internal PLL has locked, it is sending the synchronization frames in a fixed interval. This also helps to keep the alignment after the chip start-up phase.

The next step is the configuration of the chip. This can happen via the continuous reconfiguration mechanism or by sending the explicit write commands via the interface towards the servers. In both cases, the preencoded frames must be inserted into the data stream being sent out. The configuration streams need to be paused from time to time as the synchronization frames are still periodically sent.

After the configuration, the chip is ready for operation and the TTC signals are enabled. These signals are composed of flags for triggers (i.e. *Level 0 Accept (L0A)*) and resets (i.e. *Bunch Counter Reset (BCR)* or *Event Counter Reset (ECR)*) and additional identifier fields (like *Bunch Counter ID (BCID)* or *Level 0 ID (L0ID)*). If a flag is set, the corresponding action should be done in the corresponding clock cycle of the LHC clock (≈ 40 MHz).

As described in Section 3.1, the ITk subdetectors use 16 b frames sent at a rate of 160 Mb. Thus, a single frame covers four bunch crossings and therefore up to

four triggers need to be packed into a command frame. This needs to be done in a deterministic way as the time of arrival at the FE chip selects which events are marked for read out. The trigger command also needs a trigger tag to identify the event data coming back. It is not decided yet if the tag is generated by the TTC system or locally within the FE interface. If the TTC system generates the same tag for the same event in all detector parts, it needs to assume an alignment of the 16 b frames. But this means, that the FE interface needs a way to synchronize the alignment of the different links with the TTC system. If the tag is generated locally, this might not be necessary, but it would result in different tags for different fragments of the same event. As the software translates the tag back into the identifier used by the global ATLAS scheme (i.e. BCID and L0ID), this should not be a problem.

With FREDDIE being a first test implementation, it is not designed to support the data taking operation. Hence, it also has no TTC interface to receive any external event identifiers to relate a generated trigger tag to. But this is also not necessary as it is not needed to align the events coming from multiple FE chips with each other. In a later design, the alignment of the FE links can be synchronized by the BCR. The BCR is sent at the start of each orbit of the proton beams and is meant to reset the bunch crossing counter. If trigger tags are used, there is no need to use the bunch crossing counter anymore. The RD53A chip still supports usage of the bunch crossing counter for testing reasons as it is the first prototype of this new approach.

But the BCR can still be used to align the command stream as it is related to the first bunch crossing its command frame covers and would be needed to arrive at the correct clock cycle. Changing the alignment means to intentionally break the synchronization with the FE chips and rebuild it with the new alignment. This takes some time and should only be done once at the start of a run as the FE chips cannot receive any commands, while they are not synchronized.

The ECR is sent periodically (in the order of seconds) to reset the event counter (i.e. the number of L0As seen since the last ECR), but is also meant to clean the data path. So any events being stuck and unprocessed within the readout queue should be deleted upon the arrival of an ECR. While the RD53A chip has no event counter anymore (due to the trigger tags), it still uses the ECR to clean the event first-in/first-out (FIFO). This needs to be accounted for if the FE interface relies on FE internal states (which is not the case within FREDDIE).

These commands triggered by the TTC flags must be translated to the corresponding command frames and also inserted into the command stream as the last missing part. Figure 4.3 shows the overall structure.

In an attempt to build a unified firmware for both ITk subdetectors, this structure was further optimized. The part responsible for the generation of the BCR and ECR frames was converted into a configurable *fixed pattern* generator. While the part for BCR and ECR is configured at compile time by selecting the firmware flavor, it also includes an additional field to be configurable at runtime. This can be used to issue a special calibration pulse in a fixed time before a trigger command. As only one TTC based frame can be sent out at a given time, the output of these modules can be combined to a single interface. The same is true for the generator of the *idle* and

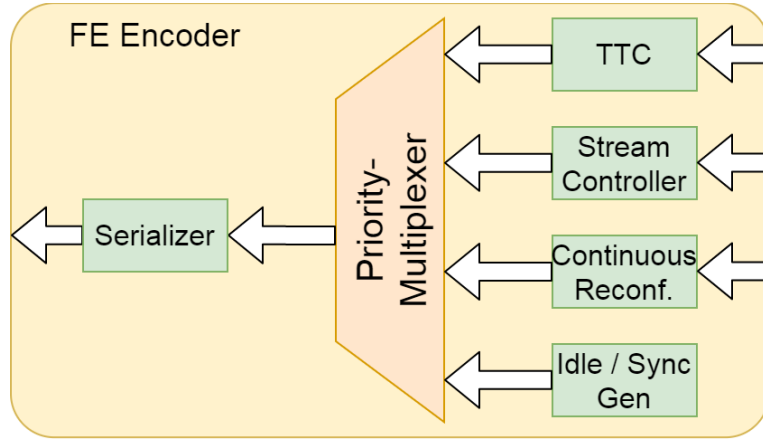


Figure 4.3: Logical structure of the FE encoder. It merges the four types of data streams according to their priority and serializes the resulting stream. The TTC signals need to be encoded while the streams from the software and the continuous reconfiguration are already encoded. The idle and synchronization symbols are inserted if needed.

synchronization frames. It would also be extended to be configurable at compile time to match the protocol of the selected subdetector.

As the continuous reconfiguration is also mutually exclusive to the commands coming from the software and both need a distribution subsystem (i.e. a firmware part distributing the data from a single interface to the selected FE encoder), both can share that distribution part and therefore use the same interface at the FE encoder. The result of these optimizations is shown in Figure 4.4.

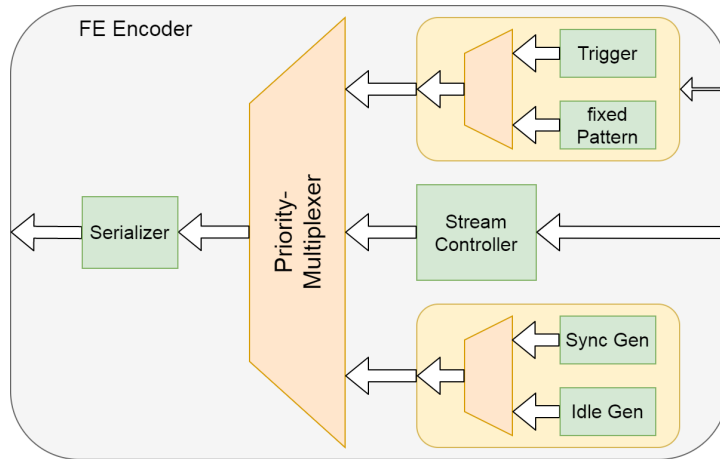


Figure 4.4: More detailed overview of the FE encoder. In this version, the firmware blocks are optimized for the different tasks to be done.

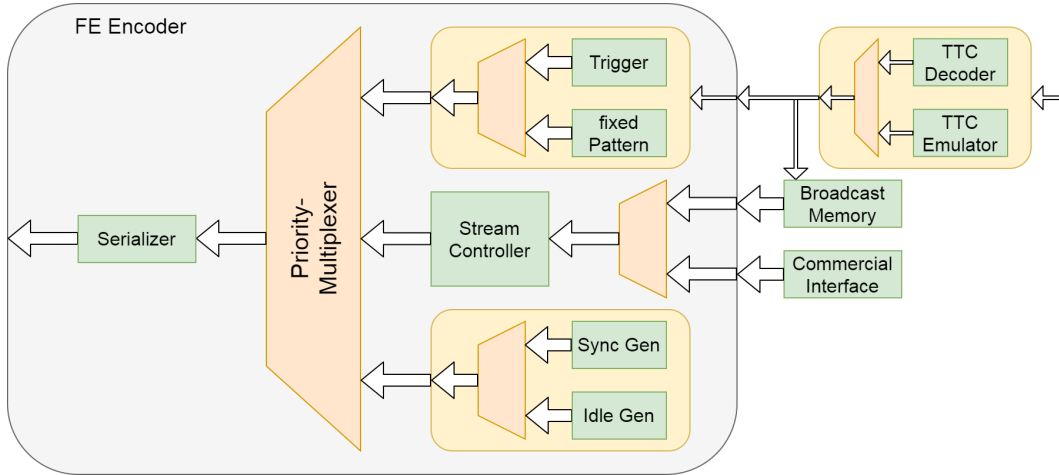


Figure 4.5: Alternative downlink structure with a broadcast memory being used to store the configuration and a TTC emulator to trigger the transmission of its content.

Another possibility to implement the continuous reconfiguration is the introduction of a *broadcast memory*. It acts like a configurable fixed pattern generator, but for much longer command sequences. It could also include space for several sequences that can be sent alternately. To optimize the space available for the sequences, it would use a single memory block with a sequence being selected by its start address and the length of the sequence. The transmission is then triggered via the TTC system or a command from the software. The downside of this implementation is that the FPGA's internal memory is not enough to implement such a block for all links within the FE interface in the final system.

To make better use of the broadcast memory and generally improve the work in small setups, the TTC interface is extended by an emulator. While this TTC emulator is not able to generate the complex pattern of the real ATLAS system, it should be enough to test most of the functions needed by the FE interface. The extended design is shown in Figure 4.5.

The last part is the inclusion of the output stage, which would be the GBT protocol in the final system. As FREDDIE does not support optical transmission and uses the FMC interface card instead, this part is left out. So the serialized bitstream is sent out via the corresponding GPIO pin of the FPGA.

4.2 Path from the detector

The data coming from the FE chips needs to be received, decoded, sorted and sent out to a server within the network for further processing. The focus of this section is on the first three parts as the network interface is described in a separate section (see Section 4.3).

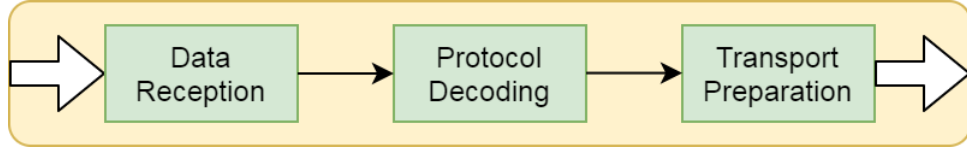


Figure 4.6: Simplified block diagram showing the processing steps needed in the receive path.

Figure 4.6 shows a simplified block diagram of this processing done in the receive part of FREDDIE. The data reception consists of the receiving of the signals sent by the FE chip and its conversion into a fixed format. The protocol decoding can be split up into a first stage for alignment and descrambling of each Aurora lane (here called *Aurora decoding*) and a second stage taking care of the lane bonding and extraction of the event and register data (here called *FE decoding*). The third stage collects the output streams of the decoding block and forms packets to be sent out via the network interface.

Data reception

The FMC interface card forwards the signals to normal GPIO pins of the FPGA. Thereby, each Aurora lane occupies two pins as the FE chip uses differential signaling for data integrity reasons.

The FE chip uses the recovered clock from the command stream to generate the clock for the transmit logic. Therefore the data stream coming back is still synchronous to the system clock of FREDDIE. Thereby, the actual data rate can be changed between four predefined values by selecting different values for the clock divider in the FE chip.

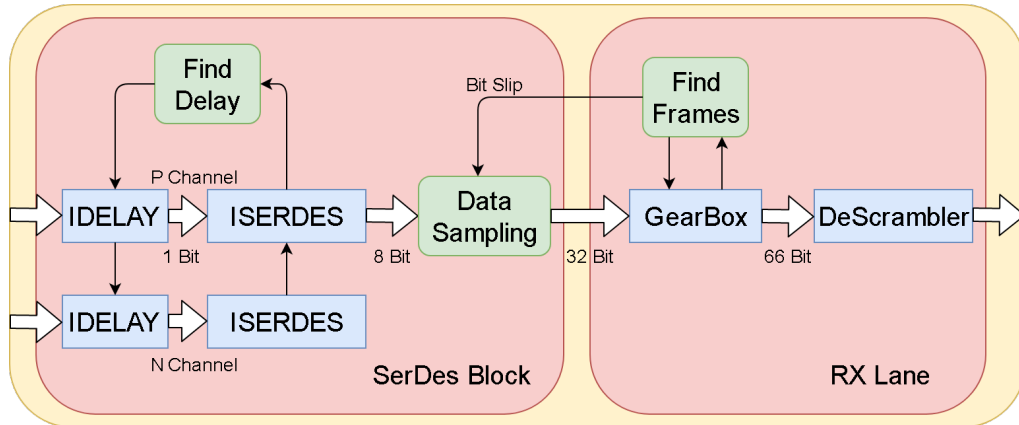


Figure 4.7: Detailed block diagram of the input stage. It contains the phase adjustment, data rate adaptation as well as the frame alignment and descrambling.

The possible data rates are:

- 160 Mb/s
- 320 Mb/s
- 640 Mb/s
- 1280 Mb/s

The receiving stage shown in Figure 4.7 runs with a fixed data rate of 1280 Mb/s, resulting in oversampling for all other data rates. This is taken care of at a later stage.

While the frequency of the signal is known, the phase shift due to the transmission delays is not. So the first stage is a delay block to shift the phase with respect to the sampling clock. This is followed by a SERDES block performing a 1 : 8 deserialization. The chain for the positive line is thereby called the *master* chain, while the one for the negative signal is called the *slave* chain.

Both stages are separated for both lines of the differential signal. This is used to scan for a stable sampling point¹. The algorithm (as depicted in Figure 4.8) tries to find the leading and trailing edge of a bit position. To achieve this, the initial delay values are set to ensure that the time offset between the master and slave delays includes a full bit position. This means that the sampling point of the master chain is behind the trailing edge and the one of the slave chain is before the leading edge (See Figure 4.8(1)). Afterwards, both delay values are successively reduced (while keeping the constant offset between master and slave) as the exact position is not known (See Figure 4.8(2)).

¹The SERDES block of the negative signal line is only used by this algorithm. The data forwarded to the next processing stages is purely taken from the SERDES block of the positive line of the differential signal.

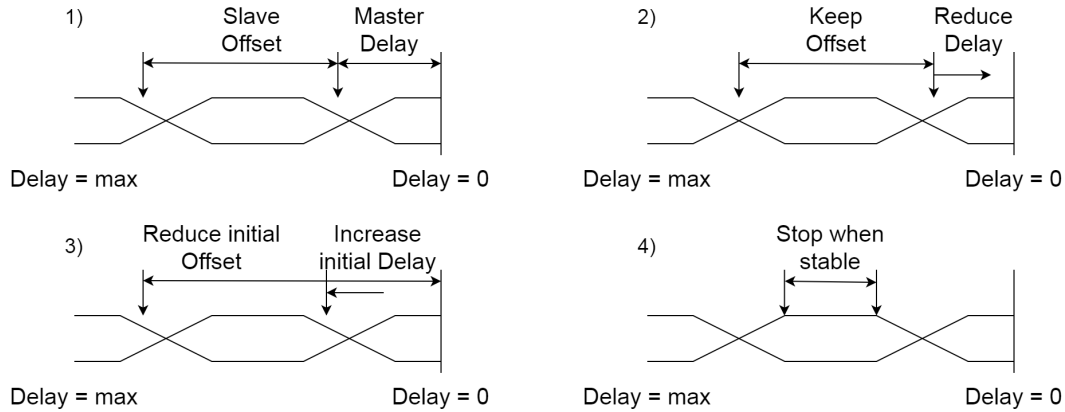


Figure 4.8: Algorithm to find a stable sampling point. The algorithm tries to find both edges of a bit by scanning over a given range while reducing the offset between both in each run. When both sampling points return the same data for a given period, a stable sampling is assumed.

For each combination of delay values, the output of both SERDES blocks is compared. If the output of the slave SERDES equals the inverted one of the master SERDES for a predefined time period, it is assumed that stable sampling points are found for both edges and therefore the algorithm ends (See Figure 4.8(4)). If the output of both chains does not match, the next set of delay values is loaded and tested.

When the master delay reaches a value of zero, it is assumed that the offset between master and slave is larger than the time between both edges. Therefore, the scan starts again with the slave offset being reduced by one. This is done by resetting the slave delay to its initial value and setting the master delay to the start value of the last scan incremented by one (See Figure 4.8(3)). In this way, the offset between both is reduced until a valid sampling point is found while keeping the overall search area the same. If the offset between both delay blocks is reaching zero, no stable sampling point was found.

It should be noted that all eight bits of the SERDES output are compared, meaning that all eight bits need to match to reach a stable sampling point. Furthermore, the algorithm assumes a data rate of 1280 Mb/s. If the real data rate is lower, a single bit of the real data rate will be regarded as several bit with the same value by this algorithm. Therefore, the initial delays should be increased accordingly to accommodate for the larger bit period. Otherwise, as the hardware is able to handle the full data rate, using the same initial values as for the full data rate should also return a stable sampling point, even if to two compared samples are not the edges of the bit position. So, the phase scan can be done independent on the actual data rate.

When a stable sampling point is found, the data from the master SERDES is fed into the next stage. Depending on the actual data rate, the corresponding bits are taken from the SERDES output and put into a shift register. The shift register outputs a fixed format of 32 b data and a *valid* signal. So, the different data rates are represented in different rates of the *valid* signal being active.

This format was chosen for several reasons. The first one is that the LHC systems are based on a 40 MHz clock (the bunch crossing frequency) and a data rate of 1280 Mb/s would result in 32 b per 40 MHz clock cycle. The next reason is the interoperability between various ways to feed the data into the FPGA. The high speed transceivers in the desired FPGA class also have a default width of 32 b at the internal interface.

But the main reason is that the aurora protocol used 66 b frames with 2 b used for the header and 64 b for the data. As the later stages need whole frames at a defined alignment, a gearbox with a ratio of 32 : 33 is needed², making a width 32 b at the input a perfect fit. That gearbox is also the next stage, but slightly modified to output 66 b every second clock cycle.

²The gearbox takes 33 words with a width of 32 b as input and converts these into 32 words with a width of 33 b. A pause is also generated after 32 output words since it takes a clock cycle less to output the resulting words than are needed to input the initial words.

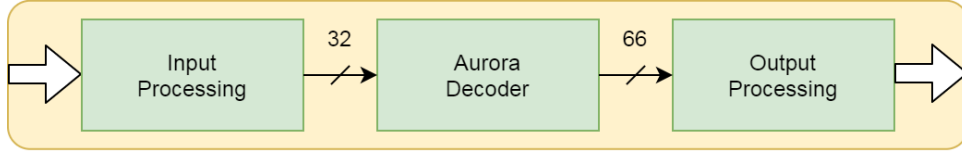


Figure 4.9: Block diagram of the very basic receive chain containing only the steps that are strictly required to be implemented in hardware. All other steps could be done in software later on.

The output of the gearbox ensures a stable alignment of the header within the 66 b word, but it still needs to be found and shifted to the uppermost two bits. The only way to find the header is to check if the two most significant bits are identical or not, as *10* and *01* are valid values for the header. If the two most significant bits have the same value, the input stream is shifted and the check starts again.

The shifting of the input stream is achieved in two steps. The gearbox has an internal counter to shift the data to the correct position and to signal the slot with no output frame. This counter can be used to shift the streams in multiples of 2 b as within two clock cycles, there are 64 b inserted and 66 b taken out. To achieve odd shifts, the shift register in the sampling stage is 33 b wide with the option to output the upper most 32 b or the lower one.

When 32 frames pass with both header bits not being identical, it is assumed that the right position is found and the stream is declared as *synchronized*. To complete the decoding of the aurora lane (the *aurora decoding*), the data part is feed into the descrambler.

With the aurora frames being aligned and descrambled, all processing steps that are required to be implemented in hardware are completed and the remaining processing can be done in the software. Even the frame alignment and descrambling could be realized in software, but would need a lot of computing power and memory operations. Therefore, it is highly recommended to use a hardware implementation for this.

At this point, all kind of *idle* frames (including synchronization and clock compensation frames) could be removed from the stream to reduce the amount of data to be transferred if only a single aurora lane is used. If the aurora link contains more than a single lane, all frames need to be kept and transmitted until the lane bonding is done. Otherwise, it would not be possible to correctly align the different lanes.

To enable further processing in software, the output of the aurora decoding needs to be connected to the network interface by a simple output processing stage taking care of building packets and bus width adaptation. Figure 4.9 shows a simplified view of how this would look like.

FE protocol decoding

The next step is the extraction of the event and register data out of the data stream. This is quite simple if the FE chip uses only a single lane, as there is no need to synchronize the different lanes to each other (called *lane bonding*). So, a FIFO is

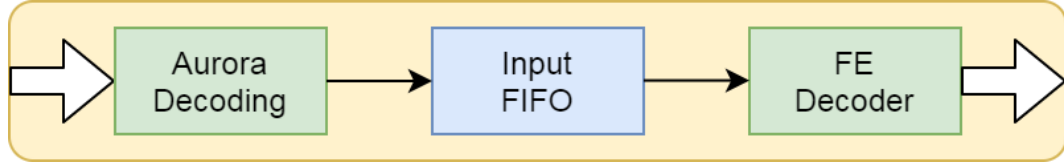


Figure 4.10: Block diagram of a simple version of the protocol decoder. Shown is the scenario with a single lane only.

added after the aurora decoding to buffer the incoming data in case the network interface is busy. Right behind the FIFO comes the FE decoder to interpret the aurora frames and extract the data. It strips the *idle* frames, extracts the register frames and separate events. Figure 4.10 shows how the system looks like in case of a single FE chip using a single aurora lane.

If more than a single lane is used, the lanes need to be aligned to each other. This is done with the help of a special kind of *idle* frames, which are inserted on all lanes at the same time and in regular intervals. The FE decoder needs to find the frames being sent at the same time and adjust the readout of each lane accordingly. Therefore, the decoder deploys counters measuring the time since the last alignment frame on each lane.

The offset between the single lanes should not be larger than a few frames in total as all data cables of the same chip should be routed as a bundle. It is more likely that the offset between the lanes is caused by slightly different sampling points, resulting in the gearboxes being in different phases of their cycle. When all lanes found at least a single alignment frame and all counters have its values in the lower half of the interval, it is assumed that the last alignment frames of all lanes belong together. The FIFO of each lane is read further until the next alignment frame was read from it. As soon as the readout of all FIFOs is paused, the channel bonding is done. Therefore, the whole channel bonding process should finish within two alignment frame intervals after all lanes reached a valid frame alignment. Afterwards, the frames are read in a *round robin* scheme (i.e. one from each FIFO and then starting again at the first FIFO). Figure 4.11 shows a simulation of the FE decoder.

This process requires that the lanes are connected in the correct order as the FE decoder is not able to determine the correct order itself. It is also not foreseen to shuffle the lanes within the FE decoder, so a false ordering of the lanes will result in the frames coming out in the wrong order³. Also, the delays between the lanes must not be higher than half the interval of the alignment frames, otherwise the alignment will be incorrect. If such cases are occurring, the interval might be increased to ease the automatic alignment.

The FE decoder can be used with any number of aurora lanes at any input port as long as the ordering is respected. The configuration can also be changed at runtime. As soon as the enabled lanes changes or an active lane loses synchronization, the FE

³This could be corrected in software again, but would result in a very complicated sorting.

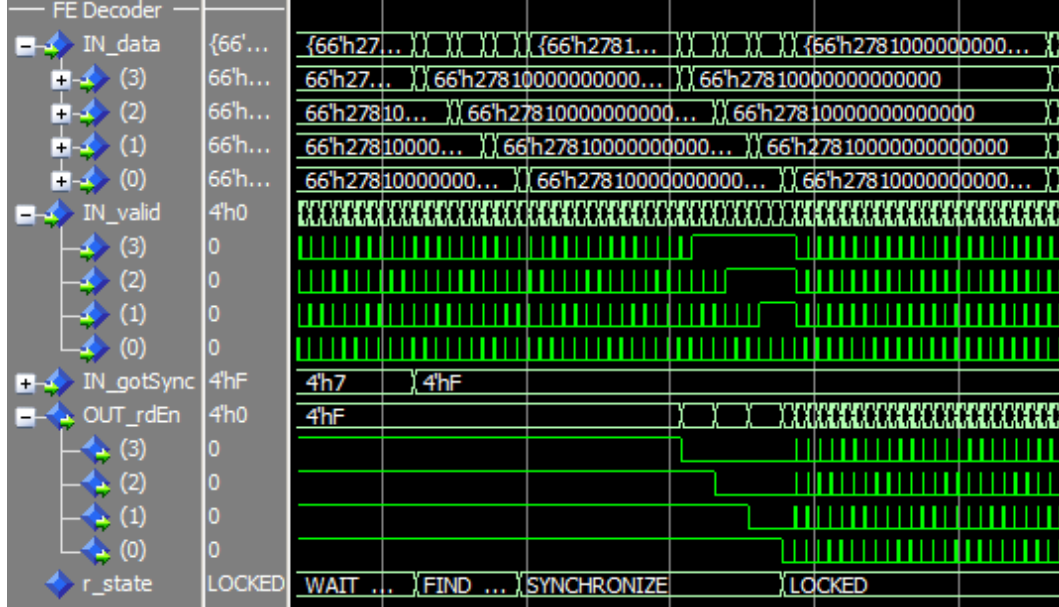


Figure 4.11: Simulation of the frame alignment process. Once all input stages signal a valid frame alignment (i.e. *gotSync* signal has the value `0xf`) and the FE decoder has found a synchronization frame on each lane, the internal state switches to *SYNCHRONIZE* and the readout of the FIFOs is stopped (i.e. the *rdEn* signal goes low) as soon as the next synchronization frame was found on the corresponding lane. When all lanes reached that state (i.e. *rdEn* signal has the value `0x0`), the lane bonding is completed, the internal state goes to *LOCKED* and the readout is started again by reading the frames one after another.

decoder will restart the alignment process to adapt for the new situation. Thereby, changes of the status of a not enabled lanes are ignored, so the FE decoder will only realign as soon as the enabled lanes change if a lane is added.

This can be used to build a runtime configurable system to be used in a changing environment like a laboratory setup. But it also helps in detector operation as only a single firmware version is needed to cover all possible connectivity variations. So, a failing FE interface unit can be replaced with a spare one by just plugging in the corresponding fibers and loading of the configuration of the connected detector segment.

To ease the scaling of the system, the final firmware will be built out of blocks serving a single optical fiber each. As the GBT protocol bundles up to six aurora streams into a single fiber, these blocks will have six aurora decoding stages and six FE decoder, with a multiplexer stage in between to shuffle the aurora streams as needed⁴.

⁴It might be that some FE chips will use all four lanes or the output of more than a single FE chip

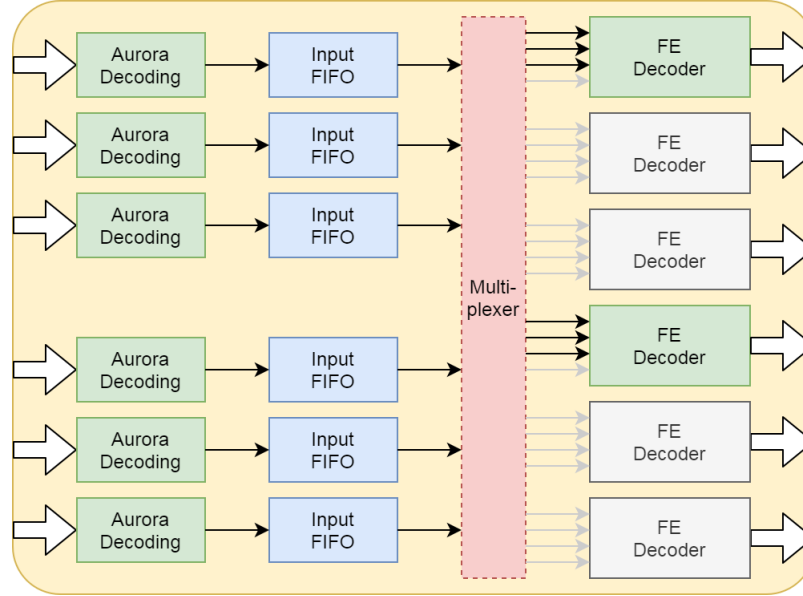


Figure 4.12: Block diagram of the final protocol decoder. Shown is a scenario with two active decoders serving three lanes each.

This multiplexer takes also care of the grouping of streams if the FE chips in the connected detector segment need more than a single aurora lane to get all the event information out in time. In that case, not all of the FE decoders would be used.

Figure 4.12 shows how the final firmware block would look like. In the shown scenario, the firmware block serves two FE chips with three active lanes each. The unused FE decoders are grayed out.

The current implementation of FREDDIE does neither use optical fibers nor the GBT protocol and the interface card groups four aurora lanes with a single command line per connector. With no modules containing more than a single RD53A chip being available, the design was simplified by connecting all four aurora lanes to the same FE decoder as shown in Figure 4.13.

Post processing

The FE decoder outputs the data part of the aurora frames in a fixed 64 b format. Specific signals are added if the current frame is a register frame or an end-of-event frame. The register and event data are still sharing the same data lines, so these need to be separated here. This is done by having separate FIFOs for register and event data. These FIFOs use already the bus format of the network interface for not slowing down the data transmission. This requires that the data width is scaled up accordingly before the FIFOs.

will be packed into a lane in the final detector. If this is the case, the number of decoders and their arrangement needs to be adjusted accordingly.

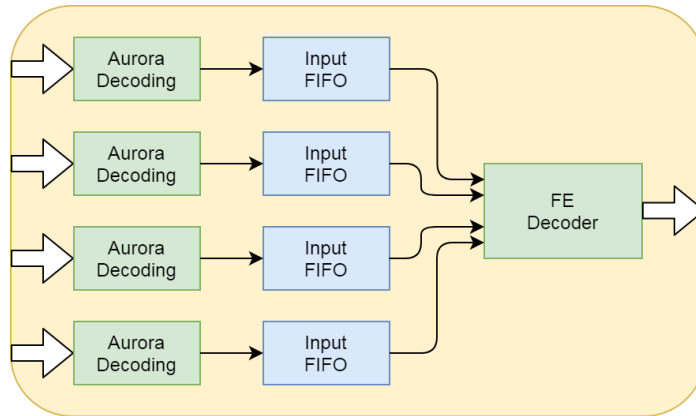


Figure 4.13: Block diagram of the protocol decoder being optimized for FREDDIE. As each connector contains four lanes and can only serve a single FE chip, four aurora decoders are combined with a single FE decoder.

Also, the data should be packed into packets of predefined size to improve the network performance. This is done by an additional firmware block in front of the data width adaption by setting the *tlast* flag accordingly. Besides the forming of the packets, this blocks also ensures that the packet is sent out if not enough data arrived within a given time period. Otherwise, the data would get lost at the end of a session.

The FIFOs are configured to work in packet mode (i.e. data is only available, if the complete packet is written to the FIFO) and are used to transfer the packets from the slower clock of the FE processing part to the much fast network clock region. Therefore, the output of the FIFOs form the packetized payload which is directly feed into the DROP wrapper.

4.3 Network interface

Overall architecture

The network stack was developed to have a way of getting larger pieces of data out of FPGA systems. Since this also includes stand alone systems, using a commercial network solution fits best into the existing infrastructure. At the same time, it also opens a way to configure the systems remotely.

The network stack was designed in a modular way, having a wrapper for each layer and a standardized bus interface between them. As the remaining network is ethernet based and is still using IPv4, the network stack was also focused on these. The wrappers of layers 2 and 3 of the OSI model (see Section 2.1) are directly housing the firmware blocks for both protocols. In contrast to this, all other protocols of these layers (like ARP and ICMP⁵) have separate wrappers.

⁵This is needed for ICMP echos, better known as *ping*.

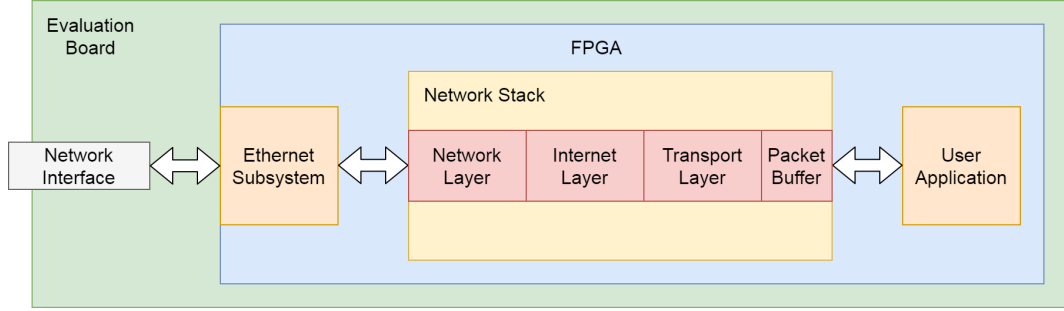


Figure 4.14: Position of network stack within the overall system. In addition to the network stack itself, access to a network also needs an ethernet subsystem as well as the appropriate connectors on the board.

Also, the network stack contains only parts of layer 2 as the Media Access Control (MAC) sublayer as defined in Ref. [26] is already included in a specialized firmware block called *Ethernet Subsystem*, which is provided by *Xilinx*. This block includes the line encoding, the checksum calculation as well as ensuring all the timing constraints. In addition to this, it also handles the lane synchronization in case of more than one lane is used. Only the logical frame itself (see Figure 2.2 on page 26) needs to be built or decoded by the network stack. Figure 4.14 gives a general overview where the network stack is located in the system.

If the newer protocol version IPv6 is required, the network stack should get also a new wrapper for layer 3. As this is increasing the size of the address fields, it requires also updated versions of some protocols (like ICMP) or introduces even new protocols to replace others (like ARP). The modular design of the network stack allows the easy replacement of the IPv4 version by this new one or the usage of both in parallel.

The parallel usage of IPv4 and IPv6 might require the duplication of the UDP wrapper as the width of the AXI interface would change. Currently, there is no multiplexer implemented which could combine the output of both wrappers.

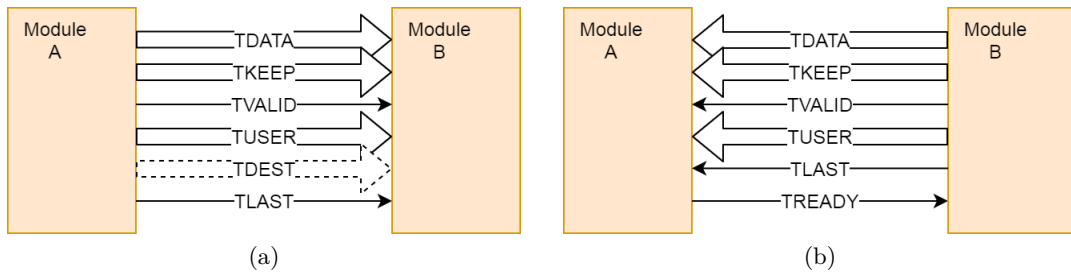


Figure 4.15: Used signals of the *AXI-Stream* interface. The width of *TDATA* selects the version of the network stack used. *TDEST* is only used in some parts. a) Receive path b) Transmit path

The network stack uses the *AXI-Stream* interface as already described at the beginning of this chapter. The combination of signals used by the different parts are shown in Figure 4.15 and described below in more details.

The receive path is not using the *TREADY* signal as the data coming in from the network cable cannot be stopped, so the processing must be able to process all data at line speed. The *TDEST* signal is used to forward the protocol identifier, so a demultiplexer can route the stream to the correct module. It would be possible to omit the demultiplexer and just forward the stream to all attached modules, but this would require each module to check if the stream is really meant for it.

The *TUSER* signal is used in the receive path mainly for the *goodFrame* flag which is sent in the last transfer (i.e. when *TLAST* is asserted) in case no error was found and the processing should proceed. This has to be in the last transfer as the ethernet checksum is calculated over all the frame data and the checksum itself is placed behind the payload. So there is no way to perform the error checking earlier. To filter faulty packets, there is a packet buffer placed between the last decoder and the application interface which deletes a packet if the *goodFrame* flag is missing. It is also the first point in the receive chain where the strict requirement of processing all incoming data is relaxed as the packets are stored in the buffer until the user application is able to get these. If a packet is not fitting into the buffer because it is running full, it is treated like if it is faulty and the *goodFrame* flag is missing. This filtering feature comes with a delay corresponding to the length of the packet as the user can only read the first data word after the last byte was written to the buffer.

The *TUSER* signal is also used to forward additional sideband information like the IPv4 address (source and destination⁶) which is needed by the protocols of layer 4 to build the pseudo header for the checksum. The user application might also want to get the source address in case it wants to send a reply.

On the transmit side, the bus is organized a bit different. Here, the *TREADY* signal must be used to stop the processing when the next processing step is still busy. A scenario where this is needed is the insertion of the header information in each encoder. As the necessary information to build the header can only be transferred with the valid signal being set, the data itself needs to be stored in a temporary register and the *TREADY* signal is deasserted. When the complete header is built, the data is taken from that register to fill up the bus transfer and the *TREADY* signal is set again. This interruption is going through the whole stack towards the user application as the encoders for the higher layers also need to stop when the ethernet encoder is inserting its header. To hide these interruptions from the user, there is also a packet buffer on the transmit side. So the user application can write a packet to this buffer and does not notice these interruptions. In contrast to the buffer on the receive side, the buffer here is deasserting the *TREADY* signal in case it is running full. This is the only way in which the user application will notice that it was writing faster than the network could handle the data.

⁶The network stack might be extended to support several local IPv4 addresses. That is why it is important to forward the actual address used for the packet.

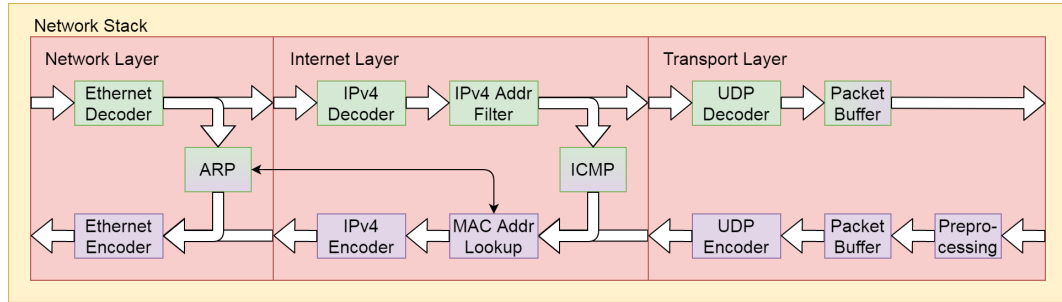


Figure 4.16: Internal structure of the network stack. The colors of the modules are depicting the type of interface used. Green stands for the receiving path variant, purple for the transmit path one.

The packet buffer also helps to get some information for the packet header which would otherwise be hard to get at this stage. For example, the UDP protocol needs the total length and the checksum across the data to build the header. But the stack itself can get the information only when the last byte is submitted to it. So the packet buffer is able to store such information with the last byte and sends it out again with the first transfer. It is important to note, that this feature comes with a delay corresponding to the length of the packet.

As both directions are using special signals, the modules used in the network stack need to be specific to one direction. This is especially true for the modules splitting or combining streams. Figure 4.16 gives a general overview of the network stack.

The initial version of the network stack was targeting a transmission rate of 10 Gb/s. Having a bus width of 64 b, a design clock frequency of 156.25 MHz was required. There was also a 8 b version implemented which needs much less resources and is intended for slower interfaces (e.g. 100 Mb/s and 1000 Mb/s). Later on, there was a generic version with a bus width of 256 b or greater added. This is possible as the headers of all currently supported protocols are smaller than 256 b, so that the logical structure is not changing with higher bus widths. But the chosen architecture fits better to smaller sizes as the resources needed for operations like the stream realignment and checksum calculations are growing drastically with increasing bus width. Also, all the registers used for buffering the output of each module are growing to a non-negligible amount. Another reason against the usage of a single bus with bus widths greater than 256 b is that the bus width is already on the order of the minimal packet length and overhead for bus operations can get up to 50 % in case of small blocks with a size of 1 B larger than the bus width. This requires to double the bandwidth of the bus to cope with all requested transfers.

All version can work with smaller transfer rates too. On the receive path, an arbitrary number of pause cycles (i.e. bus cycles with the valid signal being deasserted) can be added between two actual transfers. The transmit path uses the *TREADY* signal which can be used to throttle down the stack to a transfer rate the network can handle.

Implementation used by FREDDIE

The FPGA board which is used for FREDDIE offers four slots with a data rate of 10 Gb/s each which can be combined to a single 40 Gb/s ethernet interface. The corresponding *Ethernet Subsystem* uses a 256 b bus interface for each direction running at 312.5 MHz. To avoid additional conversion between the ethernet subsystem and the network stack, the latter one is also using these parameters.

But these parameters have already a rather limiting influence on the current architecture. For example, the checksum calculation of a 256 b word is not possible within a single clock cycle and needs to be split up. The UDP decoder and the preprocessing module in the transmit path are doing this by splitting the bus into 64 b segments and calculating a temporary sum for each. These temporary sums are then combined⁷ in a second clock cycle while the temporary sums for the next data word are calculated. In this way, these modules are still able to cope with the full data rate.

Another issue is the resource usage which increases drastically with the given bus width and clock frequency. For example, all output signals need to be buffered to meet the timing constraints. This sums up to 12.7% (receive path) and 14.3% (transmit path) of the flip flops (FFs) required for the total network stack alone for the *TDATA* signals used by UDP packets.

There are also additional buffers needed at the input stage of each module on the transmit path to prevent data loss. When the subsequent module is clearing its *TREADY* signal to indicate that it is not ready, a module needs to stop the stream by pausing the internal state machine and clearing its own *TREADY* signal. But the forwarding of the cleared *TREADY* signal takes a clock cycle in which the preceding module might transfer still send data (it still gets *I'm ready!* signaled). As the module is unable to update its own output register, it has to store the incoming data somewhere, requiring in total additional registers in the same order as the output registers. Figure 4.17 shows the structure and logic used for this.

⁷together with the result from the already processed part of the packet

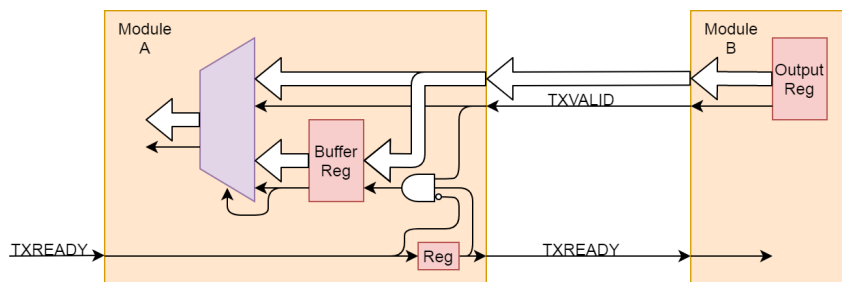


Figure 4.17: Handshake infrastructure in the transmit path for pausing transfers. A buffer register is needed in case a valid transfer is ongoing, while the module needs to stop processing and is therefore unable to handle the incoming data.

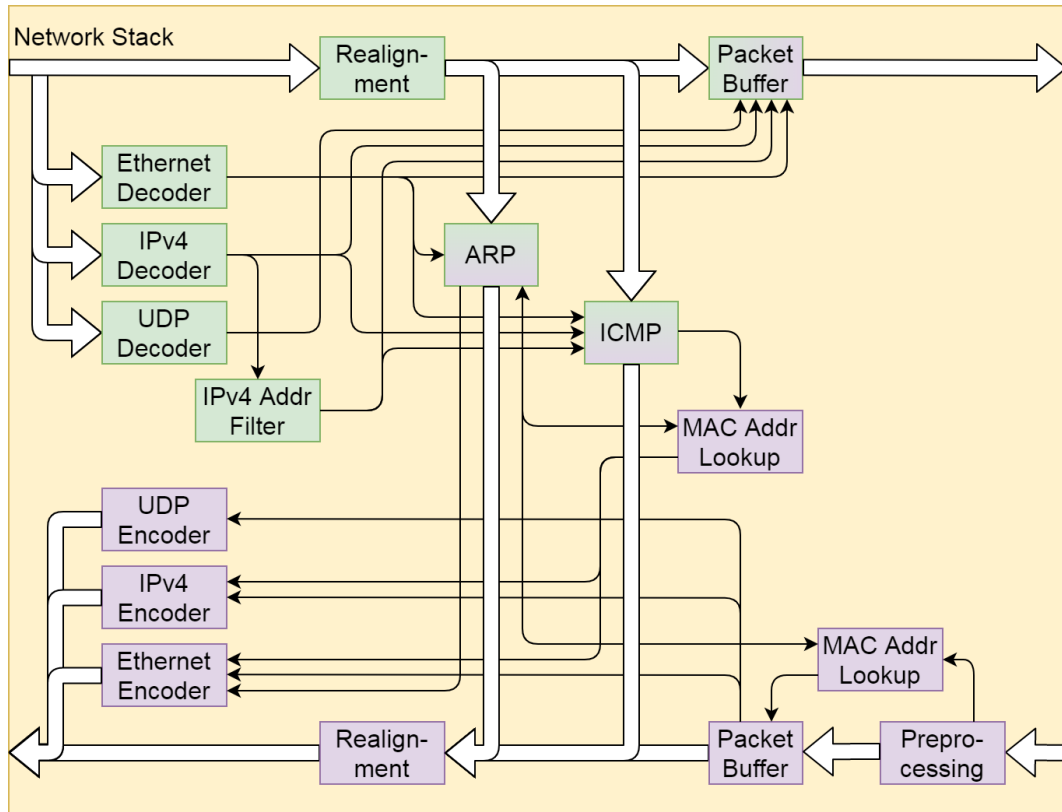


Figure 4.18: Structure of an alternative network stack architecture. All encoders and decoders are working in parallel instead of processing each layer one after each other. This should reduce both latency as well as resource usage.

This structure is working fine for small bus widths, but is responsible for an increasing part of the total resource usage. In the chosen architecture of the network stack, the only alternative to that would be to remove any need to pause the modules inside the network stack and pause all modules at the same time in case of the ethernet subsystem is clearing its *TREADY* signal. But this would generate a very high fanout for this signal and would increase the resource usage for smaller bus widths. So, the better way would be to reduce the number of modules on the data path, but this requires a change of the overall architecture.

Proposed improvements

The 40 Gb/s *Ethernet Subsystem* also offers a so-called *straggled* interface which is splitting the bus into several segments (here, two 64 b segments are used). Thereby, a new packet can start at each segment limiting the number of empty bytes at the end of a packet to one subsection, thus reducing the overhead and avoiding the need to run with double bandwidth. The MAC modules from *Xilinx* for faster ethernet standards

have only the segmented interface as the bus width is growing further. For example, the 100 Gb/s MAC module for *Ultrascale Plus* FPGAs only offers an interface built of four 128 b segments. For some of these modules, an interface conversion module was added later on to allow further usage of existing implementations. But increasing the bus width to values above of 512 b brings the problem that two packets may occur in the same bus cycle, which will require a segmented interface. This would also require to have two sets of decoders running in parallel.

As the straggled interface would require some drastic changes within the network stack, it could also be seen as an opportunity to change the overall architecture to address downsides of the current one. In the current implementation, only UDP is used to transfer data, so the modular approach is not needed and a version tailored especially on this task could be implemented more efficiently. For example, the realignment steps after each decoder could be omitted as the implemented feature set only supports fixed size headers for ethernet and IPv4. Therefore, the decoders could be placed at the corresponding offset with only a single realignment step following the UDP decoder. Also, starting all decoders in parallel preemptively could save both time and logic resources. If the output of a decoder will be used or not is then decided by the results of the preceding decoder. So, the identifier of the embedded protocol is only used to generate an enable signal if the data can be used or not (like the *goodframe* signal). This requires that all packets can be decoded independent of any other information like preceding packets as the preemptive decoding would have unwanted side effects otherwise. That requirement is fulfilled as all implemented features are stateless.

Figure 4.18 gives an overview of the alternative architecture. It should be noted that there are multiplexers needed in front of the decoders as the section of the data bus to be decoded varies depending on which segment the new packet arrives. The same is also true on the transmitting side where the header needs to be shifted to the correct position.

4.4 Configuration path

To make use of the FPGA, it needs a way to sent commands and read back status information. As the whole design is based on the network interface, it makes sense to use the network also for the configuration⁸. So, a way to set and read registers values via network is needed.

The configuration of the firmware is implemented by groups of registers being accessible via the *AXI-Lite* protocol, which is a subset of the *AXI* protocol. As the network interface uses the *AXI-Stream* protocol, which has no component like a memory address due to its optimization for streams, a path from the network interface to the *AXI* system is required. *Xilinx* already provides a lot of firmware modules to ease the integration of *AXI* systems, including a converter for this case.

⁸In this context, configuration means to set specific register values for different firmware blocks, not to configure the FPGA itself (i.e. loading the firmware into the chip).

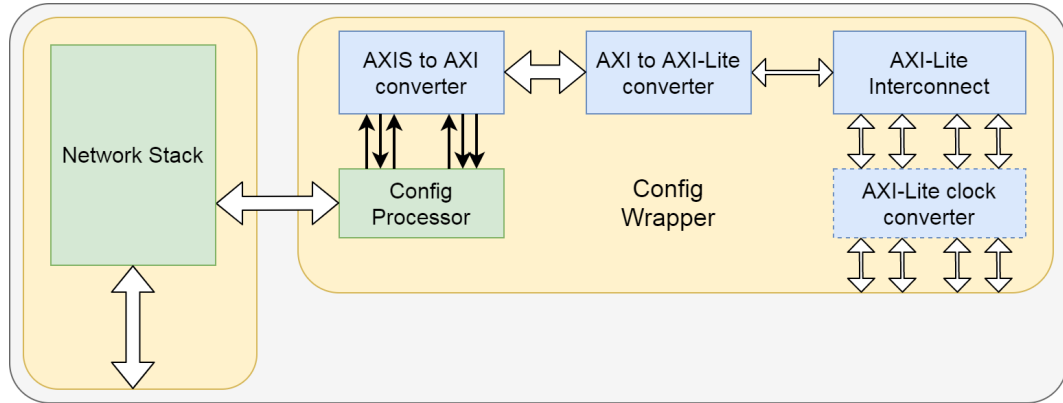


Figure 4.19: Structure of the configuration bus being extended by a remote option. It is now possible to access the configuration via the network stack within the FPGA in addition to the path through the processing system.

But this is more complex compared to the other *AXI* modules and acts more like a direct memory access (DMA) controller, copying data between the stream interface and the memory locations. So it needs an extra module to tell the converter, how many data words should be written to which address. Figure 4.19 shows how the system looks like.

Following the general modularity of FREDDIE, the configuration system has different endpoints for different function blocks as these are encapsulated within their own wrappers and should be independently useable. These are the four areas: *IO wrapper*, *data generators*, *FE processing* and the custom *DROP protocol*.

However, the required independence has the negative effect, that some blocks with only a few configuration values needed (or even a single one) require their own set of registers. As this would drastically increase the complexity, such blocks do not have these endpoints and thus have their configuration being hard-coded. One of these blocks is the *configuration wrapper* itself, which needs a port number at which it can be reached over the network.

Another example is the network stack, which has three values to be set. Two of them (i.e. the MAC and IPv4 addresses) are difficult to deal with if changed at runtime, as the network stack is unable to stop the other blocks from sending data. Even putting the network stack in reset for some time could result in undefined behavior of the whole system. So these two values can only be changed by flashing a complete new firmware into the FPGA.

4.5 Hardware implementation of the DROP protocol

The DROP protocol is implemented as a separate wrapper being located between the data generators and the FE processing on one side and the transmit path of the network stack on the other side (See Figure 4.20). The DROP wrapper was not

directly placed before the network stack since the design configuration part and the loopback connection should not use it.

The DROP implementation introduces the usage of *data streams* for all connected parts of the FPGA. Therefore, these parts send just the stream number among the data chunks instead of the network addresses (i.e. IPv4 destination address and UDP source and destination port numbers). The IPv4 source address is given by the IPv4 address of the network stack. The network addresses for each stream are configured centrally in the DROP wrapper.

Besides having all addresses managed in a central location, this solution also simplifies the merging of data from different sources into a single stream. This was used by the network characterization tests to combine streams from multiple data generators in order to achieve higher data rates. But this feature can also be used to fine tune the data rate by merging a fast and a slow data stream as the data rate of the latter one has much smaller step sizes as the former one.

While merging streams works fine for network tests, the merging of event data would require additional packaging as the data from the FE chips does not include any identifier of the chip. Therefore, the processing software would not be able to separate the data of multiple streams again.

In the current implementation, the DROP encoder generates only the default header. It has a separate counter for each stream to generate the packet identifier, which can also be read via the configuration path. With no protocol option being implemented (none of the proposed ones were needed), the flag field is set to a constant value. For testing reasons, this is currently a non-zero value. This must be reverted to zero before any extension is implemented!

However, a few precautions have already been taken in the event that some options are needed (See Section 2.5 for more details). The options can be sorted into two groups depending on if these need bidirectional communication or not.

Most options only augment the data stream with some additional information for

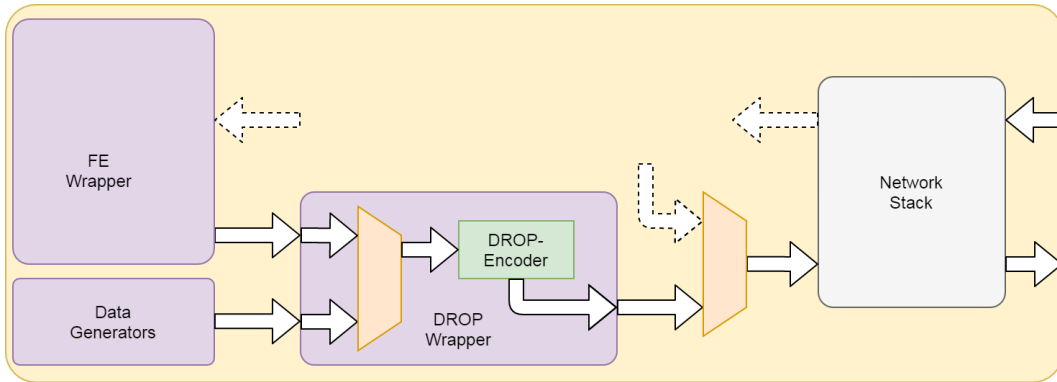


Figure 4.20: Position of the DROP wrapper within the FPGA design. It is located between the data generators and the FE processing on one side and the transmit path of the network stack on the other side.

the receiver and therefore do not require a bidirectional link⁹. These would just be integrated into the DROP encoder and be enabled for certain streams via the configuration block. If the functionality could also be used at different points, it could also be inserted as a separate block before the encoder. A good example for such an option is the implementation an FEC mechanism.

Options needing a bidirectional DROP link are more complex to be implemented as there are two prerequisites. The first one is to determine the stream to which a request belongs to. While the DROP encoder uses the internal stream number to manage the stream and their metadata, this number is not available for the receiver of the streams. Therefore, the receiver can only identify the stream by the quadruple of IPv4 and UDP addresses and respond by swapping source and destination addresses. So, a DROP decoder on the generating side (i.e. within the FPGA) needs to reconstruct the internal stream number before the request can be processed. This also functions as a safety check to reduce the probability of a false assignment.

The second part is the bus demultiplexer modules not being able to split off an address range. With the current implementation, these demultiplexers would need a separate bus segment for each UDP port used by DROP and a multiplexer afterwards to combine these again for the solitary DROP decoder. Due to the large bus width, this solution would result in a massive increase of FPGA resources to be used. Therefore, it would be easier to include a range check in the demultiplexer. This could also be used to help with the reconstruction of the internal stream number as the demultiplexer is checking the port number anyhow.

For the implementation of a retransmit extension, a DROP decoder would process the retransmit requests. It triggers the reading of the missing packets from the memory and their insertion into the data stream (with the original packet identifier) for each retransmit request received. In addition to this, the decoder would also mark all not requested packets (until the last one seen) for deletion from the memory to free up space for new packets. The latter part could also be implemented as a barrier to stop the DROP encoder as soon as it tries to overwrite a still not confirmed packet. With 2^{24} (approx. 16.8 million) available packet identifiers per stream, the system should have enough time to ensure the correct transmission of all packets until the identifier needs to be used again. It seems more likely that the amount of memory being available per stream (and therefore the number of packets to be stored) sets the tighter limit. To function correctly, the DROP encoder has to send a copy of the outgoing packets (including the DROP header) to the memory if the feature is enabled for the particular stream. The address within the memory is thereby given by the packet identifier and the maximum packet size.

The DROP decoder can also compare the last generated packet identifier with the last seen one from the receiver to check if packets got lost at the end of the transmission. Thereby, it should estimate the RTT to calculate an appropriate delay before trying to sent these again.

⁹At least not a bidirectional DROP link. Nevertheless, a way to configure the DROP encoder and the supported options (e.g. enabling these) is required.

4.6 Data generators

A well defined data stream is needed to reduce the number of influencing parameters while characterizing the network behavior. While the FE chips can generate a defined number of hits, some features like the insertion of register frames are complicating the results. Also, there are simply not enough FE chips available for this thesis to fill the network bandwidth. Therefore, FREDDIE got a set of data generators for generating well defined streams.

The most important parameters of the data generator are the clock frequency of 160 MHz¹⁰ and the width of the data signal, which is set to 64 b. Thereby, the data generators have a theoretical bandwidth of 10.24 Gb/s. This theoretical value is slightly reduced by some implementation specific properties, which are:

- Between two packets, a pause of at least one clock cycle is needed to reinitialize the generator.
- If the packet length is not a multiple of 8 B (i.e. the last data word is not fully filled), it is filled up with padding.
- The following firmware module pauses the data generation at the start for an additional clock cycle.

The last point is due to the way the transmission side of the network stack is implemented and therefore common to all modules using this scheme.

In addition to these restrictions on the maximum achievable data rate, the generated data rate can be further reduced by increasing the pauses in a targeted manner. This can be done by introducing pause cycles between two data words or as a bigger block after each packet. The former version is independent of the packet size and allows for a bigger range of possible rates, but has the downside of being rather coarse for small pause values as a value of 1 will already cut the rate in half¹¹. The latter one adds an absolute pause value and therefore results in different slowing factors for different packet sizes. So, this version can be used to also generate data rates above of 50 % of the maximum data rate. In both cases, the length of the pause is given in number of whole clock cycles as valid bus transfers must have all bytes filled for all cycles except the last one.

As the insertion of pause cycles between single data words has not the granularity needed for the network tests, only the second variant is currently used. So, the data

¹⁰They use the same clock as the FE processing part to make their output comparable to the ones from the FE decoders with respect to the timing.

¹¹The introduction of fractional values would be beneficial here. For example, adding a pause cycle only after n out of m data words could solve this problem. But this was not implemented.

rate is described by the formula:

$$\begin{aligned}
\text{data rate} &= \frac{\text{packet length}}{\text{clock cycles needed for generation}} \cdot \text{clock frequency} \\
&= \frac{\text{packet length}}{\left\lceil \frac{\text{packet length}}{8 \text{ B/clock cycle}} \right\rceil + \text{pause cycles} + 1} \cdot \text{clock frequency} \\
&= \frac{(8 \cdot (n - 1) + x) \text{ B}}{\left\lceil \frac{(8 \cdot (n - 1) + x) \text{ B}}{8 \text{ B/clock cycle}} \right\rceil + p + 1} \cdot 160 \text{ MHz} \\
&= \frac{(8 \cdot (n - 1) + x) \text{ B}}{n + p + 1} \cdot 160 \text{ MHz} \\
&= \frac{(8 \cdot (n - 1) + x)}{n + p + 1} \cdot 1280 \text{ Mb/s}
\end{aligned} \tag{4.1}$$

n = number of clock cycles per packet, ≥ 1
 p = number of pause cycles, ≥ 1
 x = remaining bytes in last word of a packet, $\in [1, 8]$

Connected to the data rate is the packet rate, which is the data rate divided by the packet length. The rate at which packets are generated and sent out is important as each packet needs a set of network headers and corresponding processing when received in software.

The packet rate is described by the following formula:

$$\begin{aligned}
\text{packet rate} &= \frac{\text{data rate}}{\text{packet length}} \\
&= \frac{(8 \cdot (n - 1) + x) \text{ B}}{n + p + 1} \cdot 160 \text{ MHz} \cdot \frac{1}{(8 \cdot (n - 1) + x) \text{ B}} \\
&= \frac{160}{n + p + 1} \cdot \text{MHz}
\end{aligned} \tag{4.2}$$

As the minimal values for n and p are 1, a maximum packet rate of 53.3 MHz can be reached. It should be noted here, that ethernet has a minimal packet size of 64 B plus additional framing elements (e.g. the so-called *interpacket gap* with an equivalent length of 12 B). So, to reach a packet rate of 53.3 MHz on an ethernet link, an corresponding transfer rate is needed.

The ratio of the actual data rate compared to the theoretical maximum can be obtained by the status output of the generators as these count both the total clock cycles running as well as the clock cycles with an active data transmission. It should be noted that this might have a non negligible error as an active transfer does not need to have all bytes enabled. This can get a problem especially for combinations of small packet sizes and packet sizes which are not a multiple of 8.

Increment within a data word						Increment with each data word						Combination of both options					
0	0	0	0	0	0	0	1	2	3	4	5	0	4	8	12	16	20
1	1	1	1	1	1	0	1	2	3	4	5	1	5	9	13	17	21
2	2	2	2	2	2	0	1	2	3	4	5	2	6	10	14	18	22
3	3	3	3	3	3	0	1	2	3	4	5	3	7	11	15	19	23

Figure 4.21: Overview of the different data pattern that can be generated. The time flows from left to right, so a column represents the four numbers to be generate in the same clock cycle while the different columns are the changes in time.

The data generators are able to generate different patterns to test different scenarios. The current implementation is based on a 16 b counter¹² with different incrementation modes. The options are:

- incrementation within a data word
- incrementation with each data word
- reset counter at the start of each packet

These are also depicted in Figure 4.21. The mostly used mode is to use both incrementation options and not reset with each packet.

The generation of the identical packet data can be mitigated by having a packet size being coprime to the number of combinations of the counter (i.e. 65536). So, having a packet length of 1998 B or 2046 B (i.e. 999 and 1023 counter values per packet respectively) would result in the generation of the same packets after 999 and 1023 full cycles (i.e. after roughly 125 MB of data being generated). Even with a packet size of 2000 B, there will be 125 full cycles (i.e. 16 MB) needed before the packets start to repeat them self.

Once enabled, the data generator is sampling the configuration and starts the data generation. It will run until a given number of bytes was generated and will restarted immediately if the enable flag is set in that clock cycle. Changes in the configuration get effective only at such a restart to prevent unexpected behavior. The length of such a block of data is given via a 64 b variable, so any realistic value is possible. Thereby, the last packet being generated will be smaller as the specified packet length when the length of such a block is not an integer multiple of the packet size. The effect of this on the average data rate should be negligible as long as the block size is much larger than the packet size.

¹²To be more precise: these are four counters in parallel to fit the bus width.

Chapter 5

Data reception

Sending out data is an easy task compared to receiving it as the transmitter is the active part having all information at hand before a packet starts. The receiver needs to recover all of this information, even from a potentially flawed transmission. This begins already with the very basic question of when does a new packet start. While the transmitter can start the transmission at any time¹, the receiver has only a minimal pause until it needs to be ready for the next packet, independent of the processing state of the previous packet. Therefore, this simple question directly leads to an important point in packet processing: How many packets will come within a given period and is the system able to handle them?

While the packet rate defines how many packet headers must be processed, this is not the only parameter that needs to be considered as it is not enough to receive the packets. The data contained in the packets also needs to be processed. So, the packet size is also an important parameter, extending the packet rate towards the data rate.

As both the maximum possible packet rate and data rate depend on the hardware components in the system, the used system is described first in this chapter, followed by an overview of the traditional processing path used within the Linux kernel. Afterwards, alternative approaches including a new technique called *eXpress Data Path* (*XDP*) are presented, before the actual user space program for the data processing is explained.

This chapter concentrates on the latest version of the receiving software and the system it is running on. Intermediate steps are only covered if these are needed to understand certain design choices. Chapter 6 will describe some additional setups as they were used for tests during the development phase.

5.1 System architecture

As the tests done in this thesis should give some estimates on what the chosen server is able to process, the system and CPU architecture are considered in the software development. Therefore, the software is not a generic solution that will deliver the same performance on any other CPU architecture, making it only partly portable.

¹This also means, that it can take all time needed to prepare the packet.

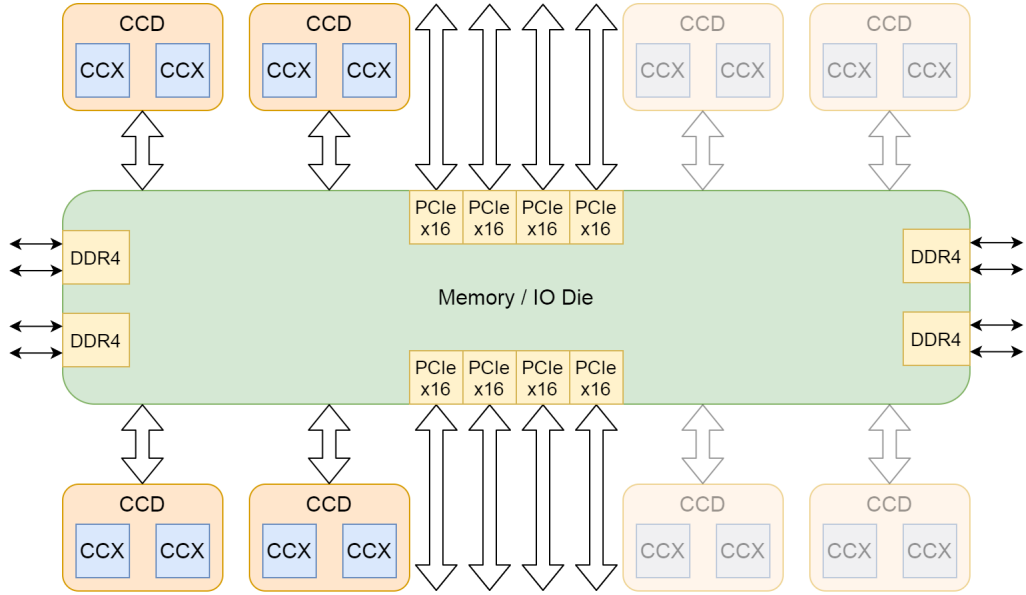


Figure 5.1: Block diagram of the *AMD EPYC 7302* processor. Only four out of eight core complex dies (CCDs) are populated. Based on Ref. [44].

The server used for the tests is a *Gigabyte R282-Z93 (rev. 100)* rack server [41] being populated with two *AMD EPYC 7302* processors [42], 128 GB of *DDR4-3200* memory and a *Mellanox ConnectX-5* 100 Gb/s network interface card (NIC) [43]. The operating system used is *Fedora* in version 30, with the Linux kernel being of version 5.2.9.

The AMD EPYC server processors of the second generation are built up of a central *Memory/IO* die with up to eight core complex dies (CCDs) connected to it, see Figure 5.1. The *Memory/IO* die has in total 128 PCIe lanes of the fourth generation with half of them being used to connect both processors in a dual processor setup. Therefore, a server with two AMD EPYC processors has also 128 PCIe lanes available to connect peripheral devices. The PCIe interface is organized as eight controllers with 16 lanes each, which can split up further to form several smaller interfaces (e.g. SSDs with a so-called *M.2* interface only support up to four lanes).

The *Memory/IO* die also houses four *DDR4* memory controllers which have two memory channels each. All memory controllers can be accessed from all compute cores of the same processor with the same latency, so the whole processor is a single Non-Uniform Memory Access (NUMA) node. This means that the performance of a software is independent of both the core it is running on and the memory location where the data is stored, as long as both belong to the same processor. The connection between both processors in the server adds additional latency, so accessing the memory on the other processor takes longer and therefore reduces the performance. But the operating system should be aware of this and locate the memory objects needed by a piece of software in the corresponding memory areas.

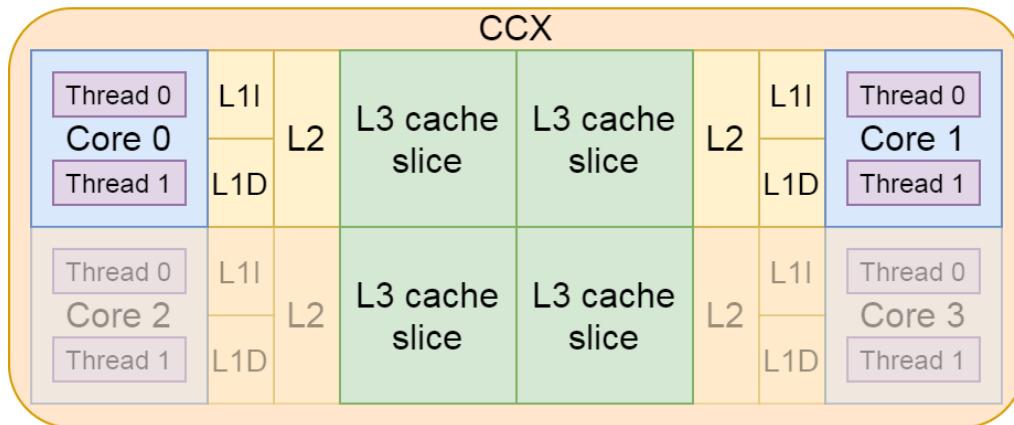


Figure 5.2: Block diagram of a CCX of the *AMD EPYC 7302* processor. Only two out of four cores are enabled, but all four L3 cache slices are available. Based on Ref. [44].

A CCD contains two core complexes (CCXs) with each CCX being built up of four computing cores, see Figure 5.2. While the four cores of a CCX can communicate with each other without involving the *Memory/IO* die, the communication between the two CCXs of the same CCD has to take the route via the *Memory/IO* die, including its latency and reducing the bandwidth for memory transactions.

The communication within a CCX is done via the so-called *third level cache* (short: L3 cache), which is shared between all cores of a CCX. It is build as a *victim* cache, meaning it holds the cache entries which where displaced in the first two cache levels, but it also includes entries needed by two or more cores.

The cores support simultaneous multithreading (SMT), meaning that two threads can be executed in parallel on a single core. But these threads share the execution as well as load/store units, even if the number of these units is already higher than for processors without SMT support. It might get to situations where a single thread would run faster (has less execution time or higher throughput) without a second thread running on the same core.

A similar restriction applies to the cache of a CCX and its connection to the *Memory/IO* die as the four cores can write with a higher data rate to the cache than the rate available to the cache for communication with the *Memory/IO* die. The data transfers between the CCX and *Memory/IO* die use an internal clock called *FClk* with the same frequency as the memory interface² and allows for 32 B being read and 16 B being written in each clock cycle (both from the CCX’s point of view). This corresponds to reading from two and writing to one memory module at the same time, or 409.6 Gb/s and 204.8 Gb/s respectively when used with *DDR4-3200* memory.

²The *Fclk* can be regarded as a copy of the memory clock, but is divided by two if the frequency of the memory clock is set to high. As these frequencies are not reached in the setup used, this detail can be omitted.

In principal this should be enough to forward the complete network stream to a single CCX. However, the connection between the CCX and *Memory/IO* die will have additional traffic which can easily fill up the whole bandwidth. A single processor core can fill the link completely, especially if the cache lines foreseen to hold the new data of a load operation contains modified data that needs to be written back before accepting new data.

Even if there are enough CCXs to split the load, it is desirable to hold all data locally in the cache. This is not only to reduce the bandwidth used on the connections between the CCXs and the memory die, but also for reducing the latency. A read operation from the DDR4 memory takes much longer than from the L3 cache as there are more components involved. A reduced latency will reduce the time spent waiting for the data and therefore increase the amount of data being processed at the same time. Therefore, the cache size of 17 MB in total³ will set a limit for the memory usage of the receiving software.

5.2 Network processing in Linux

Network processing in Linux is a large and complex field with a lot of processing steps that can be tuned. As the type of workload can differ in a wide range, the default Linux kernel is configured in a way that most applications will do fine. But this is always a compromise. A specific application will not perform as good as it could with optimal settings. The optimal parameters for each processing step need to be found.

To be able to tune the Linux network system, two main requirements need to be fulfilled:

- The characteristics of the network traffic to be processed need to be known and these should also be stable in time (at least for a given period).
- The processing steps and their parameters need to be known. In particular their possible configurations and limitations.

To meet the first requirement, some precautions have been taken on the sending side in the FPGA. There are firmware blocks collecting the data until the packets are filled or a timeout occurs as well as data generators to generate a well defined data stream for network testing (See Chapter 4).

The second requirement is the bigger issue because of the complexity of the network system within the Linux kernel. The network system within the Linux kernel serves minimalistic single board computers as well as high performance servers hosting several virtual machines each, which may also span virtual networks to separate the network traffic. Also, different manufacturers of computer components chose different ways to implement these, requiring even more complexity within the Linux kernel.

³Even if only two cores are enabled, all four L3 slices with 4 MB each are available. In addition to this come the L1+L2 caches with 512 kB per core.

Other reasons are the increasing performance of networks and computer hardware, but also an increased sensitivity for usability and security issues.

As the thesis makes use of a 100 Gb/s ethernet-based network, it will focus on such multi-gigabit networks. It will focus also on the receiving side as the data transmission done in software within the thesis is far off from being at a critical point with respect to the network processing.

Data reception within the Linux network stack

The first steps needed to receive a packet are already done before the packet itself arrives at the NIC. Besides loading of the according driver, the Linux kernel has to create and initialize several data structures in the host memory. Most important here are the ring buffers for the receive queues, where the received packets are stored in. The Linux kernel has to configure also the receive filtering, which is used to assign the packet to a certain receive queue within the NIC's network processor. An example of such a processor can be found in Ref. [45]. When all this is set up, the system is ready to receive packets.

The network packets enter a computer via the corresponding connector on the NIC⁴. Depending on the supported data rates, these are RJ45 (mainly used up to 1 Gb/s, but there are also variants up to 10 Gb/s), small form-factor pluggable plus (SFP+) (for 10 Gb/s), quad small form-factor pluggable plus (QSFP+) connectors (for four times 10 Gb/s or a combined 40 Gb/s link) or their counterparts with the extension 28 instead of the *plus* with support of 25 Gb/s per lane [46].

On the NIC, the network processor starts the processing by verifying the cyclic redundancy check (CRC) checksum. Afterwards, the receive filtering checks whether the destination address of the network packet matches the configured addresses of the system. If this is the case, the layer 3 and 4 checksums are calculated (if supported by the network processor), the designated receive queue is determined and the packet is forwarded to the internal receive FIFO.

When the entire packet (including additional information like which checksum was calculated) is in the FIFO, the DMA controller copies the packet into the corresponding ring buffer prepared by the Linux kernel. The system is not directly notified about the new packet as this would result in too much overhead. Therefore, the NIC waits until a given number of packets has arrived or a timeout ended before raising an hardware interrupt. This interrupt starts the processing by the driver and thereby the first invocation of software.

The kernel applies a mechanism is called *NAPI* (short for *New API*) which is regarded as a good compromise between interrupt and polling based processing [47]. While the interrupt based processing is more efficient for low packet rates, it generates a lot of overhead under high loads. The polling works fine under high load (i.e. when most calls of the poll function returns work to be done), but either introduces high delays (when the poll frequency is too low) or a lot of unneeded CPU utilization

⁴Wireless networks are ignored here as this thesis focus on ethernet based networks.

(when too many poll requests return with no packets to be processed) otherwise. Therefore, these were mixed to utilize the benefits of both.

The driver involved by the hardware interrupts disables these for this NIC and reports to the Linux kernel that there are packets to be processed. A poll loop is started (in case none is currently running), querying all network interfaces that reported work to be done. This poll loop pulls all packets up to a configured number from the receive ring buffers and forwards them to the network layers for processing. When a network interface has more packets than the configured maximum, the interface is inserted again at the end of the poll list. All packets arriving in the meantime are added to the list of available packets to be pulled. If there is no space left in the ring buffer when a packet arrives, the packet is dropped and is therefore lost. Only if a call pulls all remaining packets of an interface, the hardware interrupts for this device are enabled again.

The packets and their meta data are forwarded and processed by the different protocol layers as a *socket buffer (SKB)* which is most fundamental data structure in the Linux network stack. These SKBs include not only pointers to the packet and its length but also information about the identified protocol. This includes a pointer to the header of it within the packet. In addition, the SKBs provide information on the socket and the associated network device. Within the protocol processing, the addresses of the corresponding layers are checked and corresponding actions taken. This includes the routing of packets and ICMP notifications in case no application had opened the addressed port yet.

If the packet was intended for the system and an application registered a network socket for the addressed port, the packet is added to the receive buffer of that socket. As soon as the application calls the receive functions, these packets are copied into the buffer provided by the call.

So, the packet gets copied several times in the receiving process. This is a result of the various parts of the processing chain being part of conceptually different blocks (i.e. the driver, the Linux kernel and the user application) with different system access rights. To remove the number of copy cycles, a change in the architecture would be needed to provide the final memory location as early as possible.

More details about the network processing in Linux can be found in Ref. [48].

Userspace solutions

There are several efforts to get around the Linux network stack as its usage comes with additional overhead. One way to completely bypass the Linux kernel with respect to networking is moving the networking processing into so-called *userspace* programs. This means, that these programs have to integrate the drivers for the network hardware.

This is seen as an opportunity of improving the interactions between user program and the NIC to reach higher performance. The Data Plane Development Kit (DPDK) framework for example exchanges the interrupt based approach of the Linux network stack by specialized polling drivers [49]. It also provides optimized libraries for packet

processing and a so-called *Environment Abstraction Layer (EAL)* to abstract from both hardware and operation system.

While allowing for better control of the network processing, such solutions also come with the need to handle all possible situations that can occur. To keep this at a manageable level, the supported features might be reduced to a bare minimum while the packets with unsupported features are simply discarded. An example this is the *ixy* framework being developed at the *Technical University of Munich* [50], which is a purely educational framework.

Another result of these frameworks is that the whole network traffic on the chosen network interface is forwarded to the userspace program. It is not foreseen to process only parts of the traffic and forward the remaining packets to the Linux kernel. So, these kinds of frameworks should only be used for network interfaces for which no network traffic meant for the operating system is expected.

Especially the last point was a driving factor for the development of a new *fast lane* within the Linux kernel as described in the next section.

5.3 eXpress Data Path (XDP)

The power and versatility of the Linux network stack is also its problem when it comes to the performance. For example, a TCP packet to setup up a new connection (i.e. the *SYN* flag is set) would go through the whole processing chain just to be dropped because the software refuses the request as it has reached the maximum number of connection it can handle. Each additional request will also mean that the software needs some time to respond to the request, which could be better used to process the accepted requests. So it would be good to have an instance that would filter out any new requests until the software is able to accept new ones. Such a filter should be located as early as possible to reduce the work a dropped packet generates, preferably before the system needs to reserve a memory slot for this packet.

The solution for this is the XDP mechanism introduced in version 4.8 of the Linux kernel [51]. It is invoked as soon as software gets involved in the receiving process (i.e. within the driver of the NIC⁵), just before the memory for the packet and its meta data is allocated [52]. XDP uses a lightweight virtual machine-like construct within the Linux kernel called Berkeley Packet Filter (BPF)⁶ [53]. The users can load specialized programs with a restricted instruction set (e.g. loops are not allowed and calls to other BPF programs are limited to 32 nested calls) into this virtual machine. A verifier within the Linux kernel is used to ensure the guaranteed termination before the programs are *just in time*-compiled into code for the CPU architecture the system is running on. Even though the name BPF hints at the usage in the network stack, it is not limited to this and can also be used to access a growing number of different parts of the Linux kernel.

⁵If the driver does not support XDP, there is a generic implementation within the Linux network stack getting activated right after the packets are handed over from the driver.

⁶More precisely the extended version called eBPF.

The BPF programs are used by XDP to enable the checking of specific fields in a network packet and to decide, what should happen with it. In the case of the TCP requests, it can check if the *SYN* flag is set and can drop it when a defined limit of open connections is reached. Thus, the system does not process requests which will be declined anyway, keeping the system responsive in extreme situations (e.g. if a high number of requests is sent intentionally to disrupt the provided service, also known as Denial of Service (DoS) attack). The software can communicate with the BPF program via maps, so it can adapt the filtering as needed.

Next to dropping a packet or forwarding it to the network stack, the initial version of XDP also supported the redirection (i.e. insertion of the packet into the transmission queue of the same NIC). This latter variant was designed as a way of load balancing, but can also be used to analyze a portion of otherwise dropped packets. Even while it is possible for the BPF program to modify a packet (which should be done before being inserted into the transmission queue, otherwise it would just come back again), only a single action can be reported as decision. Therefore, it is not possible to send a copy of the packet to a monitoring system and forward the original packet to the network stack.

However, dropping or forwarding packets to an other node in the network does not reduce the load caused by the receiving itself. Therefore, a new path was introduced with version 4.18 of the Linux kernel allowing to redirect a packet directly to a user application. This comes with a new response type for the BPF programs and a new socket type called XDP socket (XSK). These sockets are not a bypass of the Linux kernel as the BPF is part of the kernel. This is more like an additional fast track without the need to stop at any toll station. But this also means that the packets are not processed, so the raw packet is delivered.

To use these XSKs, an application opens a network socket as usual, but with the address family type being *AF_XDP* instead of *AF_INET*. The main difference is how the packets are handed over. XDP uses a chunk of continuous memory being shared between kernel and user application to store the network packets. This memory area called *UMEM* is created by the user application in a manner that fits best for the usage and afterwards registered via a call of the *setsockopt()* function. It is divided into equal-sized frames with each frame holding a single packet, so the individual packets can be addressed by a frame identifier. The size of such a frame is defined by the user application, but must be a power of two as the kernel uses bit masks to get the frame identifier from the address offset.

To receive a packet, the user application provides a list of identifiers of empty frames where the kernel can store the received packets. This list is called *fill queue* and each entry points to exactly one frame in the *UMEM* (see Figure 5.3a). When a packet is redirected to the XSK, it is placed in the frame pointed to by the first entry of the *fill queue* and the identifier is moved to a second list called *receive queue* (see Figure 5.3b). So, this queue holds all packets received, but not delivered. The combination of both queues forms a ring buffer with the *fill queue* being on the writing side (the producer) and the *receive queue* on the reading side (the consumer).

There is also such a ring for the transmission path, being formed by the so-called

transmission queue and *completion queue*. The identifier for frames with packets to be sent are put into the former one and the processed frame is transferred back by the latter one. The frames are not cleared by the kernel so their identifiers can simply be inserted again into the *transmission queue* if a packet needs to be retransmitted. For more details about these queues see Ref. [54]. Within this theses, only the receiving path is used as the focus is on receiving the data.

There is a system library called *libbpf* which provides specialized functions to handle the XSK. The library also loads a predefined BPF program and provides a set of functions to access the received packets. The functions are the equivalent to the receive functions usually used. The main difference is that only frame identifiers are exchanged and the packet data is already in place instead of transferring an address where the packet data will be copied to.

Within the *libbpf* library, the handling of the frame identifiers is replaced by indexes of queue entries, which are in a strictly consecutive way. The queue entries then hold a pointer to the frame in the *UMEM*. This eases the packet processing as the queue indexes can simply be increased by one to get the next packet for processing.

It was initially planed that the XDP implementation should use the *zero copy* approach to gain maximum performance by avoiding the need to copy the data between different memory locations. But as it would require larger changes in the source code of the kernel and drivers⁷, it was first released without *zero copy* support. While still being planed, this feature was not released within the time of the tests for this thesis.

⁷The NIC would need to store the incoming packet directly in the designated UMEM frame instead of writing it into the ring buffer before notifying the driver.

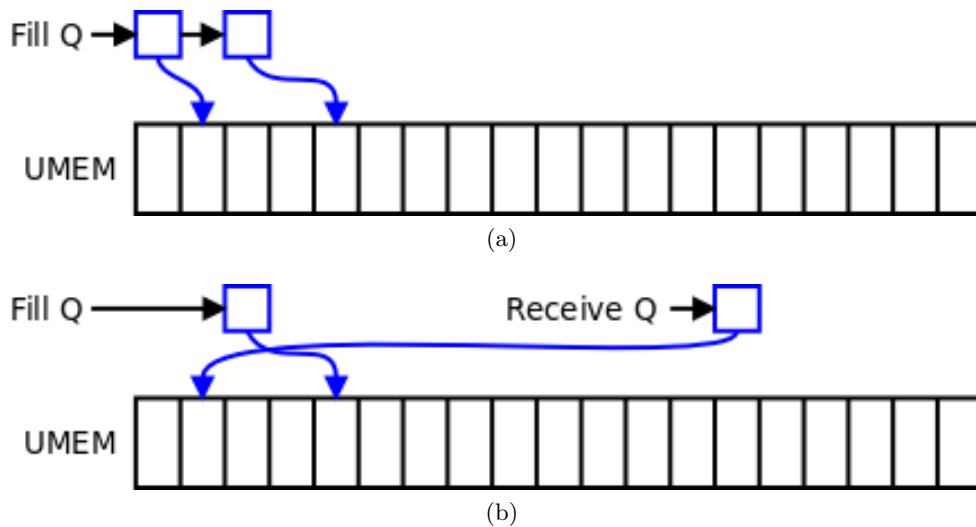


Figure 5.3: Packet reception in XDP using the *UMEM* [54]

- a) The available frame identifiers for packets are stored in the *fill queue*.
- b) When a packet is received, the frame identifier is moved to the *receive queue*.

5.4 System configuration

Before the packets are able to reach the receiving software, the systems needs some basic settings. As XDP works on receiving queues of the NIC, the various data streams need to be routed to the corresponding queue. This is done by so-called *ntuple* filtering and is configured via the *ethtool* tool. By default, the *rx-hashing* mechanism is enabled, so it needs to be disabled first:

```
sudo ethtool -K eth2 rx-hashing off
sudo ethtool -K eth2 ntuple on
```

Afterwards, the filter rules are created by specifying the type of data stream (*udp4* stands for UDP over IPv4) and the destination port as selector:

```
sudo ethtool -U eth2 flow-type udp4 dst-port 27051 action 1
```

The range starting with UDP port 27050 is chosen as an arbitrary port range with no assigned protocols to avoid mixed traffic [55]. Thereby, each port number is assigned to a receive queue. The Linux kernel maps all network traffic to queue zero by default, so the port assignment starts with port 27051 for queue one. Therefore, a data source can select a specific reception queue and the corresponding receive program by setting the respective UDP destination port.

As stated at the beginning of this chapter, receiving the data only is not the goal of the thesis. So, there are additional settings made to increase the performance. The first one is to assign a defined CPU core to the interrupt handlers of each receive queue. But before this, the *irqbalance* system service needs to be stopped. Otherwise, the assigned CPU core might be changed by this service again.

```
sudo systemctl stop irqbalance
systemctl status irqbalance
```

In the current setup, the interrupts for the receive queues start with *IRQ 126* for queue zero. Therefore, the configuration starts with *IRQ 127* for queue one on CPU core 0:

```
echo "00000000,...,00000001" | sudo tee /proc/irq/127/smp_affinity
echo "00000000,...,00000001" | sudo tee /proc/irq/128/smp_affinity
echo "00000000,...,00000004" | sudo tee /proc/irq/129/smp_affinity
```

The CPU used in the current system is organized in groups of two CPU cores as discussed in Section 5.1. With the assumption of each CPU core being able to process a data stream, two receive queues are mapped to such a group. Thereby, both queues are mapped to the same CPU core as they should not utilize the whole CPU core. It should be noted, that the affinity mapping uses enable masks. So, the value written for each affinity mapping is two to the power of the identifier of the CPU core (i.e. $2^2 = 4$ for CPU core two).

Furthermore, the CPU starting with the lower core numbers is completely excluded from the Linux scheduler, so the operating system does not put any process on these CPU cores by its own. This covers the CPU cores 0-15 as well as 32-47, with the latter ones being the second logical cores of the same physical cores as the former ones. Nevertheless, these cores are still available for the software to be explicitly placed on these CPU cores. This ensures that the processes being placed on such a CPU core will not be interrupted by other processes.

In addition to this, placing a thread on a single of these CPU cores helps in reducing memory traffic as the most often accessed memory locations will stay in the CPU cache. If the thread would switch between two CPU cores of different CCXs, the content of the CPU cache would be written back to memory and loaded again into the cache of the new CPU core, having a detrimental impact on the processing performance.

There are two more settings made to tune the network performance on system level. The first sets the rules for issuing a receive interrupt of the NIC to fixed values to avoid changing conditions while the tests are running. This is achieved by the following command:

```
sudo ethtool -C eth2 adaptive-rx off
sudo ethtool -C eth2 rx-usecs 200 rx-frames 64
```

With this setting, an interrupt is issued when 64 packets have arrived or after 200 μ s if the 64 packet limit is not reached in that time.

The second change is the value of the *maximum transmission unit (MTU)*, which defines the maximum size of a network packet and has a default value of 1500 B set by the operating system. A larger packet size increases the network efficiency as there is more data transferred with the same size of the header. It should also result in fewer resources needed for processing, as there will be less packet headers to be processed for the same amount of data. A usual value for the MTU for larger than normal packets is 9000 B, so this was set for the test system. But this resulted in a system warning:

```
mlx5_core 0000:01:00.0 eth2: XDP is not allowed with MTU(9000) > 3498
mlx5_core 0000:01:00.0 eth2: Link up
```

Therefore, the MTU is set to 3000 B (twice the default value).

5.5 Data processing

Once the system is set up, the program receiving and processing the data is the last missing part. The example design of XDP was taken as a starting point.

A single instance of the receiving program is limited to a specific CCX since the processing should be done with the whole data being in the local CPU cache. So, there are only four logical processing cores and a very limited amount of memory available.

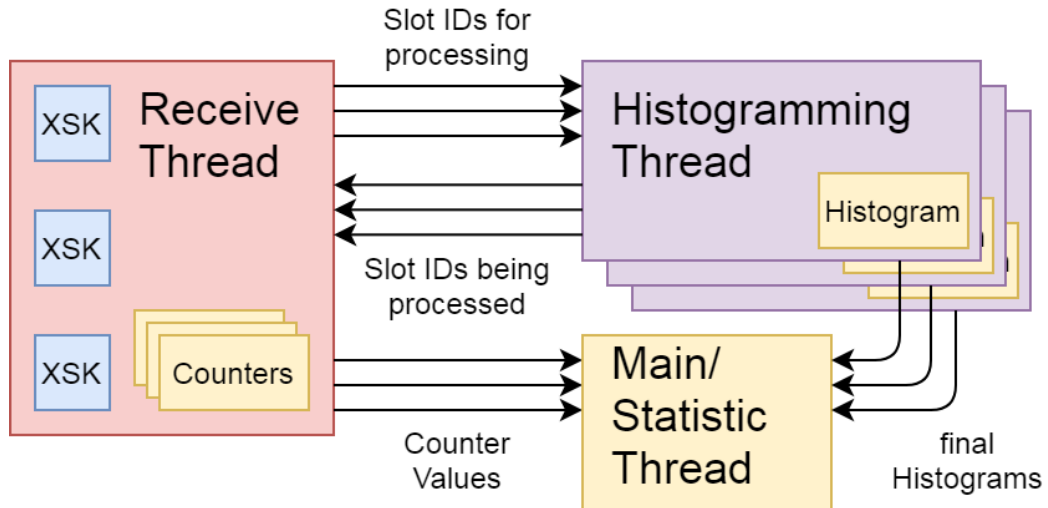


Figure 5.4: Interactions between the different parts of the receiving software.

This also limits the data rate that can be processed. To be able to process more data, several instances (one per CCX) are started across the isolated CPU. It would have been possible to combine all these instances into a single program. But this would have meant that one core would also have to administrate all the instances on other CCX, giving that specific core an additional workload compared to the others. With all instances being separate, the test conditions are more homogenous and flexible.

The receiving program is split into three threads after the initial configuration phase, which are summarized in Figure 5.4. Each thread has thereby a specialized task assigned to it:

The receiving thread uses the *libbpf* interface to receive the packet data. It also checks the DROP header and counts packets as well as errors. If the histogramming is enabled, it forwards the addresses of the packets to the histogramming thread.

The histogramming thread counts how often each 16 b word occurs in the data stream. If all current packets are processed, it waits until new packets are available for processing. If multiple data streams are configured to be received, there is a dedicated histogramming thread for each of them.

A status thread is generating a status output including the amount of data and the number of packets (both in total and since the last dump) on the console. The thread sleeps a defined period before dumping the next status. The status thread is also checking if the specified runtime is over and the program should terminate.

To have information for the status thread to print on the screen, these need to be collected at some point. As the receive thread already checks for missing packets,

it was also chosen to count the amount of data and packets. In total, it counts the following four values:

The total amount of received data is the sum of all DROP fragments including the DROP header. This is equal to the amount of UDP payload transferred as the DROP fragments are the UDP payload. The values are given in number of Bytes.

The total number of received packets is the sum of packets being reported as received when checking for new packets. If a packet is identified as not belonging to the data stream (i.e. the size does not match the expected one), this packet is ignored and also not counted.

The number of missed packets is calculated as the accumulated difference of the received packet identifier in the DROP header and the expected one. As this counter is an unsigned 32 b integer, a negative offset (i.e. the received identifier was smaller than the expected) leads to maximum sized integer value, making such a case clearly visible.

The number of missed batches counts how often a mismatch between the received and the expected packet identifier was detected, i.e. this counter is increased by one each time dropped packets are detected. This also helps in detecting cases where a positive and a negative offset neutralize each other in the missing packet counter.

These counters are normal integer typed variables written by the receive thread and read by the status thread. For these counters, there is no thread synchronization implemented as only one thread writes to their memory location and the counters are increased monotonously. The status thread does not care if the receiving thread is about to update a counter. If it gets the old value, the update will be included in the next query.

This is different for the transfer of frame identifiers for packets to be processed. The receiving thread generates a pair of pointers, one to the start of the packet and one to its length. Since the receiving thread cannot wait until the histogramming thread acknowledged the reception of such a pair, the pair is simply put into a queue. This queue (or its length respectively) is not behaving monotonously, therefore some protections are required.

The protection of the access to the queue is realized by a so-called *atomic flag*, which can be set and unset. Its property of being *atomic* means that it can be tested and set in a single instruction with the memory location being blocked for other threads for the time needed to execute. This *test and set* instruction returns the state of the flag as it was before it was set. When it was already set, a different thread is in the area protected by that flag and setting the flag again has not changed it. The current thread should wait a bit before trying again.

If the flag was not set, it will be set afterwards so that the current thread can proceed its execution without any additional costs for locking the corresponding

source code sequence. Nevertheless, it should unset the flag as soon as it leaves the protected code area to not blocking other threads. With this method, the queue can be modified without the risk of generating a mess if two threads modify it at the same time.

In the other direction, reporting a packet as processed also requires protection. But in this case, the problem is more subtle. The reporting is done by a counter of processed packets. Since incrementing the counter is (with respect to the memory areas) independent of the histogramming, the compiler would be able to move the instructions within the code to a place where it would fit best. This needs to be prevented as there is a logical connection between both that the compiler is not aware of.

To make the compiler aware of this, *atomic counters* are used. In addition to the operation being uninterruptible, they also offer so-called *memory barriers*. These barriers restrict the reordering of memory operations in a way, that all such operations need to be executed before the barrier. In this specific case, it ensures that the histograms are updated before the counter is incremented. Therefore, all the packet data is guaranteed to be counted before the frame of the packet is released. The same technique is used at the end of the program to ensure that all histograms are written before the statistic thread dumps the histogram into the log file.

Configuration phase

The configuration file used in the initial configuration phase is specified via a command line option and is divided into a section for the global configuration and for the configuration of the different data streams. There are also additional command line options to set the parameters for the first data stream if no configuration file is used. If a configuration file is used together with other command line options, the last value for a given parameter is used. The configuration file uses the *JSON* format and is described in more details in Ref. [56].

After reading the configuration, the main thread is switched to the CPU core where the interrupt handler for the given network receive queue is positioned as this core is not used for receiving or processing. The main thread will generate only negligible workload and has no timing critical task, so getting interrupted is not an issue here. The positioning of the receiving and processing threads can be changed by the configuration.

The next step is the setup of the XSKs for each data stream to be received. For this, a continuous block of memory is allocated which will be used as the packet buffer by XDP. Afterwards, the socket is set up and bound to the interface and to the receive queue⁸ given in the configuration. This also includes the loading of the BPF program for the given network interface. The loaded program is the default program just checking if there is an XSK active for the specified receive queue. This is due to the loading happening within the function creating and initializing the XSK which is part of the *libbpf*. The configuration of the sockets finishes with the

⁸This is not the queue utilized within XDP. In this context, it is the one of the NIC.

reservation of the packet slots for the consumer and producer rings. It is important to not reserve all of them for the consumer ring (i.e. where the received packets are stored) as some slots are needed for the producer ring to report the availability of the slots for new packets. The latter ones need to be reserved before the slots of the actual packets are released, so the slots of the packets cannot be directly reused.

Before the receive thread is started, a check if the data should be processed is performed and the histogrammers are started accordingly. This includes the histograms to be generated and initialized. The histograms are of the type `std::vector` with 65536 entries as the data generators within the FPGA uses 16 b counters to generate the data pattern (see Section 4.6). The vectors have the benefit that their elements are in a continuous memory area, which should ease the access to them within the cache. They are resized to the final entry number right after instantiation to prevent interruption while the vectors are growing in size. The entries are of the type of an unsigned 32 b integer, resulting in a total memory size of 256 kB for all entries.

Each entry accounts for 8 GB of data (as 2 B words are counted) before it overflows. Therefore, in case of continuously counting values as used in this thesis, a histogram will overflow after 512 TB which should be acceptable. It takes a bit more than five days with a 10 Gb/s data stream to reach such an amount of data. The events of an overflowing histogram are rare enough within a test run and easy to correct in the data analysis, so the usage of 64 b values for the entries is not foreseen as it would double the memory footprint.

After starting the histogrammer threads, the receiving thread is started. This order was chosen to ensure that all components are up and running before the receiving thread is started. Otherwise, the receiving thread would start checking packets and the majority of the slots will be occupied by packets waiting for processing instead of being available to buffer new incoming ones.

The configuration phase ends with the main thread becoming the statistics thread as there is no further task to be done by the main thread.

Receiving thread

The receiving thread takes care of handling the received packets. It performs the test for missing packets and collects statistics about these. If the processing of packet data is enabled, the data is forwarded to the histogramming thread of the corresponding data stream.

In contrast to the histogramming threads, there is only a single receiving thread taking care of all sockets and data streams assigned to the current instance of the program. At a first view, this looks like an unnecessary serialization of otherwise parallel processes as only one data stream can be checked at a given time. But all data streams use the same network link and are therefore serialized anyway. Also, the packets are received in batches within the software and a batch can be checked in less time than it takes to collect the next batch. The important point is that the receiving thread never reaches a point where it needs more processing power than is available as this would result in overflowing buffers and therefore packets being

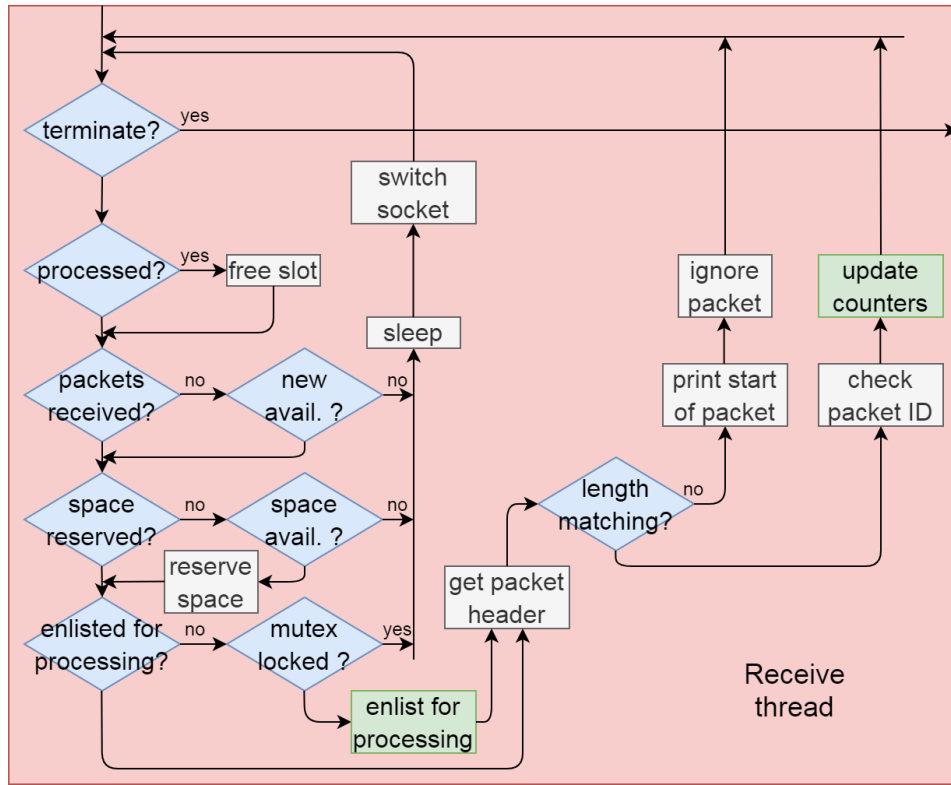


Figure 5.5: Internal structure of the receiving thread.

dropped. The main loop of the receiving thread is shown in Figure 5.5.

Before entering the main loop for data reception, all counters and other variables for the different data stream are initialized and the starting time is acquired. The main loop starts with a check if the histogrammer of the current data stream has finished the processing of additional packets. This is done by comparing an atomic counter of the histogramming thread counting the processed packets to the number of freed slots in the receiving thread. If the number of processed packets is higher, the corresponding slots are freed. Freeing a slot means that the entry with the pointer to this slot is removed from the receiving queue after a new entry pointing to the same slot is inserted into the filling queue. The memory itself is not freed and stays valid all the time.

At the same time, slots of potentially wrongly received packets are also freed. The current implementation for this might result in some slots being freed before being processed as such a wrongly received packet could come behind a still unprocessed packet. This scenario is considered as acceptable as the memory location is still valid but its content may have changed in between, resulting in a flawed histogram. Instead of fixing the handling of such packets, which would increase both the complexity of the source code and the processing time needed, the occurrence of such packets should

be avoided. This is achieved by moving the check of the packets to the BPF program.

The next check is whether there are still some packets to be checked and processed or if a new batch of packets should be acquired from the receive queue. In case of the latter, a function of the *libbpf* is called to get a new batch of packets. As the entries pointing to these are in consecutive order in memory, only the number of available entries and therefore packets is returned. This function also has a parameter for a maximum batch size. This is used to achieve a more homogenous operation of the receive thread and can be specified in the configuration of the receive program. If no new packets are available, the processing is paused with a call of the *sleep* function, afterwards the data stream is set to the next one and the loop restarts at the beginning. If there are new packets, the number of available packets is stored in a variable and the thread proceeds.

As an entry in the fill queue is needed for each packet received to store the pointer of actual slot, this is checked next. If there is not enough space available, the same steps as in the case of no new packets being available are done. The processing is paused, the data stream is set to the next one and the loop is restarted.

The last check is whether the address and length of the previous packet were forwarded for processing. If these were not, this is done now, otherwise this step is skipped. The forwarding step is protected by an atomic flag⁹ (which can be tested and set within a single command) as it might include access to variables being shared between receiving and histogramming thread. The execution is again paused, the data stream switched and the loop restarted if the flag was already set before (i.e. the histogramming thread is accessing the list) to avoid blocking and packet loss. If the flag was not set before and the processing of the packet data is enabled, the address and length are inserted as a pair into the list¹⁰ of packets waiting for processing. If the processing of the packets is disabled, the counter of processed packets is directly incremented as there is no histogrammer thread to do this.

Finally, when all checks and remaining tasks have finished, the packet header itself is analyzed. As the system is configured to sort incoming packets according to the UDP destination port, it is assumed that all packets are UDP packets. Therefore, the raw packet should start with an ethernet header, followed by the ones of IPv4 and UDP. So, these are parsed and the start and length of the DROP packet is extracted. The length is checked against the nominal packet size of the data generator and the one of the last packet of a data block (which should be smaller) to verify that the packet belongs to the data stream and is not any random packet.

If the length matches, the packet identifier of the DROP header is read and compared to the expected values to determine if packets got lost in between. If the difference is not zero and it is not the first packet received, the counter of missed packets is increased by the difference while the counter of missed batches is incremented by one. There is also an output generated both in the console and the log

⁹This is a more lightweight variant of a mutex.

¹⁰This is implemented as another queue. It is referenced as a list to prevent confusion with respect to all the other queues.

file including the last packet identifiers seen as well as the current and the expected packet identifiers. The date and time of the event is also added to check for simultaneous events in parallel streams. Afterwards, the expected packet identifier for the next packet is calculated by increasing the one of the current packet by one.

Histogramming thread

The reception of packets and the checking of headers alone does not give a reliable picture of what amount of processing is needed. With reading only the header of the packets, only a small fraction of the received data is transferred, masking possible problems resulting from memory bandwidth or cache size limitations. Furthermore, synchronization of threads generates extra work load. Reading all data bytes and counting the occurrence of each word adds a further plausibility check and therefore helps to determine the quality of the data transmission. Figure 5.6 shows the loop of the histogramming thread.

It is important to not exploit any prior knowledge about the pattern of the test data streams to get realistic measurements. The data streams of a real detector environment will not have a fixed pattern included that would allow for a shortcut in processing. The setup should also set a realistic upper limit on the amount of data to be processed. If this limit would be artificially increased by the usage of the internal structure of the data streams, the requirements of the overall system size could be underestimated.

The usage of large network packets (to increase the network efficiency) implies that the histogrammer will most likely be the bottleneck in packet processing. This means, that some special effort should be applied to optimize this part of the receiving software, even if the complexity of it is rather low.

The main task is to count the occurrence of each 16 b data word. This is done by using a pointer of the type *uint16_t* and a *while* loop to iterate over the packet data until the end is reached. In each iteration, the content of the memory location that the pointer is pointing to is taken and used as the index for the vector used for the histogram. The value at this index is then incremented by one.

That quite simple loop accounts for the majority of the work load needed for a received packet. Therefore, the code for this loop was subject of additional investigations if it could be further optimized. An online tool for exploring the generated assembler code was used [57]. The result is that the chosen implementation is already a very optimized solution:

```
while(ptr != end)
    (*histo)[*(ptr++)]++;
```

A variant of this with a manually unrolled loop was also tested and found to produce smaller code with less frequent checks for the end of the packet. But this version showed up to have less throughput when tested for no obvious reason. Therefore, the design was reverted and kept as shown above.

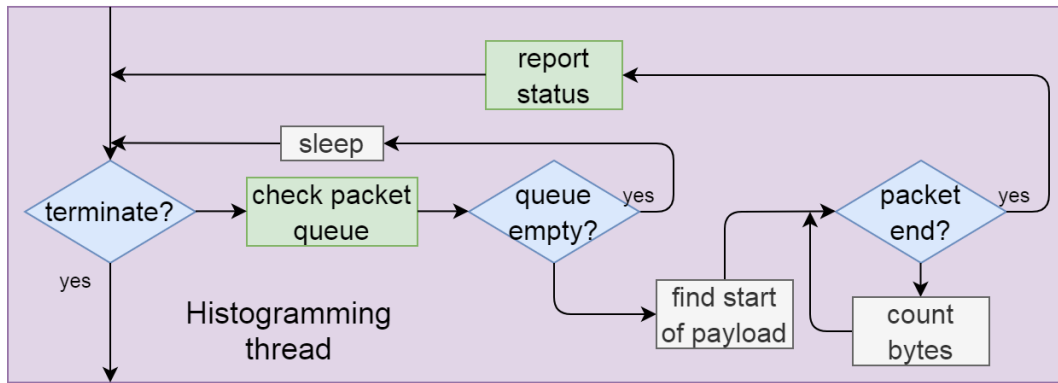


Figure 5.6: Internal structure of the histogramming thread. The main work is done in the loop counting the data words (depicted on the right side).

When the end of the packet is reached (i.e. the pointer is pointing to the first element behind the packet data), the counting loop is aborted and the status is reported by incrementing the counter of processed packets. Afterwards, the main loop is restarted with the check for new packets. The histogramming thread also checks the atomic flag before accessing the list of packets to be processed. In contrast to the receiving thread, the histogramming thread retries to get the flag until it gets it as there is no other work to be done here. When the flag was acquired, the list of packets is checked for a packet to be processed next. If the list is empty, the histogramming thread pauses execution by a call to the sleep function and restarts the main loop afterwards.

As soon as the next packet is found, the histogramming thread sets the pointer iterating over the packet to the start of the data block¹¹ as well as a second pointer to the first element behind the data block. The latter one is used to determine the end of the packet while counting the data words.

Statistic thread

The statistic thread generates several indicator values from the counters of the receive thread. Therefore, it runs an endless loop consisting of a sleep (default value is 10 s) and a call to the *dumpStats* function. As a call to the sleep function does not last exactly the given time period, a precise time stamp is acquired in the *dumpStats* function to calculate the time since the last call. To have a reference for the first call of *dumpStats*, such a time stamp is also loaded in the receive thread before the packet handling is started.

The values for each counter of the receiving thread are shown directly in addition to the difference to the last dump. In addition to the absolute values, there are also

¹¹The DROP header is skipped here as it would result in a inhomogeneous histogram as it contains a field with a constant value.

```

status (running time: 82110.793s, interval time: 10.000s ):
      value (interval/total) | rate (interval/total)
received data   : 25549120672 / 209762978053832 | 20439.079 / 20437.068 Mbps
received packtes : 12749138 / 104672356438 | 1274.900 / 1274.770 kpps
missed packtes  : 0 / 0 | 0.000 / 0.000 mpps
missed Batches  : 0 / 0 | 0.000 / 0.000 mbps

status (running time: 82120.793s, interval time: 10.000s ):
      value (interval/total) | rate (interval/total)
received data   : 25532541580 / 209788510595412 | 20425.814 / 20437.066 Mbps
received packtes : 12740808 / 104685097246 | 1274.067 / 1274.770 kpps
missed packtes  : 8406 / 8406 | 840590.984 / 102.361 mpps
missed Batches  : 237 / 237 | 23699.746 / 2.886 mbps

```

Figure 5.7: Shown is the example output of the statistic thread. While the first interval saw no packet loss detected, there were 8406 packets dropped distributed over 237 batches in the second interval.

the rates (i.e. the values divided by the corresponding time period) displayed for each value. The displayed rates are scaled to have more useful magnitudes, so the resulting units are:

- *mega bit per second* (in contrast to the amount of data being displayed in Bytes) for the data rates
- *kilo packets per second* for the packet rates
- *milli packets per second* for the *missed packets* rates
- *milli batches per second* for the *missed batches* rates

In addition to the counters and rates, there is also the overall running and interval time displayed. If more than a single stream is configured, the values for each stream are plotted in separate table.

In Figure 5.7 an example of the output generated by the statistic thread is given. It shows the output of two intervals with a separation of 10s after a running time of 82110s. The data stream received ran at about 20.48 Gb/s with a packet size of 2004B (including the 4B DROP header). Within the interval between both prints, 8406 packets distributed over 237 batches (i.e. about 35.5 packets/batch) got dropped. These were the first packets lost in this test run, resulting in a ratio of lost to received packets of 8.03×10^{-8} .

Chapter 6

Tests and measurements

To verify the assumption that packet loss can be avoided by the measures described in the previous chapters, the network transmissions need to be tested and characterized. Since the individual network components themselves are already quite complicated, the interaction can make it difficult to determine the exact reasons for a transmission problem. Therefore, the testing is split up into four phases with a dedicated topic in each phase.

The first phase was about the network stack in the FPGA. In these tests, the data rate of the network stack was limited to 10 Gb/s. In addition, some basic measurements were done to get first numbers on achievable user data rates and packet drop. Since the DROP protocol is not part of the network stack itself, it was not used within this phase.

In the second phase, the tests focused on the receiving side. Based on the results of the first phase, the network stack was adapted for higher data rates, reaching 40 Gb/s. Phase two also covers the receiving software and includes first results of the XDP technique. Some fundamental design choices were made based on these results. This phase made use of mainly a single data stream due to the focus on the characterization of the data transmission.

The tests of phase one and two used a desktop PC with an *Intel(R) Core(TM) i7-4790K* CPU, offering four physical cores with a clock frequency of up to 4.00 GHz. Thereby, each physical core provided two logical cores by utilization of the *Hyper Threading* technology, resulting in eight logical cores being available. Next to the very limited number of central processing unit (CPU) cores, the quite old system architecture proved to be unable to handle the desired data rates.

Since a single data stream worked and the old setup was not suitable for the full bandwidth, the setup was extended by a new server to allow further testing. This also resulted in an adaption of the receiving software, which was further optimized for the new CPU architecture (see Section 5.1 for more information about this). Therefore, the topic of the third phase are the effects of scaling up the data transmission in terms of streams (and therefore overall data rate) as well as the new system running the receiving software. The main results were achieved in this phase demonstrating the full capabilities of the system in terms of data transmission.

While the third phase concentrated on a single set of parameters and the overall concept proved to be working, a broader validation was missing. Therefore, the fourth testing phase was a round up and tested a wider range of the parameter space. In

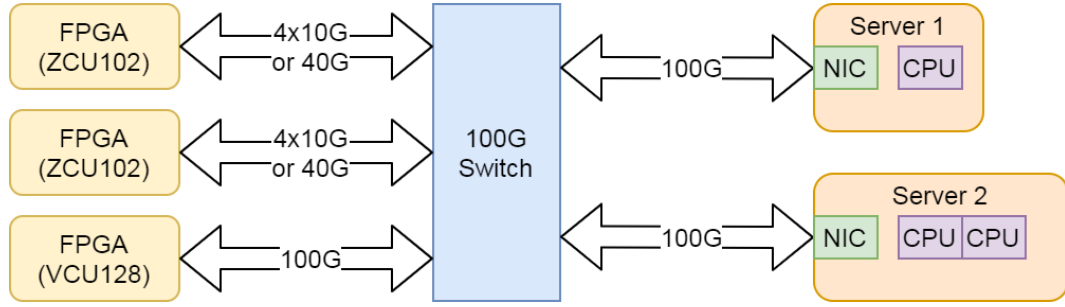


Figure 6.1: Overview of the test setup. In the final setup, there were three FPGA evaluation boards (two *ZCU102* and a *VCU128*, all from *Xilinx*) and two computers (one desktop PC with eight cores and a server with two times sixteen cores) available. The servers and the FPGA boards were connected via a network switch.

addition to this, a new FPGA board was available for the tests allowing the extension of the network stack to a data rate of 100 Gb/s.

Figure 6.1 shows the final setup with the different data rate options.

6.1 First Phase: Hardware Testing

The goal of the first phase of testing was the validation of the network stack in the FPGA. The initial version of the stack was the 10 Gb/s version (64 b @156.25 MHz) as this was supported by the existing hardware.

The tests started with the verification of the network stack. Therefore, a data generator and a pattern checker were implemented to be able to perform tests with different data patterns (e.g. running number or pseudo random number, 16 b words each) [58]. The tests were extended in length to measure the bit error rate (BER).

In tests with different data patterns, a combined duration of approximately 160 h and 689 TB of transferred data, not a single error was noticed. To check if the setup was working correctly, one end of the fiber was pulled out slightly and the system started to show errors with the rate being dependent on how far the plug was pulled out. This proved the network stack to be working, so the detailed performance of it was further investigated.

To still calculate a bit error rate, it is assumed that the next received bit has flipped, which results in:

$$\text{BER} = \frac{1 \text{ b}}{689 \text{ TB} \times 8 \text{ b/B}} = 1.81 \times 10^{-16} \quad (6.1)$$

This is a very good result for a first test. But as a dropped packet would contain several thousand bytes, it would take a much higher amount of data to be transferred to reach the same rate in such a case. Nevertheless, the BER is still important for

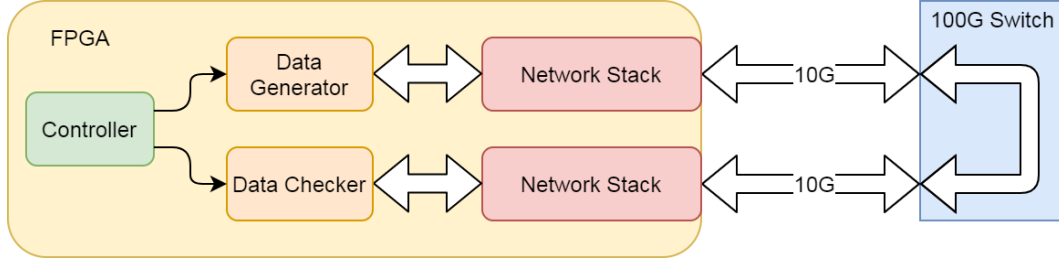


Figure 6.2: Overview of the test setup used for the delay measurements. The two network stacks were instantiated separately within the same FPGA to ease the control of both instances. On the network side, the network stacks were connected via a network switch.

the correct reception of packets. Also, the network interface card (NIC) will drop packets in case of the checksum of a packet does not match, so a single bit error can easily result in a dropped packet.

The second set of tests in phase one measured the latency of the transmission path and the receive path dependent on the packet size. The two network stacks were placed within the same FPGA to allow both the data generator and the data checker to use the same clock reference and therefore having a common time reference. This enabled the latency to be defined as the time from the last data word leaving the data generator to it reaching the data checker. To also include the contribution of the ethernet subsystem, the data had to leave the FPGA and therefore includes the optical transceivers and fibers. The test setup was also extended to included a network switch to have a network component with a different clock source in the transmission path. With the network traffic taking the full path from the data generator to the network switch and back to the data checker within the FPGA, the time needed by the packets is also called *Round-Trip Time (RTT)* or t_{rt} for short. The complete setup is shown in Figure 6.2.

As long as the additional delay being introduced by the network switch is independent on the packet size, the network switch contributes to the constant offset of the optical transmission path and can therefore be regarded as a part of the optical fiber. The network stack itself is expected to have a packet length dependent delay coming from the two packet buffers (one in the transmit path and one on the receive side) as well as a constant part coming from the packets running through the encoder and decoder blocks. So the expected latency is defined by:

$$t_{rt} \approx 2 \times \text{delay}_{\text{buffer}} + \text{delay}_{\text{fiber}} + \text{delay}_{\text{stack}} \quad (6.2)$$

The time within the FPGA is measured in clock cycles as the data generator and data checker get their clock signal from the same source:

$$t_{rt} \approx 2 \times \frac{\text{delay}_{\text{buffer,cycle}}}{f_{\text{clock}}} + \frac{\text{delay}_{\text{fiber,cycle}} + \text{delay}_{\text{stack,cycle}}}{f_{\text{clock}}} \quad (6.3)$$

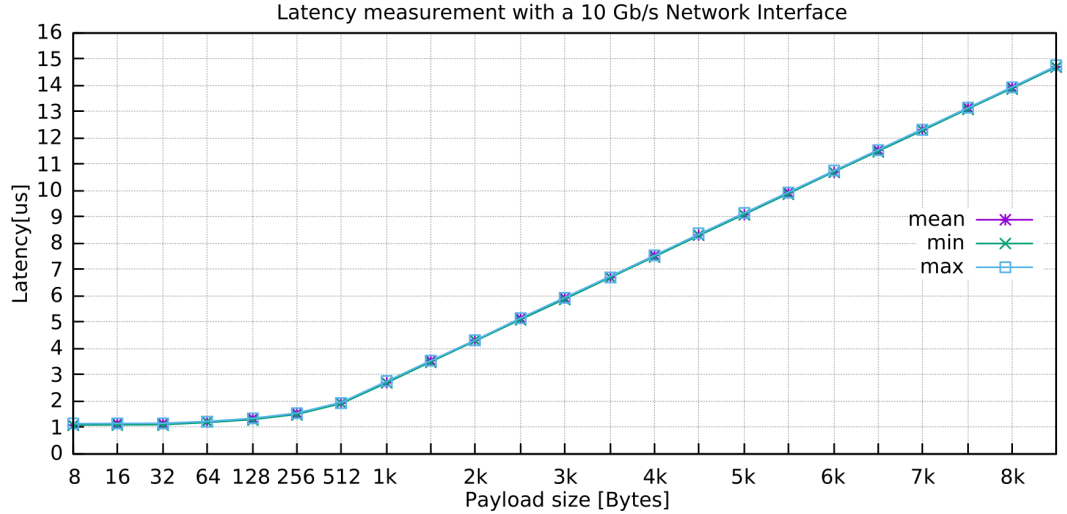


Figure 6.3: The latency as measured in [58]. The two components (i.e. the constant offset and the slope) are clearly visible.

Figure 6.3 shows the result of the measurements. It is clearly visible that there is a component depending on the packet size as well as a constant offset. The slope of the distribution is on the order of 4 bytes/cycle, meaning each additional data word (i.e. 8 B of payload) increases the delay by two clock cycles (i.e. 12.8 ns). This corresponds to the two packet buffers being included in the network stack as both can write a data word with each clock cycle.

Therefore, there seems to be no other payload dependent delay in the setup. This is important as the test also included the network switch, which is now confirmed to also have no packet size dependent component. This means, that the network switch is directly forwarding the incoming packet instead of storing it in buffers until it is completely received.

The offset is the accumulated delay caused by the network stack, the fibers and the network switch¹ and has a mean value of 173 clock cycles (i.e. slightly above 1 μ s as the network stack and the data generators run at 156.25 MHz). With 8 bytes/cycle, this would correspond to a packet with a length of 1384 B. Thus, for packets with a size above this value, the first bytes of a packet would be already written into the receive buffer, while the last byte had left the transmit buffer. This also means, that such packets spend more time waiting in the transmit buffer than is needed to reach the designated destination in the network.

The overall formula to calculate the latency is therefore:

$$t_{rt,meas} \approx 2 \times \frac{\text{bytes/packet}}{8 \text{ bytes/cycle} \times 156.25 \text{ MHz}} + \frac{173 \text{ cycles}}{156.25 \text{ MHz}} \quad (6.4)$$

¹This measurement cannot distinguish them.

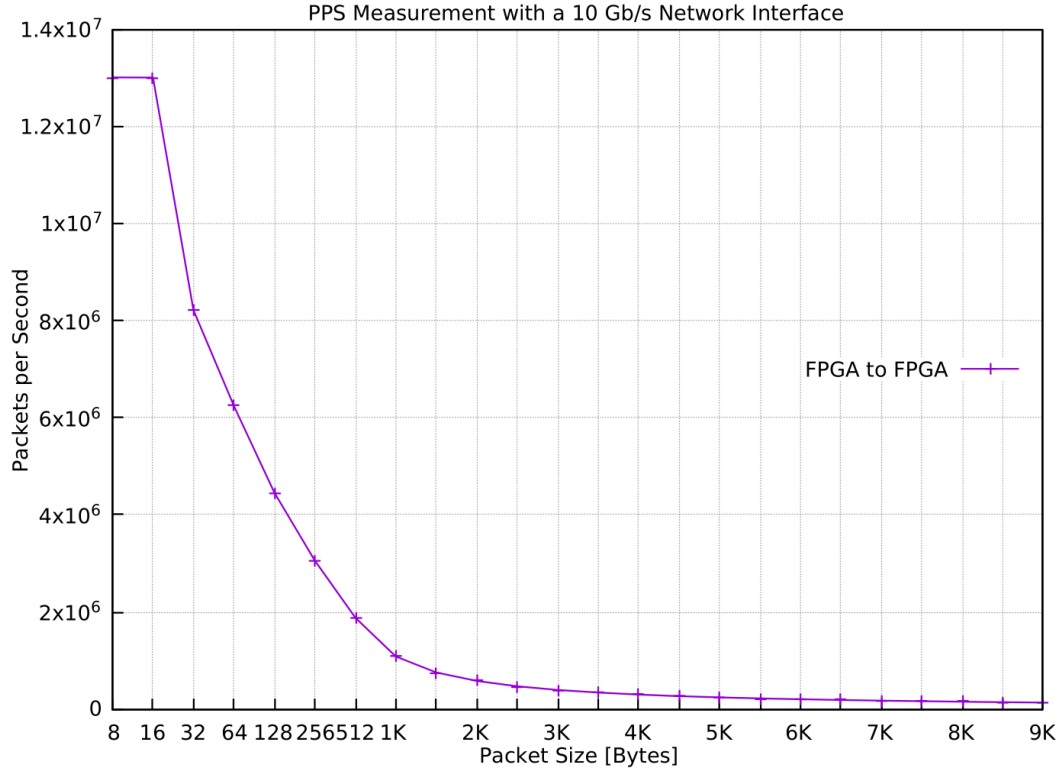


Figure 6.4: Rate at which packets can be transferred from FPGA to FPGA [58]. The values for 8 B and 16 B payload are the same as both are below the minimal frame length of ethernet.

The next test performed was a measurement of the number of packets which can be sent and received per second. This was done first from FPGA to FPGA and later in both directions between FPGA and PC as a comparison².

In the FPGA-to-FPGA scenario, the results shown in Figure 6.4 form the expected curve. The values for a payload size of 8 B and 16 B are expected to be the same as the amount of data per packet is not enough to fill an ethernet frame. This has a minimal length of 64 B, of which 42 B are needed by the different protocol headers and 4 B being reserved for the checksum at the end. So in both cases, additional padding is inserted to fill the remaining 18 B, making the total length the same.

For these small packet sizes, a maximum packet rate of about 13 million packets per second is achieved. With a clock frequency of 156.25 MHz, a new packet is received every twelve clock cycles. This hints at additional overhead in packet generation since a packet with the minimum packet size needs only eight clock cycles to be transferred on the data bus. Even the preamble and interpacket gap being inserted by the ethernet protocol should only account for two to three clock cycles (i.e. 20 byte

²There were also tests done with the embedded ARM CPU but these are not relevant here. For more information about these tests, see Ref. [58].

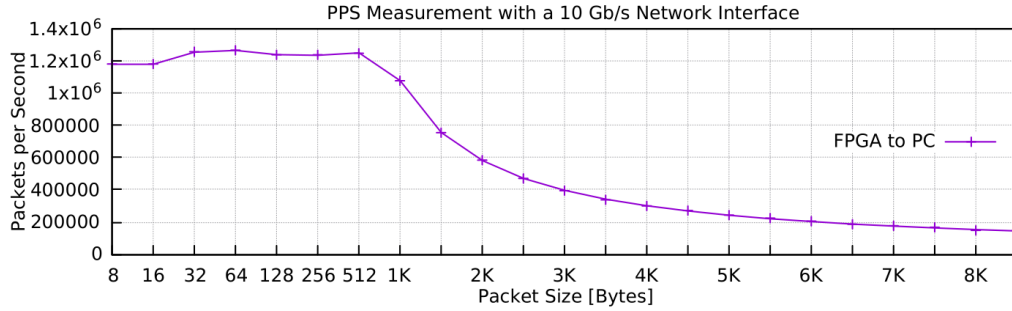


Figure 6.5: Rate at which packets are received when transferring from FPGA-to-PC [58]. It is clearly visible that the PC has a limit on how many frames per second can be processed (independent of their sizes).

for both combined), leaving at least one clock cycle for further overhead.

Figure 6.5 shows the results for the FPGA-to-PC case. It is clearly visible that the PC can process only a limited number of packets per second (around 1.2 million here). If the packet rate gets higher, the PC has to drop all additional packets. With higher packet lengths, the packet rate decreases and the PC is able to process most of the packets again. Some packets are still lost over time for various reasons, but its number cannot be quantified here. This starts at a payload size of 1000 B bytes as the FPGA was able to send packets with a payload size of 512 B at a rate of near to 2 million packets per second.

But both tests show only the rates seen by the receiver. These can neither give information on the number of lost packets nor if the theoretical limit of the network is reached. The case of data transmission from the FPGA to the PC gives at least a hint which side is causing the limitation.

For getting a better picture of this, the directions are swapped (i.e. PC to FPGA now) and the test is repeated. Thereby, the measurements switched from the pure number of packets to the bandwidth. It is important to note here, that the bandwidth within this thesis is given as the data rate of the payload (i.e. the output of the data generator or input of the data checker). This does not include the overhead of the underlying protocols (i.e. ethernet, IPv4 and UDP). So, these values cannot reach the data rate of the network link itself.

As Figure 6.6 shows, the PC has again problems to generate small packets at high rates, while the FPGA is able to process all of them. But with larger packet sizes, the FPGA starts to see fewer packets than the PC is sending. The architecture of the network stack ensures that all received packets can be processed³, the missing packets must be dropped somewhere else. Due to a link speed of 100 Gb/s between PC and network switch, the PC is able to generate a higher data rate than the 10 Gb/s connection between network switch and FPGA can handle. Therefore, the network switch has to drop all packets above the capacity of that link.

³All error checking can be disabled, so only the protocol identifier need to match to count a packet as seen.

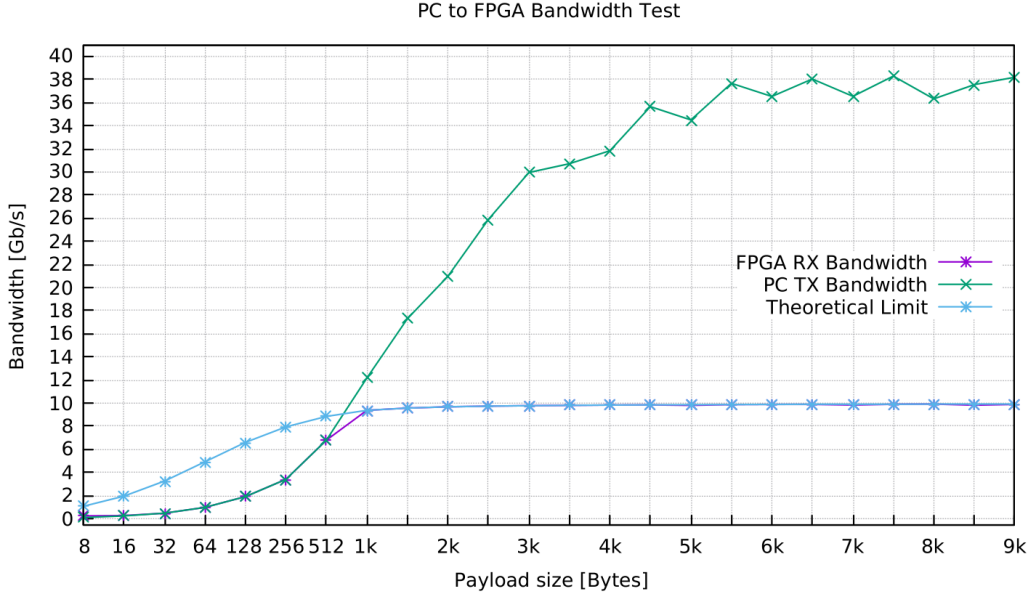


Figure 6.6: Comparison of the bandwidth that the PC is able to send with the one that the FPGA sees (as measured in Ref. [58]). The PC can reach a higher bandwidth as the network link between PC and network switch is much faster than the one connecting the FPGA.

The capacity can be described as the practical limit which can be measured. In contrast to this, there is also a theoretical limit which can be calculated from the properties of the protocols, with the ratio between both as an indicator of the quality of implementation used.

For the tests done in this phase, the theoretical limit was defined in Ref. [58] as the maximum reachable UDP payload bandwidth:

$$\text{bandwidth}_{\text{theo}} = \text{bandwidth}_{\text{network}} \times \frac{\text{payload}_{\text{UDP}}}{\text{payload}_{\text{UDP}} + \text{overhead}_{\text{total}}} \quad (6.5)$$

with

$$\begin{aligned} \text{overhead}_{\text{total}} &= \text{overhead}_{\text{ethernet}} + \text{overhead}_{\text{IPv4}} + \text{overhead}_{\text{UDP}} \\ &= 66 \text{ B} \end{aligned} \quad (6.6)$$

This definition of the theoretical limit is missing a part describing the minimal size of ethernet frames. In addition, the insertion of the padding is done by the ethernet subsystem (see Section 4.3). Some delays from the network stack are hidden by this for small packets. Therefore, the values for packets with a payload of 8 B or 16 B need to be taken with some caution in the following figures.

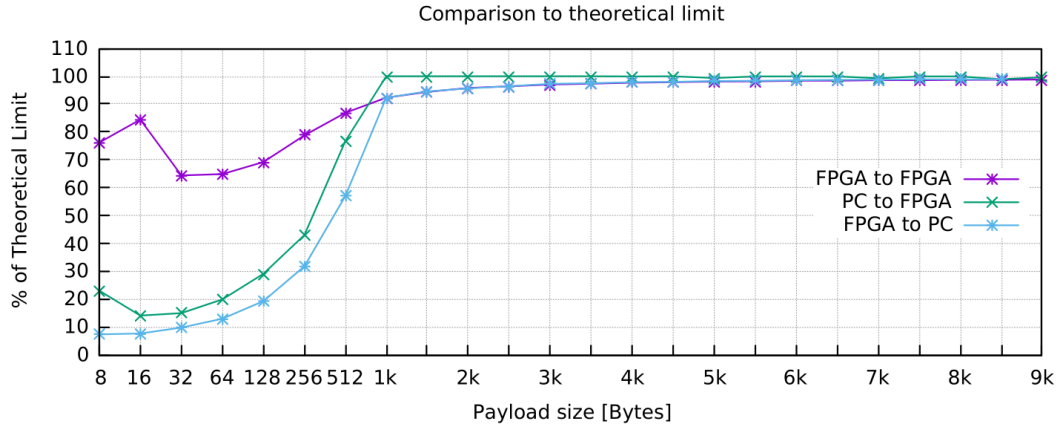


Figure 6.7: Bandwidth comparison relative to the theoretical limit [58].

Figure 6.7 shows the bandwidths reached in relation to the theoretical limit for a better understanding in how far this limit is reached. It clearly states that the efficiency rises with increasing packet size and a payload size of at least 1000 B is needed to reach efficiency values above of 90 %, so larger packet sizes should be preferred for bulk transfers.

Furthermore, the measurement shows a dependence on the direction of the data transmission. While the receive path of the network stack within the FPGA is able to reach the theoretical limit (as soon as the PC is able to generate it), the transmit path fails at this point. Therefore, the transmit path was the limiting factor in the FPGA-to-FPGA test.

This also confirms the discrepancy for small packets found in the packet rate measurements. The ratio reached for a packet size of 32 B is in the same order as the ratio found at that measurement⁴.

The transmit path of the network stack has some processing steps which pauses the data transmission for a short period. The overall delay from this seems to be too high to be masked by the processing steps of the ethernet subsystem, so the theoretical limit cannot be reached. One of these steps is getting the MAC address for a given IPv4 address, which needs to block data transmission. That takes a few clock cycles even if the MAC address is known to the system. Figure 6.8 shows the complete picture with the bandwidth reached by the different scenarios.

The FPGA implementation not reaching the theoretical bandwidth needs to be included in the bandwidth calculation for the whole system. Therefore, some headroom should be reserved on top of the data to be transferred which might result in additional network interfaces being needed per FREDDIE. If this is not possible, the input bandwidth needs to be reduced by spreading the FE chips over more readout units.

⁴The value for 32 B is taken as the values for 8 B and 16 B has a wrong description for the theoretical limit.

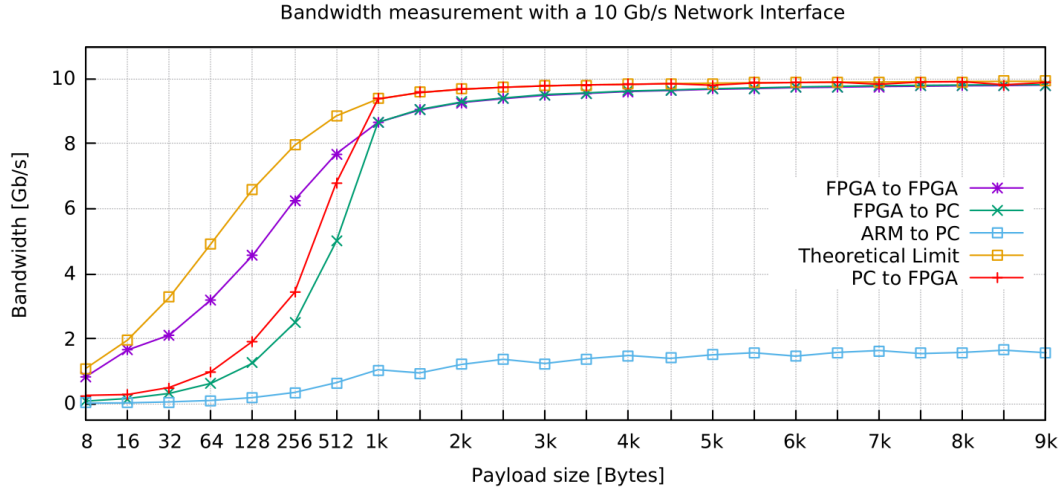


Figure 6.8: Comparison of the bandwidth reached in different scenarios as measured in Ref. [58].

The tests of the data rates still leave the question about the packet loss open. Additional measurements were done to address this open issue. Figure 6.9 shows the relative number of packets being dropped for the different scenarios. Here, it becomes visible what fraction of packets (and therefore also payload data) gets lost. It should be noted, that the rate of lost packets never reached zero completely in the cases where the PC is the receiver. For the larger packets, it settled around 0.008 % and is therefore too small to be seen in the plot.

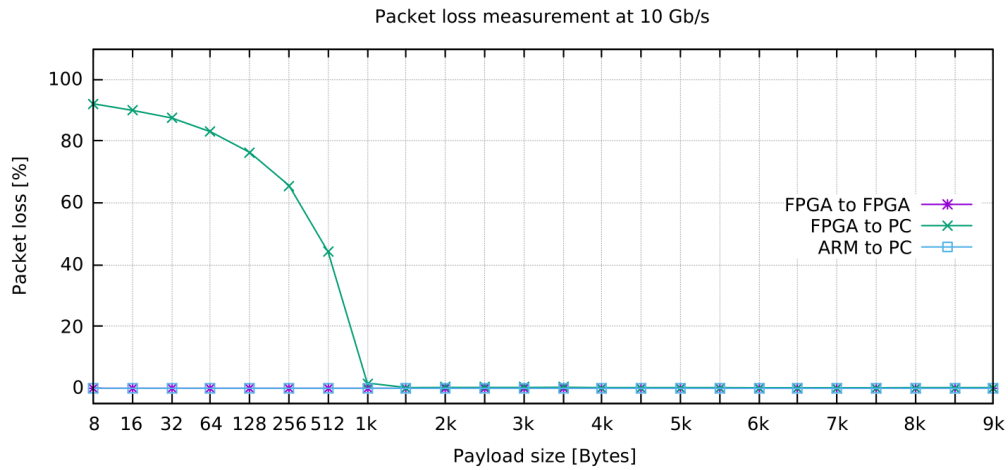


Figure 6.9: Rate of lost packets as measured in Ref. [58]. The number of dropped packets does not reach 0 for the FPGA to PC case, but is too small to be noticed. Only the FPGA to FPGA scenario reaches a value of 0.

This first testing phase showed that the network stack works. While the receive path is able to process packet rates up to the theoretical limit, the transmit path has some additional delay preventing it from reaching the full bandwidth. This is in contrast to the PC, which was able to generate higher data rates for a given packet size than it was able to process. Larger packet sizes performed generally better than smaller ones. Thus, the final system should use packet sizes as large as the system is allowing. This might be limited due to latency requirements or some components being unable to handle packets above a certain limit. The observed ratio of lost to received packets of 0.008 % makes the goal of avoiding retransmissions appear reachable.

6.2 Second Phase: Transmission characterization

With the network stack working at a rate of 10 Gb/s, it was extended to also support a line rate of 40 Gb/s. This was needed to enable parallel full bandwidth data streams from the same FPGA system. Furthermore, the DROP protocol was implemented during this phase, allowing for better network characterization by usage of the packet identifier.

On the PC side, the receiving software was improved to give intermediate results (instead of only a summary at the end), enabling real time tracking of configuration changes. The first tests in this second phase still used the *traditional* approach of Linux network programming.

With both sides of the transmission having changed, the tests restarted with a simple packet rate measurement to verify the changes. The program creates a timestamp as soon as a million packets are received and compares it with the last timestamp. After printing to the console, the current timestamp is saved and the packet counter reset. As the first phase suggested the usage of larger packets, payload sizes of 4000 B and 8000 B were used.

The minimalistic output already shows the variations of the time needed to collect a given number of packets and therefore indicates that also the processing will encounter changing work loads. For a payload size of 4000 B, the time varied by several hundred μ s. For a payload size of 8000 B, this time varied by several ms.

One reason for such fluctuations is that the network connection between the network switch and the PC needs to transfer also other packets and thus might interrupt the even distribution generated by the FPGA. Other reasons are the packet processing in batches of varying size and non-constant delays within the various processing steps. It is important to keep these variations in mind as the periods with slightly higher work load will be the ones resulting in packet loss. So, there is some headroom needed on top of the average packet rate to handle such spikes.

The next step was to include the counting of missing packets by checking the packet identifier within the DROP header. This should not only give a number of total missed packets, but also an idea of their distribution. Therefore, the number of missing packets is printed to the console for each mismatch of the expected and


```

1 packets missed
1 packets missed
1048576 packets in 3285291 us = 0.319 Mpps
1048576 packets in 3283557 us = 0.319 Mpps
1048576 packets in 3283426 us = 0.319 Mpps
1048576 packets in 3283057 us = 0.319 Mpps
3 packets missed
1 packets missed
1 packets missed
15 packets missed
5 packets missed
1 packets missed
1048576 packets in 3283402 us = 0.319 Mpps
5 packets missed
2 packets missed
^C
received 396370144 packets with 1585453736864 bytes in 1241.185 s (0.319 Mpps, 10218.967Mbps)
missed 24448 packets (0.000 Mpps)

```

Figure 6.10: The occurrence of missing packets has no obvious pattern. This test used a payload size of 4000 B.

received packet identifier.

Figure 6.10 shows that the rate of missing packets as well as the size of missed blocks has variations similar to the packet rate, hinting at the same cause.

In addition to this, these tests showed that the relative packet loss for a payload size of 4000 B (in the order of 6.17×10^{-5}) is smaller than for 8000 B (in the order of a few percent, see Figure 6.11). An explanation for this could be that the smaller packets should fit into a single memory page of 4096 B whereas the larger ones need two pages of continuous memory, which might generate additional overhead. This result stands in contrast to the suggestion of the first testing phase as it puts a limit on the packet size to be used. Therefore, a compromise needs to be found.

The distribution of the number of packets being lost in a batch also led to some decisions in the design of the DROP protocol. With double-digit numbers of lost packets per batch, the usage of a FEC mechanism to rebuild lost packets is regarded as not feasible. As the additional data for the FEC should be kept small with respect to the protected data, the information for the reconstruction would have to be spread over several hundred packets. On the receiving side, there is neither enough memory to hold such an amount of packets to be able to do the necessary calculations in the case of missed packets. Nor is there enough processing power to do the calculations for each block so that the slots holding the received packet could be reused as soon as the packet data was taken into account.

With a ratio of missed to received packets in the order of 1×10^{-5} , the packet loss is already quite low. But periods with no packet loss at all raised expectations

```

received 7903468780 packets with 63227750240000 bytes in 50583.349 s (0.156 Mpps, 9999.773Mbps)
missed 196121513 packets (0.004 Mpps)

```

Figure 6.11: Test with huge packets showed higher packet loss. In this test with a payload size of 8000 B, about 2.5 % of the packets got lost.

```
cannot pass map_type 17 into func bpf_map_lookup_elem#1
xdpsock_user.c:xsk_configure_socket:315: errno: 1/"Operation not permitted"
```

(a)

```
mlx5_core 0000:01:00.0 eth2: XDP is not allowed with MTU(9000) > 3498
mlx5_core 0000:01:00.0 eth2: Link up
```

(b)

Figure 6.12: Initial problems encounter after switching to XDP.

that the packet loss could be overcome completely. At this point, it seemed more promising to use a new technique called XDP (see Section 5.3) than a more detailed search for the reason of the variations.

To start with this emerging technique, the first tests were done with the kernel example designs. But the support was still quite fragile and had some incompatibilities between the system library *libbpf* and the Linux kernel as shown in Figure 6.12a. There was no way around this problem as the BPF programs used for the XDP mechanism require this particular map type. This was only solved with an upgrade to Linux kernel 5.2.9 which was recently released at that time.

Even with that upgrade, the test parameters had to be adapted as XDP limited the packet size to less than a memory page (see Figure 6.12b). As the packet buffer used by XDP requires the frame size to be a power of two, the payload size was cut in half again to a value of 2000 B. Thus, even with the protocol headers included, the total packet size stayed below 2048 B. This had the benefit of doubling the number of packets to be buffered within the same amount of memory.

As soon as the problems were solved and the example design ran, it was adapted to learn how the new technique worked and eventually ended in being the base for the new receiving program. With this, the results directly showed significant improvements compared to the *traditional* approach. With a bit of fine tuning of the processes within the receive program, a single data stream of 10 Gb/s could run for a few days without losing a single packet as shown in Figure 6.13. This example reached a ratio of lost to received packets of 4.62×10^{-12} , assuming the next packet would have been dropped.

Since the PC used for these tests has four physical CPU cores only, each core would need to process more than a single 10 Gb/s data stream to make full use of the

```
status (running time: 339253.677s, interval time: 10.000s ):
      value (interval/total) | rate (interval/total)
received data   : 12749716160 / 432514264424896 | 10199.688 / 10199.194 Mbps
received packtes :    6374871 /   216257570470 |    637.482 /    637.451 kpps
missed packtes  :         0 /                   0 |     0.000 /     0.000 mpps
missed Batches  :         0 /                   0 |     0.000 /     0.000 mpps
```

Figure 6.13: One of the first long-term tests done with XDP using a single 10 Gb/s data stream. No packet was dropped within 94 h.

```

status (running time: 82120.793s, interval time: 10.000s ):
      value (interval/total) | rate (interval/total)
received data      : 25532541580 / 209788510595412 | 20425.814 /      20437.066 Mbps
received packtes   : 12740808 / 104685097246 | 1274.067 /      1274.770 kpps
missed packtes     : 8406 / 8406 | 840590.984 /      102.361 mpps
missed Batches     : 237 / 237 | 23699.746 /      2.886 mbps

```

Figure 6.14: Doubling the data rate to a value of 20 Gb/s, first bursts of dropped packets (the values of the current interval are the same as for the total test) were seen after 23 h.

100 Gb/s network interface. Therefore, the output of two or more data generators within the FPGA are merged into a combined data stream. This is possible as the DROP header is added after combining the streams and the version of the receive program used in these tests had no histogramming implemented.

As Figure 6.14 shows, the tests for a 20 Gb/s data stream were also doing fine for a long period before a burst of dropped packets occurred (e.g. after about 23 h in this test). With help of the increased data rate, the ratio of lost to received packets was still in the same order as for the 10 Gb/s tests. But for the latter ones, the ratio was only an upper limit of this ratio rather than a measured value. This indicates, that a single CPU core gets to its limited with such a data rate.

Increasing the data rate even further, it became obvious that a limit of the test system was reached as the frequency of missed packets increased drastically. For example, a data stream with a data rate of 25 Gb/s had already many smaller bursts of dropped packets within the first hours of running as it can be seen in Figure 6.15.

At first, it was seen that the CPU utilization of the receive program reached 100 % from time to time, so it was unable to handle all packets arriving at that time. This was also seen for a data rate of 20 Gb/s, but without losing packets at each occurrence. It was noted, that also the CPU utilization for the network interrupt was quite high (around 60 %). In these tests, the CPU core being used by the test was isolated from the Linux scheduler, meaning that no other process ran on this core.

```

status (running time: 5420.069s, interval time: 10.000s ):
      value (interval/total) | rate (interval/total)
received data      : 31962108016 / 17323500305720 | 25569.406 /      25569.416 Mbps
received packtes   : 15949188 / 8644478714 | 1594.901 /      1594.902 kpps
missed packtes     : 0 / 662 | 0.000 /      122.139 mpps
missed Batches     : 0 / 54 | 0.000 /      9.963 mbps

status (running time: 5430.069s, interval time: 10.000s ):
      value (interval/total) | rate (interval/total)
received data      : 31961897596 / 17355462203316 | 25569.286 /      25569.416 Mbps
received packtes   : 15949083 / 8660427797 | 1594.894 /      1594.902 kpps
missed packtes     : 71 / 733 | 7099.936 /      134.989 mpps
missed Batches     : 4 / 58 | 399.996 /      10.681 mbps

```

Figure 6.15: Using a data rate increased to 25 Gb/s, bursts of dropped packets were seen after a shorter time period and with higher frequency.

To verify if the seen limitation is due to usage of a single stream (i.e. a single CPU core running a solitary instance of the receive program), the test was repeated with the two data generators being configured to use separate streams of 10 Gb/s each. For this, a second CPU core was isolated from the Linux scheduler to allow for an additional instance of the receive program.

The result of this test also showed periodical packet loss, but with each lost batch containing fewer packets. At this point, it became clear that the PC itself was the limitation and a new one would be needed to reach the intended data rates, especially as the tests in this phase were not processing any content of the received packets.

6.3 Third Phase: Scaling up

With the old PC being unable to handle the required data rates, a new server was ordered. Once the new server had arrived, the third phase was started.

As the new CPU has a lower clock frequency than the old one, the load per CPU core needs to be evaluated again. But it would be enough to receive and process a single 10 Gb/s data stream on a group of two CPU cores (i.e. a CCX, see Section 5.1 for more information) as the server has two CPU sockets with 16 physical (i.e. 32 logical) CPU cores each. Therefore, the tests started with a data rate of 10 Gb/s per stream. The payload size was kept at 2000 B as the new server got the same Linux installation as the previous PC (i.e. the *Fedora* distribution in version 30, with the Linux kernel of version 5.2.9).

These tests were quite successful, also with more than 10 Gb/s per stream. Besides tests with higher data rates for a single stream, the main test configuration became the usage of four data streams with 10 Gb/s per stream. The main reason for this was that the used CPU allowed for eight data streams and a combined data rate of 80 Gb/s was regarded as close enough to the maximum data rate of the NIC, leaving some headroom for further network traffic. This is especially true as the data rate values of the data streams are for the payload data, therefore not including any protocol overhead like headers. Due to this overhead, the fourth stream never reached the full bandwidth. With the network link of the FPGA only supporting a line rate of 40 Gb/s at that point, the fourth data stream had to use the remaining available bandwidth, resulting in an actual data rate of about 8 Gb/s. The tests including both FPGA boards used six data streams with a data rate of 10 Gb/s and two data streams with about 8 Gb/s.

Nevertheless, there were some open points to be addressed. First of all, there was still some sporadic packet loss for no obvious reason. A test could run for a few days without any packet getting lost and then lose about a hundred packets. These events seemed to have a pattern, as all streams lost roughly the same amount of packets at the same time. In addition to this, the packets were always lost in two batches within the same second. But the time of day at which it happened seemed to be random, so the output for each missed batch was enabled again and extended by a timestamp as shown in Figure 6.16.

```

last values: 370476, 370477, 370478, 370479, 370531
Sat Jan 25 07:02:53 2020
last values: 433973, 433974, 433975, 433976, 434016
Sat Jan 25 07:02:53 2020
status (running time: 136061.548s, interval time: 10.000s ):
      value (interval/total) | rate (interval/total)
received data   : 12723755200 / 173101812317792 | 10178.875 / 10177.853 Mbps
received packets : 6349192 / 86378324560 | 634.911 / 634.847 kpps
missed packets  : 92 / 92 | 9199.884 / 0.676 mpps
missed batches  : 2 / 2 | 199.997 / 0.015 mbps

```

Figure 6.16: Example of packets being dropped at a Saturday morning. A similar event happened in the following night around 4:17 am.

```

2020-01-25T06:02:47Z DEBUG Repo: Herunterladen von Remote: updates-modular
2020-01-25T06:02:49Z DEBUG Repo: Herunterladen von Remote: updates
2020-01-26T03:17:47Z DEBUG Repo: Herunterladen von Remote: updates
2020-01-26T06:19:57Z DEBUG Repo: Herunterladen von Remote: updates-modular

```

Figure 6.17: Excerpt of the log file of the packet manager. It shows entries roughly at the same times as the packet loss occurred. The timestamps in this particular log file are given in Coordinated Universal Time (UTC) instead of Central European Time (CET) as the ones printed by the receiving program.

The weekday and time of day (e.g. early morning at weekend) indicate that the packet loss was not caused by human interaction. Therefore, the system log was checked and there were some entries found at a similar time as shown in Figure 6.17. The timestamps in this particular log file are given in UTC while the receiving program uses CET. The log file belonged to *dnf* (the packet manager of the Linux distribution used) and the entry stands for getting an update of the remote repository. Even if the timestamps do not match exactly, this process takes several seconds and therefore spans over the occurrence of the packet loss.

To be sure about this, an explicit test was executed by triggering a similar lookup by hand. This resulted again in packets being missed as Figure 6.18 shows, so the cause was found.

```

missed batches : 0 / 0 | 0.000
last values: 12635877, 12635878, 12635879, 12635880, 12635929
Fri Jan 31 11:17:34 2020
last values: 12699314, 12699315, 12699316, 12699317, 12699368
Fri Jan 31 11:17:34 2020
last values: 14404738, 14404739, 14404740, 14404741, 14404785
Fri Jan 31 11:17:36 2020
last values: 14468212, 14468213, 14468214, 14468215, 14468264
Fri Jan 31 11:17:36 2020
status (running time: 669747.662s, interval time: 10.000s ):
      value (interval/total) | rate (interval/total)
received data   : 12723744908 / 852153181519296 | 10178.874 / 10178.797 Mbps
received packets : 6349187 / 4252269985808 | 634.911 / 634.906 kpps
missed packets  : 193 / 534 | 19299.769 / 0.797 mpps
missed batches  : 4 / 12 | 399.995 / 0.018 mbps

[~]$ dnf list dnf-automatic
Fedora Modular 30 - x86_64                29 kB/s | 23 kB | 00:00
Fedora Modular 30 - x86_64 - Updates      28 kB/s | 19 kB | 00:00
Fedora Modular 30 - x86_64 - Updates      482 kB/s | 822 kB | 00:01
Fedora 30 - x86_64 - Updates              150 kB/s | 19 kB | 00:00
Fedora 30 - x86_64 - Updates              5.3 MB/s | 5.4 MB | 00:01
Fedora 30 - x86_64                        51 kB/s | 23 kB | 00:00
Verfügbare Pakete
dnf-automatic.noarch                      4.2.17-1.fc30 updates
[~]$

```

Figure 6.18: Provocation of an update of the remote repository while having a test running proved the packet manager to be the reason for the sporadic packet loss.

```

status (running time: 605549.282s, interval time: 10.000s ):
      value (interval/total) | rate (interval/total)
received data   : 12723809036 / 770488638028176 | 10178.893 / 10179.038 Mbps
received packetes : 6349219 / 384476145884 | 634.912 / 634.921 kpps
missed packetes  : 0 / 0 | 0.000 / 0.000 mpps
missed batches   : 0 / 0 | 0.000 / 0.000 mbps

```

(a)

```

status (running time: 605549.017s, interval time: 10.000s ):
      value (interval/total) | rate (interval/total)
received data   : 10234299112 / 619737806626204 | 8187.304 / 8187.450 Mbps
received packetes : 5106946 / 309251027971 | 510.686 / 510.695 kpps
missed packetes  : 0 / 0 | 0.000 / 0.000 mpps
missed batches   : 0 / 0 | 0.000 / 0.000 mbps

```

(b)

Figure 6.19: The results of the test after disabling the update service. The second and third data stream have similar values as the first one being depicted in a). The fourth one being shown in b) has a lower data rate due to the limitation of the network link between FPGA and network switch.

In principle, such processes of the operating system should not affect the tests as the whole CPU was isolated from the Linux scheduler and therefore reserved for the tests. Also, the server has a separate 1 Gb/s ethernet interface configured as default network interface, which is connected to a separate network infrastructure and provides the external connections. Therefore, the 100 Gb/s NIC should be limited to the test data streams⁵.

At a first glance, disabling the corresponding service seems to be problematic as this is also needed for security updates of the whole system. But the server running the tests is located in a special network within the laboratory, being separated from the external network by a gateway. The same applies for the networks of the experiments at the LHC, which have large separated control networks for the detector operation. It should be safe to assume, that external threads being able to bypass the gateway would also be able to gain access to the server regardless of installed software versions.

With the update service being disabled, the tests were repeated for verification. This time, not a single missing packet was detected within seven days of continuous running for all four data streams used. With about 1.462×10^{12} packets, this resulted in an upper limit of the ratio between missed and received packets of about 6.837×10^{-13} (see Figure 6.19).

Seeing no errors leads to the question if the error checking mechanism works correctly. Therefore, a test to verify the error detection code was conducted by generating a mismatch of the expected and received packet identifier. This can be achieved by assigning a second data stream (with its own packet identifier counter) to the same receiving program. As the program will now get interleaved packets from both streams, it is reporting a lot of missed packets.

⁵There is some housekeeping (e.g. neighbor detection) done on all interfaces, so there will always be some additional traffic on an active interface.


```

status (running time: 220.003s, interval time: 10.000s ):
      value (interval/total) | rate (interval/total)
received data      :      0 / 189625395320 |      0.000 /      6895.370 Mbps
received packetes  :      0 /  94623642 |      0.000 /      430.101 kpps
missed packetes   :      0 /      0 |      0.000 /       0.000 mpps
missed batches    :      0 /      0 |      0.000 /       0.000 mbps

Negative Offset found!
last values: 10737607, 10737608, 10737609, expected: 10737610, got: 7550024
Batch size: 14
Wed Feb 12 13:00:02 2020
status (running time: 230.003s, interval time: 10.000s ):
      value (interval/total) | rate (interval/total)
received data      : 4183978568 / 193809373888 | 3347.130 /      6741.098 Mbps
received packetes  :  2087818 /  96711460 |  208.778 /      420.479 kpps
missed packetes   : 4291779710 / 4291779710 | 429171160482.854 / 18659642740.497 mpps
missed batches    :      1 /      1 |    99.998 /       4.348 mbps

```

Figure 6.20: Result of the test of the error detection mechanism. After the first stream was stopped, a second stream with a different value as packet identifier was sent to the same instance of the receiving program. This resulted in a mismatch between expected and received value and therefore proved the error detection to be working as expected.

A better way to show this is to start a test with a single data stream and stop the stream at a random point in time (without stopping the receiving program), leaving the packet identifier counter at an arbitrary value. Afterwards, a second data stream was sent to the same instance of the receiving program. With the second data stream having a different value for its packet identifier counter⁶, the first packet arriving at the receiving program will trigger the error detection mechanism and a block of missing packets is reported. The result of such a test can be seen in Figure 6.20.

The receive program compares the expected packet identifier with the one of the current packet. If these differ, the expected identifier is subtracted from the received one to calculate the number of missing packets as it is more likely that the received identifier is ahead with respect to the expected one. With the usage of unsigned 32 b variables, a negative result of this calculation is displayed as a very high positive number in the order of 10^9 as the variable underflowed. With the packet identifier being a 24 b counter, such a large number cannot be reached by dropped packets in a single batch. If such a huge number of packets would be dropped, the packet identifier would have overflowed already several times in this period. Also, the tests used a packet rate of about 630 kpackets/s, so it would take nearly 2 h to generate this amount of packets, while the test ran only for a few minutes. This shows that no packet got lost in the test with two streams and the displayed number results purely by the (here intended) jump in the packet identifiers.

A different way to verify the correct reception is to check the received data. With the payload being a counting pattern, each data word (i.e. 16 b of data) should occur

⁶This can be checked by reading the counter values via the configuration path. Setting a specific value would also enable such tests, but this is not implemented as it would most likely lead to more problems as it can solve.

```

log file generated at: Mon Mar 16 11:05:56 2020

status dumped at: Mon Mar 23 13:28:03 2020

status (running time: 613324.167211s, interval time: 6.407873s ):
Socket 0:
      value (interval/total) | rate (interval/total)
received data      : 8152919004 / 780364936055096 | 10178.627493 / 10178.825199 Mbps
received packetes  : 4068313 / 389404448262 | 634.892891 / 634.908046 kpps
missed packetes    : 0 / 0 | 0.000000 / 0.000000 mpps
missed batches     : 0 / 0 | 0.000000 / 0.000000 mbps

histogram :

1646860999, 1646860999, 1646860999, 1646860999, 1646860999, 1646860999, 1646860999, 1646860999,
1646860999, 1646860999, 1646860999, 1646860999, 1646860999, 1646860999, 1646860999, 1646860999,
1646860999, 1646860999, 1646860999, 1646860999, 1646860999, 1646860999, 1646860999, 1646860999,

```

Figure 6.21: First lines of a log file containing histogram. The counter values are printed as a list of integer values.

with the same frequency. Therefore, counting the occurrence of each 16 b word in a flawless transmission should result in homogenous values. This also has the positive side effect of processing the transmitted data which was not done until this point of time. Therefore, such a test gives a more realistic scenario when estimating the processing capabilities needed to process the event data.

To allow for such checks, the receive program was expanded by the histogramming feature, including a dump of the histogram (i.e. the counting values for each data word) in the log file. Figure 6.21 shows the head of such a log file. This starts with the timestamps of start and end of the test, includes a copy of the final status output and ends with the dumped histogram. If a lost packet would be detected, this would be added before the timestamp at the end of the test.

When checking the histogram against the number of received bytes, it should be noted that the number of received bytes (as displayed in the status output) is the accumulated UDP payload length of all received packets. As this also includes the DROP headers, 4 B per packet need to be added to the number of data words counted in the histogram.

$$\begin{aligned}
 \text{bytes}_{\text{received}} &= \sum \text{bytes}_{\text{payload}} + \sum \times \text{bytes}_{\text{DROP headers}} \\
 &= 2 \text{ B} \times \sum \text{frequency}_{\text{word}} + 4 \text{ B} \times \text{number}_{\text{packets}}
 \end{aligned} \tag{6.7}$$

In case of the test results presented before, this becomes:

$$\begin{aligned}
 \text{bytes}_{\text{received}} &= 2 \text{ B} \times (54\,440 \times 1\,646\,860\,999 + 11\,096 \times 1\,646\,860\,998) + \\
 &\quad 4 \text{ B} \times 389\,404\,448\,262 \\
 &= 215\,857\,364\,838\,736 \text{ B} + 1\,557\,617\,793\,048 \text{ B} \\
 &= 217\,414\,982\,631\,784 \text{ B} \neq 780\,364\,936\,055\,096 \text{ B}
 \end{aligned} \tag{6.8}$$

The result of this calculation appears to be far off the counted bytes. But with the difference between both being the sum of 512 TB and 2000 B, the reason for the mismatch becomes clearer. The share of 512 TB is the amount of data needed to completely fill a histogram (2 B per data word, 2^{16} different word in a histogram, 2^{32} occurrences per word to overflow the counter), so the whole histogram seemed to have overflowed once. The correct way of accounting for this would be to use bigger integer types for the histogram and prevent the overflow. But with the known pattern of the data generators and the payload size not being a power of two, it is safe to assume that overflows are evenly distributed and each entry overflowed exactly once.

The remaining 2000 B correspond to a single packet. As the receiving thread counts the payload of a packet before it is enlisted for processing and the histogramming thread also quits as soon as the receiving thread stops, there is no guarantee that all received packets are processed when the program terminates. So, a difference of a single packet sounds reasonably fine as the remaining difference.

These histograms can also be plotted to give a better overview. Within these plots, each pixel stands for the number of occurrences of a single 16 b word. The 16 b words are split into the upper and lower byte to form a two dimensional plot out of the one dimensional array. Thereby, a single packet with a payload size of 2000 B correspond to a bit less than four lines.

Figure 6.22 shows the plot of a histogram with no packets lost. It is clearly visible, that there are only two different values for the occurrence, with the higher value filling the lower half. The words represented by higher values have each a single occurrence less as the test stopped before the histogram was fully filled.

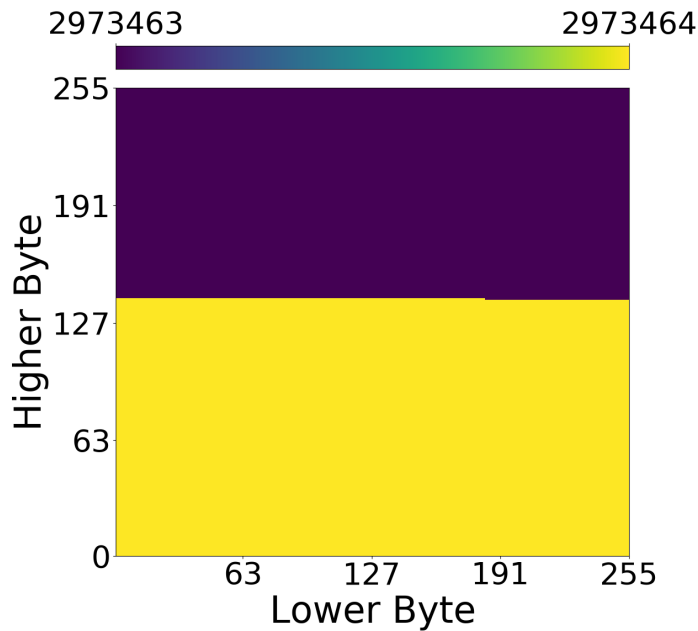


Figure 6.22: Histogram of a flawless data transmission as shown in Ref. [56]

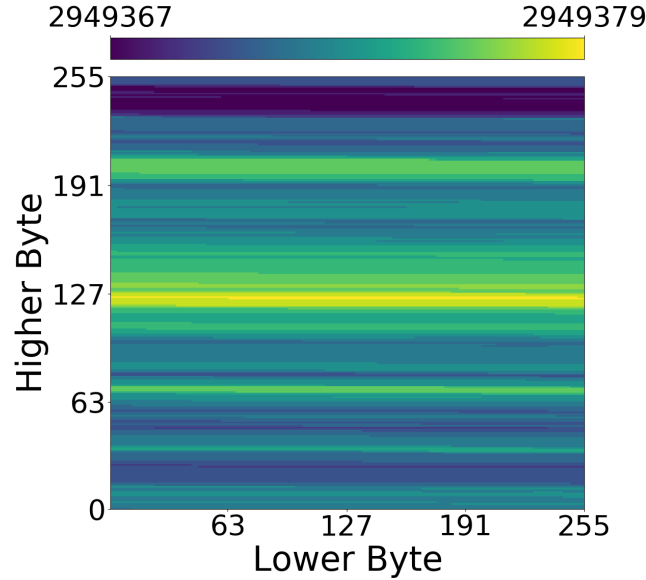


Figure 6.23: Histogram of a flawed data transmission as shown in Ref. [56]. The greenish-blueish areas are missing many packets.

This is expected as the counting pattern is counting up, starting with a value of zero. Therefore, the histogram is filled from left to right and bottom to top. As there is no gap in this filling scheme, a missing packet like in the aforementioned calculation must be the last one. That fits perfectly to the explanation given above.

In contrast to this, the plot shown in Figure 6.23 is an example of a plot showing lost packets. The difference between the highest and lowest value counted is larger than one and the stripes of missing packets are clearly visible.

Having a payload size being not a power of two helps in distinguish different blocks of missed packets as this shifts the start and end with the plot. For example, with a payload size of 2048 B, all packets would start at the left edge and end at the right edge, exactly filling four lines. Therefore, a block of 65 packets being dropped would have the identical color distribution as a single missed packet, with only the occurrence being off by one. With a payload size of 2000 B, the start of each packet is shifted to the left by 24 pixels, making it easy to distinguish a block of 65 packets from a single packet.

It is important to note that these histograms are created for each data stream separately, so each of these needs to be checked. Also, to get the combined performance of the data transmissions, the results of the different streams need to be summed up.

With the histogramming feature being verified, the measurement with a runtime of a week was repeated. In this, two FPGAs with a 40 Gb/s network stack each were used to create a combined workload of 80 Gb/s. It was built of eight data streams out of which six had a payload data rate of 10 Gb/s and the remaining two reached about 8 Gb/s each.

Lasting 604 806 s (i.e. about seven days), 5 855 156 146 597 424 B (i.e. 5325.233 TB) were transferred within 2 921 740 513 336 packets (i.e. 2.92 trillion). Again, not a single lost packet was detected, resulting in a ratio of lost to received packets of less than 3.423×10^{-13} . With this test being the third one presented in this thesis, the reproducibility of the results was demonstrated.

As the setup reaches its capacity limits (a bit of headroom should be kept), a significant improvement of the ratio of lost to received packets would require a significant increase in runtime of the measurements or a multiplication of the setup. For example: To reach a ratio of 1.0×10^{-14} , the tests would need to run about 34 weeks (i.e. two-third of a year) with the current setup. This is already on the order of the duration of the annual data taking phase of the ATLAS detector. The multiplication of the setup would shorten the time, but would also cost a lot of money to purchase the servers and FPGA boards.

Therefore, the tests about the scaling of the data transmission were stopped at this point. They proved that the transmission of a huge amount of data can be done without the burden of guaranteeing a reliable transmission. But as these tests used a fixed set of parameters for each stream, a last phase of testing was started to check the performance with a broader parameter space.

6.4 Fourth Phase: Round up

The fourth phase of testing aimed at a broader understanding of the network transmission. Therefore, a series of additional tests were conducted as described in Ref. [56]. This phase also had small changes in the test setup as a new FPGA evaluation board was available for the tests. The new FPGA enabled the expansion of the network stack to support 100 Gb/s. Thus, that sole FPGA was able to supply the complete server with data streams. This new FPGA design also included more data generators (up to 16) to be more flexible in testing.

The receiving side was slightly changed as well. The whole test procedure was automated as it was clear that this phase needed a high number of rather short tests. Furthermore, the receiving program was created in a version with disabled error reporting as some tests would generate a lot of these. Otherwise, the receiving program would be busy with reporting and would lose additional packets.

In a first step, the tests started with a verification of the data and the packet rate calculation since this is the basis to make precise predictions. These tests confirmed the Equations 4.1 and 4.2 of Section 4.6.

The confirmed formulas were used afterwards to determine the maximum packet rate the server is able to handle. Therefore, a single data stream with a payload size of 64 B was used to generate a packet rate of 16 Mpackets/s. The result was the receiving program seeing only a packet rate of 4.66 Mpackets/s with disabled packet processing (3.88 Mpackets/s if the processing was enabled). With a second stream configured in the same way, both receiving programs got the same result each, doubling the total received packet rate.

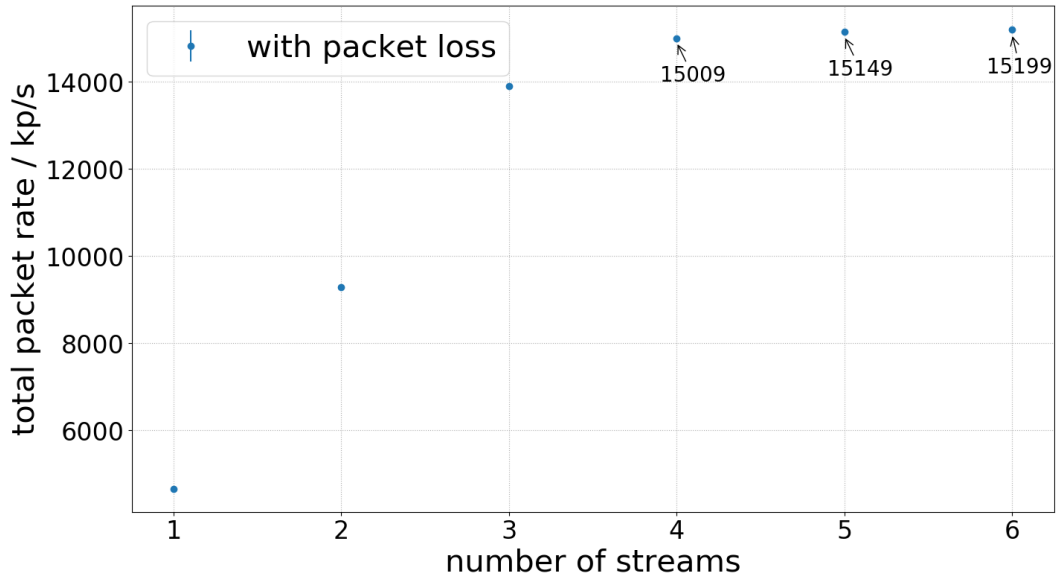


Figure 6.24: Measurement of the maximum packet rate achieved with different numbers of data streams [56]

It is therefore concluded, that there is a single element in each receive queue that limits the performance of the whole chain. As the difference between enabled and disabled processing lies in the time, a particular frame in the packet buffer is occupied, this limitation might come from the fact that not enough frames are available to store more incoming packets. Following the approach of holding all data in the local CPU cache, the packet buffer are limited in size. A larger buffer would also have only a limited effect if the software cannot process the packets as it was seen for the enabled data processing. With the combined packet rate of two streams of this test being already above the combined packet rates of the tests done in the third phase and the program in its current implementation being proved to work, this is regarded as a relative small issue and no changes in the code were done.

The number of data streams were further increased until the overall limits of the system were reached. As Figure 6.24 shows, this seems to be the case for four data streams when the packet rate got stuck at a value on the order of 15 Mpackets/s. With the packet processing being enabled, the same limit was seen with five data streams.

At this point, additional data streams were not increasing the overall packet rate to be seen by the receiving programs. With the network stack running roughly with twice the frequency and having eight times the bus width of a single data generator, the FPGA design can be excluded as the source of the limitation. Also, with an overall packet size of 110 B, the data rate on the network link was around 13.2 Gb/s, so the network itself (including the switch) should not be the cause.

Therefore, a component within the server is regarded as source for this. This could

be the network processor or the DMA controllers of the network card running at their maximum, but also the PCIe interface when being occupied by a huge amount of small transactions to be executed. In contrast to this, it is rather unlikely to be a software component of the Linux kernel as the whole processing should already be split up into processing queues. Thus, for example, the software interrupts are further distributed with increased number of streams.

With the maximum packet rate per data stream being already limited, the payload size was increased to 350 B, which corresponds to a packet rate of 3.5 Mpackets/s if no additional pause cycles are applied. This value was chosen to leave some headroom compared to the encountered limit in the previous test while still being able to reach the system limit with five data streams. Nevertheless, the next test series started with an artificially reduced packet rate due to the insertion of extra pause cycles. During the tests, the number of pause cycles is reduced to increase the packet rate from test run to test run. This path was chosen to enable better control of the packet rate while approaching the system limit.

As shown in Figure 6.25, the first four runs did not detect any packet loss within 60 min of runtime each. With the packet rate increased further up to 15 Mpackets/s, packets started to be dropped, keeping the packet rate at a level slightly above of 12 Mpackets/s.

This a bit of a surprise as the limit was found to be higher before and was rather attributed to the number of packets, as the data rate was far from the line rate. The only explanation for this behavior is that the maximum packet rate depends on the

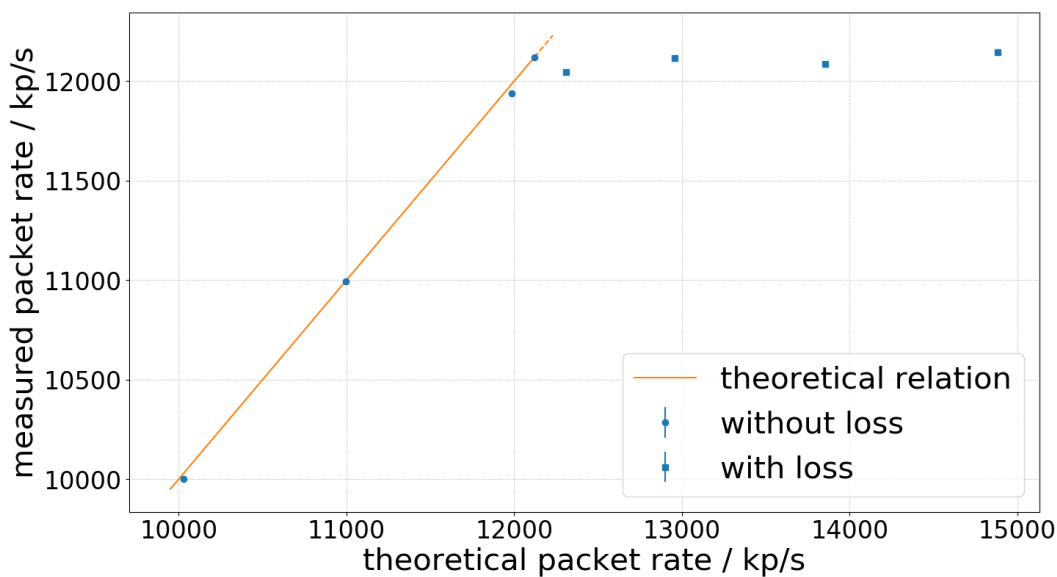


Figure 6.25: The measurement of the maximum packet rate was repeated with an increased packet size [56]. The result was a lower maximum packet rate as it was achieved with smaller packets.

packet size. Therefore, the reason for this limitation is assumed to be a hardware component having a very poor efficiency in case of small network packets.

To further investigate this dependency, several test runs were conducted to determine the maximum packet rates for different packet sizes. Until now, the packet size was defined by the payload size. As the receiving chain sees whole packets, the payload sizes are now chosen in a way to form packets of specific size. This leads to quite odd payload sizes.

At first, the range of packet sizes was tested in rather coarse step sizes. In this way, larger steps were found at packet sizes which are a power of two up to 2048 B. This upper limit was selected as XDP does not allow packets with a size of 4096 B. The lower limit of these test was set to a value of 64 B as it is the minimum length of ethernet frames.

Next to the steps at a packet size of a power of two, smaller steps were found at multiples of 64 B (e.g. 192 B or 320 B). These were not tested for the complete range as this would have taken too much time, but it is assumed that this behavior is also valid for larger packet sizes. Smaller step sizes were also not tested, so the 64 B might not be the original granularity of the steps. The reasons for these steps is still not clear. All steps found are shown in Figure 6.26.

This means, that the tests done in phase two and three used packets being only slightly below such a step. With a payload size of 2000 B, 4 B of DROP header and 42 B for the ethernet, IPv4 and UDP, the packets had a size of 2046 B. As the tests

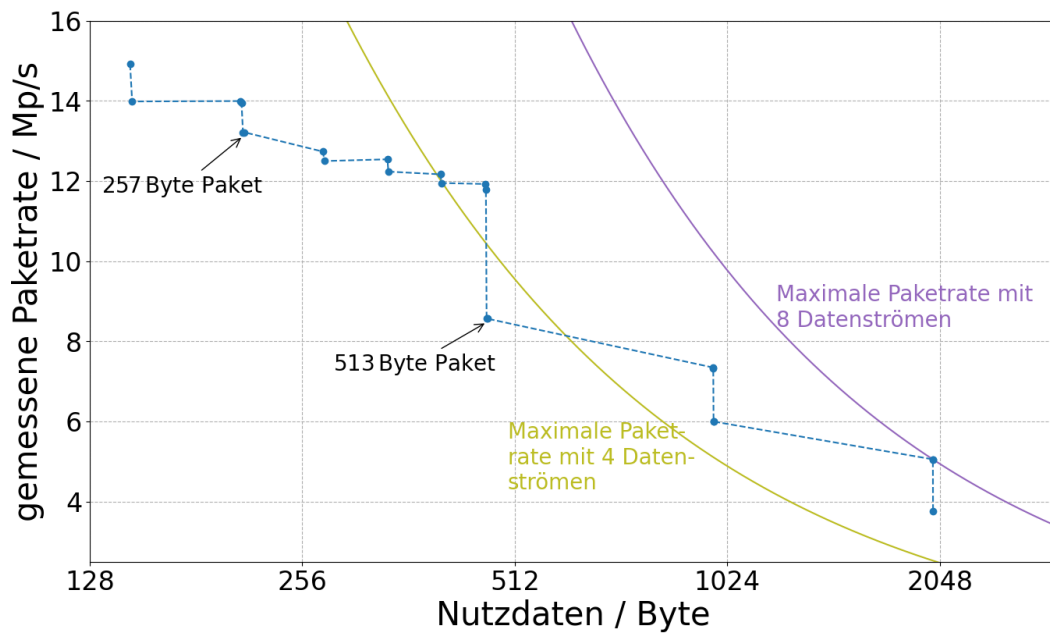


Figure 6.26: The measurements reported in Ref. [56] reveal several steps in the maximum packet rate depending on the packet size used. These were found at sizes being a power of two as well as general multiples of 64 B.

revealed a considerably drop in the maximum packet rate above of a packet size of 2048 B, the packet rate needed for the desired data rate would not been achievable if the DROP header would include a second header word. This confirms the choice made on the payload size for tests of phase two and three.

But this also confirms the proposal made in Section 2.5 about the usage of options of the DROP protocol. It was proposed to restrict the number of options to be used within a DROP packet to a single option and to omit any data in such a packet. Together with a maximum length for the option equal to the size of the data blocks, this should ensure that the desired data rate is achieved.

The dependence on the packet size means that the largest possible packet determines the maximum packet rate as a burst of these will slow down the whole processing chain. This might cause smaller packets following the larger ones with a higher rate to be dropped.

On the other side, there is also a lower limit on the packet size which is needed to achieve a desired data rate when a maximum packet rate is given. If the packet size is smaller, additional pauses needs to be inserted between the packets to limit the data rate. This size can be calculated by the Equations 4.1 and 4.2. Figure 6.27 shows this correlation.

As long as the packet size is smaller than the minimum for the desired data rate, some pauses are needed between the packets to not surpass the limit of the packet rate. With increasing packet size, larger parts of these pauses are replaced with data,

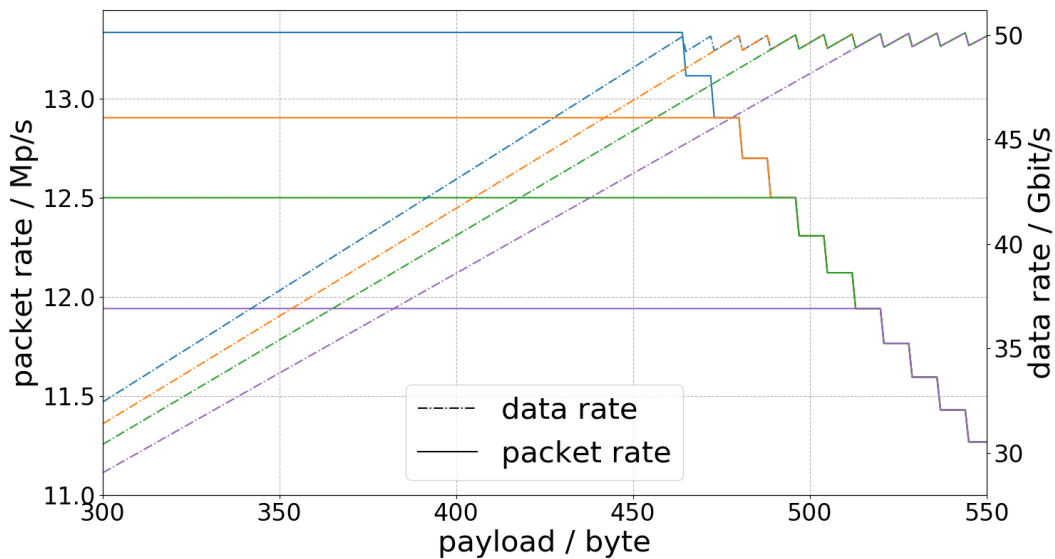


Figure 6.27: Correlation between packet and data rate for different limits for the maximum packet rate [56]. If the packet size is too small, additional pause cycles need to be inserted to not surpass a given packet rate. The steps seen on the right side are a result of the way how the data generator works.

leading to a point where no more pause remains. The packet rate starts to sink with clearly visible steps if the packet size is further increased beyond this point. These steps are a result of the data generators and their 8 B wide output signal. As the generation of a packet within the data generator is limited to integer values of clock cycles, eight consecutive packet sizes need the identical number of clock cycles to be generated and therefore result in the same packet rate.

While the packet rate is constant, the data rate is increasing linearly when the packet size is increased. The reason for this is that the data rate is simply the packet rate multiplied by the packet size. Thus, the data rate keeps increasing while within each step of the packet rate.

This correlation is still true when the no pauses are left. But with no pause cycles left to be filled with data, the additional clock cycle needed for the increased packet size leads to a decreased packet rate. With the step in the packet rate being larger than the increase of the packet size, the data rate decreases.

The reduction of the packet rate per step gets smaller with each step done as the relative size of the added clock cycle compared to the overall packet length gets smaller. Therefore, the data rate keeps growing overall. This behavior was verified by a measurement as shown in Figure 6.28.

To finish this phase of testing, a last set of tests were conducted to evaluate the maximum data rate that can be processed on a single CCX (i.e. two physical or four logical cores). For this, the best distribution of the different threads on a CCX was

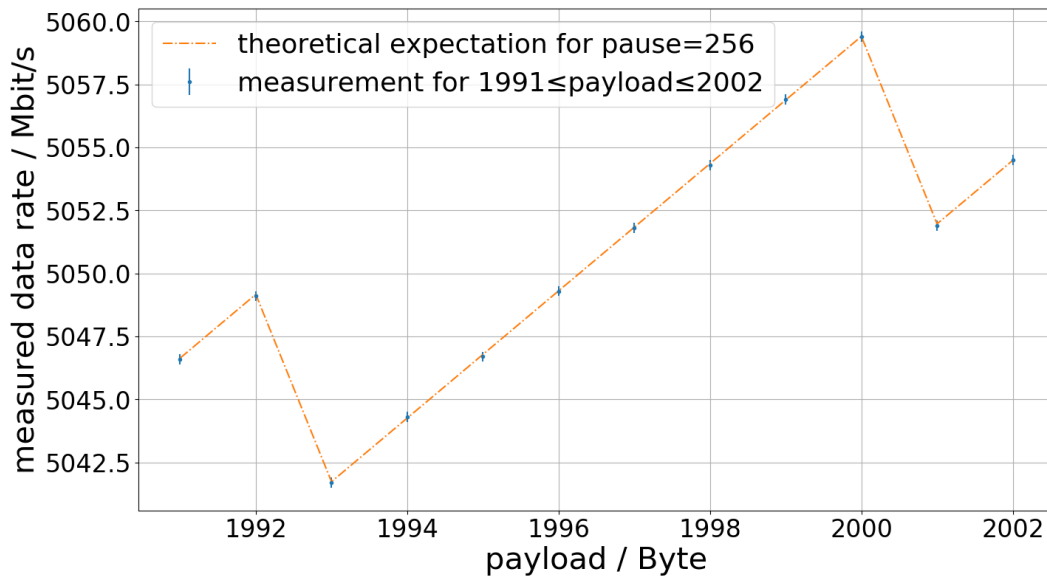


Figure 6.28: Measurement of the data rate for increasing payload sizes as presented in Ref. [56]. The result matches perfectly with the calculated values. Even with the step downwards in the packet rate every 8 B, the overall data rate is increasing.

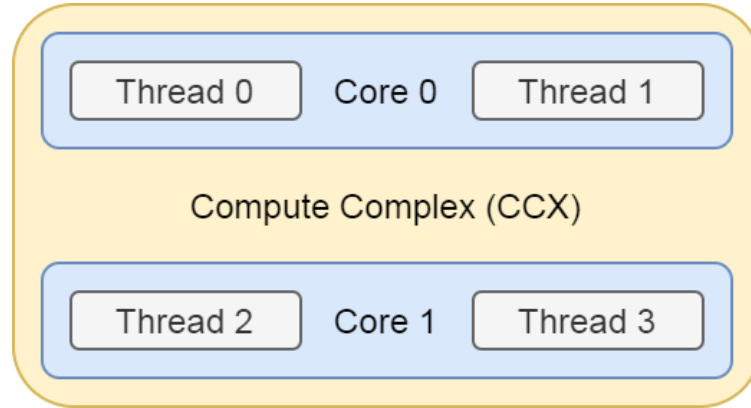


Figure 6.29: Numbering of the logical and physical cores within a core complex (CCX) as defined by [56]. The CCX are the building blocks of the new AMD CPUs.

determined for different numbers of data streams. As the number of combinations rises with the number of data streams (i.e. number of histogramming threads), the tests were started with a single data stream to find combinations with conflicting positions. These combinations were then used to derive rules for further tests as such combinations will also not work with multiple data streams. This helps to keep the overall effort manageable. It was also assumed, that the behavior of the data stream will be the same, meaning that switching the position of two histogrammer threads will not change the overall result.

With the position of the interrupt handling being fixed by the system configuration⁷ and the status thread generating a negligible workload⁸, only the positioning of the receive and histogrammer threads is varied. The logical cores are labeled as shown in Figure 6.29. The data streams ran with a data rate of 10 Gb/s each if not stated otherwise.

The main result with a single data stream proved what was quite obvious. As the histogramming thread has the highest CPU utilization (about 85 % of a logical CPU core) of all threads used, it was unable to cope with the data rate as soon as it had to share a logical core with the interrupt handler or the receive thread. All other combinations showed no packet loss.

The conflicting combination revealed no general pattern in terms of usage of specific cores. While the combinations with both the receiving and histogrammer thread running on the logical cores 2 or 3 (i.e. the second physical core) had less packet loss as both running on the logical core 1 (i.e. the same physical core as the software interrupt), it was still higher than the combination where the histogrammer ran together with the software interrupt on logical core 0 and the receive thread on

⁷It is set to use thread 0 of each CCX.

⁸Besides being also fixed in position.

logical core 1 (i.e. all threads on the same physical core)⁹.

Therefore, only a single rule was derived from this test:

- A histogramming thread should not share a logical core with any other thread.

The next round of tests was done with two data streams being enabled, adding up to six different combinations according to the aforementioned rule. This resulted in no combination being able to cope with the data rate as all lost at least some packets.

The highest packet loss was observed with both histogrammers running on the second physical core (i.e. the logical cores 2 and 3), with only little difference whether the receiving thread was running on logical core 0 or 1.

In contrast to the previous test, this one gives a clear indication about the cores as having the two threads with the highest utilization on the same physical core resulted in the worst performance. The packet loss for these were on the order of having all three threads of the previous test on the same logical core. This does not seem to be caused by the individual memory interfaces of each core as both histogrammers lose approximately the same fraction of packets. It is more likely that a shortage of components being shared between the two logical cores is the reason. If this is the case, it would be a first limitation found with respect to the CPU architecture.

The least packet loss was achieved with the two combination of both histogramming threads running on different physical cores (i.e. the first one running on logical core 1 while the second histogramming and the receive thread ran on thread 2 and 3). As Figure 6.30 shows, the reason for the packet loss was in both cases a CPU utilization of 100 % for the first histogrammer.

A major result here is that both histogramming threads did not behave the same as it was assumed. Therefore, the tests for the two cases with the lowest packet loss were repeated with swapped receive queues. This test showed, that the high CPU usage is a property of the first histogramming thread independent on the configured receive queue. Nevertheless, even with both histogrammers not behaving the same, the assumption of exchangeability of both data streams is still valid. Thus, the tests of previous test runs were not repeated.

The addition of the second data stream seems to have a significant impact on the communication between the different threads as the two combinations with the least

⁹The lowest packet loss of the conflicting combinations was achieved with the histogrammer running together with the software interrupt on logical core 0 and the receive thread on logical core 2.

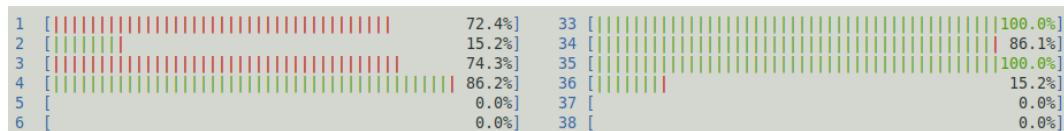


Figure 6.30: CPU utilization for two scenarios with two data streams each [56]. The first histogrammer reached a value of 100 % in both scenarios (i.e. cores 33 and 35) and is therefore not able to handle the data rate.

packet loss see a CPU utilization of 100 % for the first histogrammer while the same setup without the second data stream had a CPU utilization of 85 %. The work of counting data words is the same for both cases (as the data rate was not changed) and the execution of the histogrammer thread is paused if no new packet is available (i.e. the work queue is empty), which result in not reaching the 100 % level. Only the loop checking the synchronization flag is able to eat up the remaining CPU resources if the lock cannot be acquired in a timely manner. Therefore, the synchronization of the threads should be improved in future versions.

A different explanation for this behavior could be the same problem as seen with the two histogrammers running on the second physical core. As the software interrupts of the second data stream came on top of the first one, the load on the first logical core is also higher as in the scenario without the second data stream. Therefore, a shortage in shared components like the load or store units could also slow down the histogrammer thread on the same physical core, resulting in not being able to handle the data rate.

To resolve this issue, the data rates for both streams were reduced until no packet loss is detected anymore. For the combination with the receiving thread running on core 2, this resulted in a data rate of 7.5 Gb/s for the first stream and 8 Gb/s for the second. Any other combination was still losing packets at these data rates. Thus, the best combination with two data streams achieved a combined data rate of 15.5 Gb/s. As the first data stream started to lose packets again when the data rate of the second stream was slightly increased, this might point to the second explanation mentioned before.

With the receiving thread only using about 15 % of the CPU core, a third data stream was added to this logical core in a try to reach a higher total data rate. It would be clear, that this would also have only a reduced data rate, but might get some surplus data rate in the end.

The result was that the third data stream introduced again packet loss for the first two data streams, so that the combined data rate had to be lowered. This is little surprising. But the overall data rate settled again at the same order as it was reached with only two data streams. Therefore, the usage of three data streams is not recommended.

As a result, the maximum data rate to be processed on a single CCX was evaluated to be 15.5 Gb/s. Based on this findings, using six out of the eight CCXs contained in the CPU is enough to make full usage of the 100 Gb/s NIC (including some headroom left for other network traffic). But as the limits of the CPU architecture might be hit already, this is not recommended as it might be not stable enough in the long term.

Therefore, the usage of a single data stream per CCX is chosen as the best solution, even if it does not allow for the full usage of the network interface. But this solution leaves some headroom for the decoding of the data structure of the real FE chips.

Chapter 7

Summary and Outlook

In this thesis, a new front-end (FE) interface system called FRontEnD Data InterfacE (FREDDIE) was designed and implemented to connect the FE chips of the future ATLAS pixel detector with the data acquisition system. Due to the link from the Trigger, Timing and Control (TTC) system still not being defined and only early prototypes being available for the FE chips, the focus of the implementation was on the network link from the FE interface system towards the processing software.

For this network link, an ethernet based network stack supporting a line rate of up to 100 Gb/s was implemented within an FPGA and tested intensively by making use of specifically designed data generators. Based on the design choice of using several network interfaces within a single FPGA, the question of guaranteed data transmission arose, since the implementation of network protocols like the Transmission Control Protocol (TCP) would result in a very complex FPGA design. This dilemma can be circumvented if the rate of dropped network packets is reduced to an acceptable level. Therefore, reaching a very low rate of dropped network packets became the main topic of this thesis.

The investigations started using the User Datagram Protocol (UDP) to determine the actual rate of dropped packets. However, UDP itself is missing an ordering structure and could thus not give any information about the distribution of dropped packets. Therefore, a new protocol called Data ReadOut Protocol (DROP) was developed, adding a packet identifier to the header. In addition, the newly developed protocol offered optional extensions for mitigating the packet drop if it would be close to but still above of an acceptable level.

While first results showed a rate of dropped to received packets of 0.008 %, this was still considered as being too high. With a packet rate of several Mpackets/s, this would result in a few hundred packets being dropped per second and network link. The distribution of the dropped packets over time showed that the packets are mostly dropped in batches. The size of these batches rendered the proposed FEC extension of the DROP protocol impossible.

At the same time, a new technique in Linux network processing was emerging, named eXpress Data Path (XDP). It offers a greatly reduced processing overhead and therefore helps reducing packet loss. To take advantage of this new Linux feature, the receiving software was adapted accordingly.

But this new technique also brought new challenges and restrictions, most notably a maximum packet size of 3498 B. After these issues were solved or circumvented,

it enabled a huge reduction of packet loss, resulting in packets being dropped only sporadically. These results encouraged the search for the reasons of these, which was found in the packet management of the used linux distribution. While it is still unclear how this affected the system, disabling the automatic updates proved it to be the cause. With this being fixed, a lossless data transmission was achieved. In the validation test, 2.92×10^{12} packets (i.e. 5.2 PB) were transferred within 168 h (i.e. a week) without a single missing packet.

To validate the results, the software was extended by a payload processing step, counting the occurrence of each 16 b word. Next to proving the correct transmission, this scenario is also closer to the final application. With this feature being implemented, the measurements were repeated with the same result of no packet being dropped. This test thereby also proved the result to be reproducible. Therefore, the base assumption that the packet loss can be brought to an acceptable level was not only confirmed but even surpassed.

After this prove of concept was done, the measurements were rounded up by measurements with a broader parameter range while the aforementioned tests had concentrated on a single set of parameters. These additional investigations showed some surprising results. One result of these is that the overall packet rate the test system was able to process depends on the used packet size. Another result was the hint that the performance of a particular thread of the receive software might get reduced by a different thread running on the other logical core of the same physical core. This means that some limitation of the CPU architecture were found.

Therefore, the suggested scenario is to process a single data stream per CCX, leaving some headroom for additional processing steps on the transferred data. This will be needed as the data processing in the tests was purely based on the counting of 16 b words, while a more realistic use case would at least include some kind of address decoding or index calculation.

Before measurements can be repeated with real data from the RD53A chips, a way to generate a constant data stream with a predefined data rate needs to be implemented. This applies in particular to the configuration of the actual chips as well as the TTC emulator, which is currently missing.

Since the performance of CPUs will continue to increase with each generation and since the completion of the ITk detector is at least five years away, the processing power of the final servers might be enough to allow for higher data rates per CPU for the decoding of the real detector data.

But the protocol of the FE chips will change in the next version, getting more complex to achieve a better compression rate. This also means, that the decoding in the software will get more complex. It needs to be seen if this can be done with the remaining CPU resources.

When the software is adapted for realistic data, there are some more changes that should be considered. First of all comes the improved synchronization between the different threads to reduce the overhead for this. Also, the possibility to distribute the processing of a single data stream over two processing threads is missing and might be needed for the next generation of FE chips. In this context, it should

also be investigated if the BPF program could be loaded outside of the initialization process and with a custom program. This would ease the testing process as the receive program would not need to check for this.

On the side of the FPGA system, the interface for the FE chips need to be scaled up to fill the full network bandwidth. As there are not enough electrical connectors available to connect each FE chip directly, a data aggregator is needed. The natural choice would be the lpGBT as it will be used in the final detector. But this uses optical transmission, so the whole detector interface will be changed.

In this process, it might be necessary to rewrite the network stack in a less resource intensive way as proposed in this thesis. This should also ensure the further usage if no additional conversion modules are available anymore for future FPGA generations.

Acknowledgments

Many people helped me and were involved in different ways in order to complete this work. I would like to thank them all.

My thanks go to Professor Wolfgang Wagner, my supervisor at the University of Wuppertal. He put his trust in me, even if it took longer as initially planned. Without his advise and counsel, this work would not been possible.

My thanks also go to Professor Dietmar Tutsch, who made it possible for me to do my doctorate in electrical engineering.

Many thanks go also to Tobias Flick, who guided me through the years in Wuppertal. His experience with the detector readout, contacts and knowledge of the experiment was a great help. It was a great pleasure to work with him.

Another big thank you goes to Marius Wensing for fruitful discussions on FPGA designs and for administration of the laboratory systems.

Special thanks also go to Niklaus Lehmann, who was always available with a helping hand even outside of working hours, for exploring the surroundings of several conference locations and for co-hosting the town hall meetings.

Also thanks to the students who worked for me and performed measurements used in this work. These are specifically: Marvin Geyik, for volunteering happily for all kinds of tasks to be done. Maren Stratmann, for implementing the 8 b version of the network stack together with Marvin. Marcel Strohmeyer, for intensive tests of the first version of the network stack. Timo Göhring, for performing the investigations on a broader parameter set.

I thank them all for their hard work.

Further, I'd like to thank Kerstin Lantsch and Jens Dopke for the great insights into the old readout systems. Thanks to Kerstin I was also able to keep my head in situations where it might be teared off otherwise.

I'd like to thank Jens Krüger for extensive support in various situations, especially the career start and moving to Wuppertal. Without him I would probably not have become an FPGA expert.

Many thanks to family and friends who were interested in my work and listened to my explanations. Without the support from my parents, this would not have been possible. They encouraged me in all my goals and during the long years of my studies. Thank you very much for everything!

Bibliography

- ¹K. Akiyama, K. Bouman, and D. Woody, “First m87 event horizon telescope results. i. the shadow of the supermassive black hole”, *Astrophysical Journal Letters* **875** (2019) (cit. on p. 1).
- ²B. P. Abbott et al., “Observation of gravitational waves from a binary black hole merger”, *Phys. Rev. Lett.* **116**, 061102 (2016) (cit. on p. 1).
- ³N. Madsen and G. Stutter, “Observation of the 1s2s transition in trapped antihydrogen”, *Nature* **541**, 10.1038/nature21040 (2016) (cit. on p. 1).
- ⁴CERN, *The Large Hadron Collider*, <https://home.cern/science/accelerators/large-hadron-collider> (visited on 01/22/2021) (cit. on p. 1).
- ⁵CMS Collaboration, “Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC”, *Physics Letters B* **716**, 30–61 (2012) (cit. on p. 1).
- ⁶ATLAS Collaboration, “Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC”, *Physics Letters B* **716**, 1–29 (2012) (cit. on p. 1).
- ⁷ATLAS Collaboration, *Technical Design Report for the ATLAS Inner Tracker Pixel Detector*, tech. rep. CERN-LHCC-2017-021. ATLAS-TDR-030 (CERN, Geneva, Sept. 2017) (cit. on pp. 2, 13, 14).
- ⁸M. Brice, “Aerial View of the CERN taken in 2008.”, July 2008 (cit. on p. 5).
- ⁹L. Evans and P. Bryant, “LHC machine”, *Journal of Instrumentation* **3**, S08001–S08001 (2008) (cit. on p. 6).
- ¹⁰R. Alemany-Fernandez et al., “Operation and Configuration of the LHC in Run 1”, (2013) (cit. on p. 6).
- ¹¹CERN, *The HL-LHC project*, <https://hilumilhc.web.cern.ch/content/hl-lhc-project> (visited on 06/06/2020) (cit. on pp. 6, 174–176).
- ¹²R. Bailey and P. Collier, *Standard Filling Schemes for Various LHC Operation Modes*, tech. rep. LHC-PROJECT-NOTE-323 (CERN, Geneva, Sept. 2003) (cit. on p. 7).
- ¹³ATLAS Experiment, *$Z \rightarrow \mu\mu$ candidate event with 65 additional reconstructed primary vertices recorded in 2017*. https://twiki.cern.ch/twiki/bin/view/AtlasPublic/EventDisplayRun2Physics#High_pileup_displays (visited on 09/22/2020) (cit. on p. 8).

- ¹⁴Wikipedia, *Standard Model of Elementary Particles*, (Sept. 2019) https://commons.wikimedia.org/wiki/File:Standard_Model_of_Elementary_Particles.svg (visited on 06/03/2020) (cit. on p. 9).
- ¹⁵J. Pequeno and P. Schaffner, “How ATLAS detects particles: diagram of particle paths in the detector”, Jan. 2013 (cit. on p. 10).
- ¹⁶ATLAS Collaboration, “The ATLAS experiment at the CERN large hadron collider”, *Journal of Instrumentation* **3**, S08003–S08003 (2008) (cit. on pp. 11, 12).
- ¹⁷ATLAS IBL Collaboration, “Production and integration of the ATLAS Insertable B-Layer”, *Journal of Instrumentation* **13**, T05008 (2018) (cit. on p. 11).
- ¹⁸ATLAS Collaboration, *Technical Design Report for the ATLAS Inner Tracker Strip Detector*, tech. rep. CERN-LHCC-2017-005. ATLAS-TDR-025 (CERN, Geneva, Apr. 2017) (cit. on p. 13).
- ¹⁹P. Vankov, *ATLAS Upgrade for the HL-LHC: meeting the challenges of a five-fold increase in collision rate*, tech. rep. ATL-UPGRADE-PROC-2012-003. ATL-UPGRADE-PROC-2012-003, Comments: Presented at the 2011 Hadron Collider Physics symposium (HCP-2011), Paris, France, November 14-18 2011, 3 pages, 3 figures (CERN, Geneva, Jan. 2012) (cit. on p. 13).
- ²⁰ATLAS Collaboration, *Technical Design Report for the Phase-II Upgrade of the ATLAS TDAQ System*, tech. rep. CERN-LHCC-2017-020. ATLAS-TDR-029 (CERN, Geneva, Sept. 2017) (cit. on pp. 14, 19).
- ²¹ATLAS FELIX Group, *The ATLAS FELIX Project*, <https://atlas-project-felix.web.cern.ch/atlas-project-felix/> (visited on 02/17/2020) (cit. on p. 17).
- ²²ATLAS Collaboration, *ATLAS TDAQ System Phase-I Upgrade Technical Design Report*, tech. rep. CERN-LHCC-2013-018. ATLAS-TDR-023 (CERN, Geneva, Nov. 2013) (cit. on p. 18).
- ²³ATLAS FELIX Group, *FELIX Phase-1 Technical Specification and Implementation, Final Design Review*, tech. rep. (CERN, Geneva, Mar. 2018) (cit. on p. 18).
- ²⁴*Information technology Open Systems Interconnection Basic Reference Model: The Basic Model*, Standard (International Organization for Standardization, Geneva, CH, July 1994) (cit. on p. 23).
- ²⁵J. Postel, *INTERNET PROTOCOL - DARPA INTERNET PROGRAM - PROTOCOL SPECIFICATION*, RFC 791 (Sept. 1981) (cit. on p. 25).
- ²⁶*IEEE Standard for Ethernet*, Standard (IEEE, Aug. 2018) (cit. on pp. 26, 27, 88).
- ²⁷D. Plummer, *An Ethernet Address Resolution Protocol*, RFC 826 (Nov. 1982) (cit. on p. 27).
- ²⁸J. Postel, *TRANSMISSION CONTROL PROTOCOL - DARPA INTERNET PROGRAM - PROTOCOL SPECIFICATION*, RFC 793 (Sept. 1981) (cit. on p. 28).

- ²⁹Sergiodc2, Marty Pauley, Scil100, *State transition diagram of a TCP/IP Socket*, https://commons.wikimedia.org/wiki/File:Tcp_state_diagram_fixed_new.svg (visited on 09/30/2020) (cit. on p. 31).
- ³⁰R. Zaghal and J. Khan, “Efsm/sdl modeling of the original tcp standard (rfc793) and the congestion control mechanism of tcp reno”, (2005) (cit. on p. 31).
- ³¹S. Mills and S. Parkes, “TCP/IP Over SpaceWire”, in *Dasia 2003 - data systems in aerospace*, Vol. 532, ESA Special Publication (Jan. 2003), p. 53.1 (cit. on p. 31).
- ³²W. Stevens, *TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms*, RFC 2001 (Jan. 1997) (cit. on pp. 33, 35–37).
- ³³J. Postel, *User Datagram Protocol*, RFC 791 (Aug. 1980) (cit. on p. 37).
- ³⁴L. Tlustos, “Performance and limitations of high granularity single photon processing X-ray imaging detectors”, Presented on 1 Apr 2005 (2005) (cit. on p. 55).
- ³⁵H. Spieler, *Semiconductor Detector Systems* (Oxford Science Publications, 2005) (cit. on p. 56).
- ³⁶M. Garcia-Sciveres, *The RD53A Integrated Circuit*, tech. rep. CERN-RD53-PUB-17-001 (CERN, Geneva, Oct. 2017) (cit. on pp. 57–61).
- ³⁷XILINX, *Aurora 64B/66B Protocol Specification*, https://www.xilinx.com/support/documentation/ip_documentation/aurora_64b66b_protocol_spec_sp011.pdf (visited on 07/27/2020) (cit. on p. 61).
- ³⁸G. Project, *lpGBT link architecture*, <https://espace.cern.ch/GBT-Project/LpGBT/default.aspx> (visited on 10/15/2020) (cit. on p. 64).
- ³⁹T. F.-t. Heim and B. Gallop, *ITk DAQ Requirements*, tech. rep. ATL-COM-ITK-2018-018 (CERN, Geneva, Mar. 2018) (cit. on p. 68).
- ⁴⁰ARM, *AMBA Specifications*, <https://developer.arm.com/architectures/system-architectures/amba/specifications> (visited on 04/15/2020) (cit. on p. 74).
- ⁴¹Gigabyte, *R282-Z93 (rev. 100) product page*, <https://www.gigabyte.com/us/Rack-Server/R282-Z93-rev-100> (visited on 10/30/2020) (cit. on p. 102).
- ⁴²AMD, *AMD EPYC 7302 product page*, <https://www.amd.com/de/products/cpu/amd-epyc-7302> (visited on 11/03/2020) (cit. on p. 102).
- ⁴³Mellanox, *ConnectX-5 EN Card product brief*, <https://www.mellanox.com/files/doc-2020/pb-connectx-5-en-card.pdf> (visited on 11/03/2020) (cit. on p. 102).
- ⁴⁴Timothy Prickett Morgan, *A Deep Dive Into AMDs Rome Epyc Architecture*, <https://www.nextplatform.com/2019/08/15/a-deep-dive-into-amds-rome-epyc-architecture/> (visited on 11/06/2020) (cit. on pp. 102, 103).
- ⁴⁵Intel Corporation, *Intel[®] Ethernet Controller I350 Datasheet*, <http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/ethernet-controller-i350-datasheet.pdf> (visited on 12/18/2020) (cit. on p. 105).

- ⁴⁶fs.com, *SFP vs SFP+ vs SFP28 vs QSFP+ vs QSFP28, What Are the Differences?*, <https://community.fs.com/blog/sfp-vs-sfp-vs-sf-p28-vs-qsfp-vs-qsfp-p28-what-are-the-differences.html> (visited on 09/17/2020) (cit. on p. 105).
- ⁴⁷Jamal Hadi Salim (Znyx Networks), Robert Olsson (Uppsala University/Swedish University of Agricultural Sciences) Alexey Kuznetsov (Swsoft/INR), “Beyond Soft-net”, *5th Annual Linux Showcase & Conference (2001)* (cit. on p. 105).
- ⁴⁸packagecloud, *Monitoring and Tuning the Linux Networking Stack: Receiving Data*, <https://blog.packagecloud.io/eng/2016/06/22/monitoring-tuning-linux-networking-stack-receiving-data/> (visited on 09/17/2020) (cit. on p. 106).
- ⁴⁹The Linux Foundation, *DPDK, Data Plane Development Kit*, <https://www.dpdk.org/> (visited on 11/18/2020) (cit. on p. 106).
- ⁵⁰P. Emmerich et al., “User Space Network Drivers”, in *Acm/ieee symposium on architectures for networking and communications systems (anncs 2019)* (Sept. 2019) (cit. on p. 107).
- ⁵¹KernelNewbies, *Linux 4.8 has been release*, https://kernelnewbies.org/Linux_4.8#Support_for_eXpress_Data_Path (visited on 10/21/2020) (cit. on p. 107).
- ⁵²Tom Herbert, Alexei Starovoitov, *eXpress Data Path (XDP), Programmable and high performance networking data path*, https://github.com/iovisor/bpf-docs/blob/master/Express_Data_Path.pdf (visited on 10/21/2020) (cit. on p. 107).
- ⁵³Cilium Authors, *BPF and XDP Reference Guide*, <https://docs.cilium.io/en/latest/bpf/> (visited on 10/21/2020) (cit. on p. 107).
- ⁵⁴Jonathan Corbet, *Accelerating networking with AF_XDP*, <https://lwn.net/Articles/750845/> (visited on 10/21/2020) (cit. on p. 109).
- ⁵⁵IANA, *Service Name and Transport Protocol Port Number Registry*, <https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml> (visited on 12/07/2020) (cit. on p. 110).
- ⁵⁶T. Gohring, “Performance Measurements of High-Bandwidth Transmission between FPGAs and Server Computers”, Presented 28 Sep 2020 (Bergische Universität Wuppertal, Sept. 2020) (cit. on pp. 114, 141–150).
- ⁵⁷Matt Godbolt, *Compiler Explorer*, <https://godbolt.org> (visited on 12/08/2020) (cit. on p. 118).
- ⁵⁸M. Strohmeier, “Evaluation von High-Speed Ethernet Schnittstellen für das Auslesesystem des Atlas Inner Trackers” (Fachhochschule Münster, Oct. 2017) (cit. on pp. 124, 126–131).

Acronyms

ADC	analog to digital converter. 65
AIMD	Additive Increase/Multiplicative Decrease. 37
AMBA	Advanced Microcontroller Bus Architecture. 74
API	application programming interface. 23
ARP	address resolution protocol. 27, 87, 88
ASIC	application specific integrated circuit. 64, 66
AXI	Advanced eXtensible Interface. 74, 88, 89, 93, 94, 168
BCID	Bunch Counter ID. 76, 77
BCR	Bunch Counter Reset. 76, 77
BER	bit error rate. 124
BPF	Berkeley Packet Filter. 107–109, 114, 117, 134, 155
CCD	core complex die. 102, 103, 168
CCX	core complex. 103, 104, 111, 112, 136, 148, 149, 151, 154, 168
CERN	European Center for Nuclear Research. 1, 5
CET	Central European Time. 137
CPU	central processing unit. 123, 135, 136, 138, 144, 149–151, 154
CRC	cyclic redundancy check. 105
DCS	detector control system. 65
DMA	direct memory access. 94, 105, 145
DoS	Denial of Service. 108
DPDK	Data Plane Development Kit. 106

- DROP** Data ReadOut Protocol. 45–48, 51, 74, 87, 94–96, 112, 113, 117, 119, 120, 123, 132, 133, 135, 140, 146, 147, 153
- EAL** Environment Abstraction Layer. 107
- ECR** Event Counter Reset. 76, 77
- FE** front-end. iii, 15, 17, 18, 46, 55–70, 73, 74, 76–80, 83–87, 94, 95, 97, 130, 151, 153–155, 167
- FEC** forward error correction. 15, 41, 43, 51, 52, 96, 133, 153
- FELIX** Front-End LInk eXchange. 17–19
- FF** flip flop. 91
- FIFO** first-in/first-out. 77, 83–87, 105
- FMC** FPGA Mezzanine Card. 73, 76, 79, 80
- FPGA** field programmable gate array. iii, 2, 3, 17–20, 46, 51, 64, 69, 70, 73, 79, 80, 82, 91, 93–96, 104, 115, 123–125, 127–132, 135, 136, 138, 142–144, 153, 155, 157
- FREDDIE** FRontEnD Data InterfacE. 73, 74, 76, 77, 79, 80, 86, 91, 94, 97, 130, 153
- FSM** finite state machine. 30
- GPIO** general purpose input/output. 70, 79, 80
- HBM** High Bandwidth Memory. 70
- HL-LHC** high luminosity LHC. iii, 13, 174
- HPC** high performance computing. 43
- HTTP** Hypertext Transfer Protocol. 39, 40
- IBA** InfiniBand Architecture. 42
- IBL** insertable B-layer. 11, 15, 16, 58
- IC** integrated circuit. 17
- ICMP** Internet Control Message Protocol. 47, 87, 88, 106
- ID** inner detector. 11, 13, 14, 17
- IP** interaction point. 7, 11, 14

- IPv4** Internet Protocol version 4. 25–27, 29–31, 37, 38, 41, 42, 46, 47, 73, 87–89, 93–96, 110, 117, 128–130, 146
- IPv6** Internet Protocol version 6. 88
- ITk** inner tracker. iii, 13–15, 19, 55–57, 64, 68, 76, 77, 154
- L0A** Level 0 Accept. 76, 77
- L0ID** Level 0 ID. 76, 77
- LHC** Large Hadron Collider. iii, 1, 2, 5–7, 11, 13, 17, 76, 82, 138, 171, 174
- MAC** Media Access Control. 42, 88, 92–94, 130
- MSS** Maximum Segment Size. 30, 35, 36
- MTU** maximum transmission unit. 111
- NIC** network interface card. 102, 105–111, 114, 125, 136, 138, 151
- NUMA** Non-Uniform Memory Access. 102
- OSI** Open Systems Interconnection. 23, 24, 87, 171
- P2P** peer-to-peer. 63
- PCB** printed circuit board. 69
- PLL** phase locked loop. 76
- PON** passive optical network. 63
- QoS** Quality of Service. 45
- QSFP+** quad small form-factor pluggable plus. 105
- RDMA** Remote Direct Memory Access. 42
- RTT** Round-Trip Time. 36, 40, 43, 96, 125
- SCC** single chip card. 73
- SCT** semiconductor tracker. 11
- SFP+** small form-factor pluggable plus. 73, 105
- SKB** socket buffer. 106

Acronyms

- SM** Standard Model of elementary particle physics. 1, 8–10
- SMT** simultaneous multithreading. 103
- TCP** Transmission Control Protocol. 2, 27–41, 44–46, 107, 108, 153, 167
- ToT** Time over Threshold. 57
- TRT** transition radiation tracker. 11, 14
- TTC** Trigger, Timing and Control. 18, 62, 63, 68, 69, 73, 76–79, 153, 154
- UDP** User Datagram Protocol. 2, 37–39, 45–47, 73, 88, 90, 91, 93, 95, 96, 110, 113, 117, 128, 129, 140, 146, 153
- UTC** Coordinated Universal Time. 137
- VoIP** Voice over IP. 39
- VPN** Virtual Private Network. 39
- XDP** eXpress Data Path. iii, 3, 101, 107–111, 114, 123, 134, 146, 153, 168
- XSK** XDP socket. 108, 109, 114

List of Figures

1.1	LHC aerial view	5
1.2	LHC bunch structure	7
1.3	Pile-up at LHC	8
1.4	Elementary particles of the Standard Model	9
1.5	Identifying particles	10
1.6	Overview of the ATLAS detector	12
1.7	ITk detector with pile up of 230	13
1.8	Layout of the ITk detector	14
1.9	First prototype FE on test card	15
1.10	Comparison of SiROD and the IBL version	16
1.11	Architecture of the Phase-I FELIX	18
1.12	Proposed Phase-II network architecture	19
2.1	IPv4 Header	25
2.2	Ethernet frame structure	26
2.3	TCP Header	28
2.4	TCP Checksum	29
2.5	TCP finite state machine	31
2.6	Establishment and closing of a TCP connection	32
2.7	TCP data streams	33
2.8	TCP receive window	35
2.9	TCP slow start behavior	36
2.10	UDP Header	37
2.11	UDP Checksum	38
2.12	QUIC streams	41
2.13	DROP mapping	46
2.14	DROP streams	48
2.15	DROP options	50
2.16	DROP reliability options	51
2.17	DROP FEC stream	52
3.1	Silicon pixel sensor with bump-bonded FE chip.	55
3.2	Composition of a Pixel module.	56
3.3	Analog front end electronics	56
3.4	RD53A Layout	57
3.5	Analog front-end flavors	58
3.6	RD53A trigger commands	59

3.7	RD53A command overview	60
3.8	RD53A output format	61
3.9	System overview	62
3.10	TTC topology variants	63
3.11	Overview of the GBT scheme	64
3.12	Link sharing of Pixel modules	65
3.13	Mask stepping	66
3.14	Calibration loop structure	67
4.1	Organization of the different firmware blocks	74
4.2	Overview of the application wrapper	75
4.3	Logical structure of the FE encoder	78
4.4	More detailed overview of the FE encoder	78
4.5	Alternative downlink structure	79
4.6	Simplified block diagram of the receive chain	80
4.7	Block diagram of the input stage	80
4.8	Delay scanning algorithm	81
4.9	Block diagram of the very basic receive chain	83
4.10	Block diagram of a protocol decoder for a single lane	84
4.11	Simulation of the alignment process	85
4.12	Block diagram of the final protocol decoder	86
4.13	Block diagram of the optimized protocol decoder of FREDDIE	87
4.14	Position of the network stack within the FPGA system	88
4.15	AXI-Stream interface	88
4.16	Overview of network stack	90
4.17	Transmit path handshake infrastructure	91
4.18	Alternative architecture of network stack	92
4.19	Configuration bus structure	94
4.20	Position of the DROP wrapper within the FPGA	95
4.21	Different pattern of the data generator	99
5.1	Block diagram of the <i>AMD EPYC 7302</i> processor. Only four out of eight CCDs are populated. Based on Ref. [44].	102
5.2	Block diagram of a CCX of the <i>AMD EPYC 7302</i> processor. Only two out of four cores are enabled, but all four L3 cache slices are available. Based on Ref. [44].	103
5.3	Packet reception in XDP	109
5.4	Interactions between the different software components	112
5.5	Internal structure of the receiving thread	116
5.6	Internal structure of the histogramming thread	119
5.7	Example output of the statistic thread	120
6.1	Overview test setup	124
6.2	Loopback setup used for the delay measurements	125

6.3	Latency measurement	126
6.4	Packets per second FPGA to FPGA	127
6.5	Packets per second FPGA-to-PC	128
6.6	Bandwidth comparison PC and FPGA	129
6.7	Bandwidth comparison	130
6.8	Bandwidth comparison absolute	131
6.9	Packet loss	131
6.10	Distribution of missing packets	133
6.11	Test result for 8000 B payload size	133
6.12	Initial problems of XDP	134
6.13	Early long-term test of XDP	134
6.14	Test result for doubled data rate	135
6.15	Test result for further increased data rate	135
6.16	Example of lost packets	137
6.17	Excerpt of the log file of dnf	137
6.18	Prove by provocation of an update	137
6.19	Results after disabled packet manager	138
6.20	Test of the error detection mechanism	139
6.21	First lines of a log file containig a histogram	140
6.22	Histogram of a flawless transmission	141
6.23	Histogram of a flawed data transmission	142
6.24	Measurement of the maximum packet rate	144
6.25	Repeated measurement with increased packet size	145
6.26	Steps found in the maximum packet rate	146
6.27	Correlation between packet and data rate	147
6.28	Verification of the steps seen with increasing payload size	148
6.29	Numbering of the cores within a CCX	149
6.30	CPU utilization with two histogrammers	150
A.1	Schedule of the HL-LHC	174
A.2	LHC beam parameter part 1	175
A.3	LHC beam parameter part 2	176

List of Tables

1.1	Overview LHC runs	6
2.1	Layers of the OSI model	24

Appendix A

LHC numbers

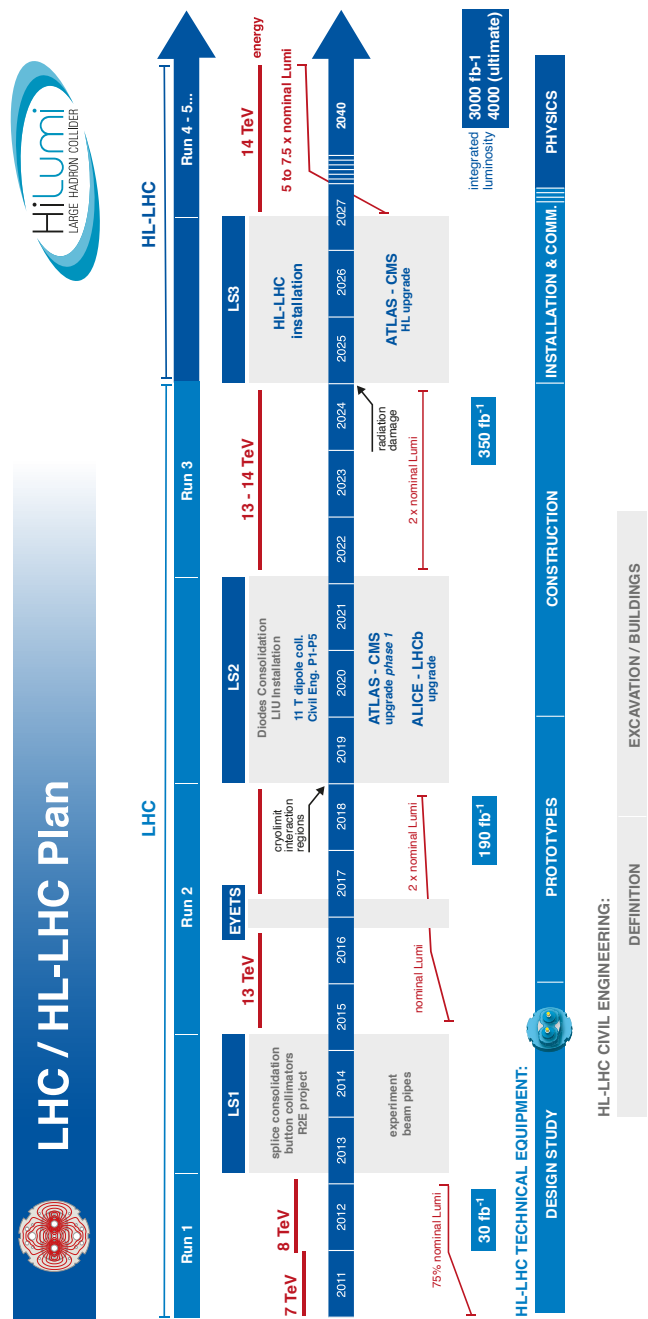


Figure A.1: Schedule for the LHC operation and upgrade to the HL-LHC [11]

Parameter	Nominal LHC sign Report]	[De-	HL-LHC 25ns [standard]	HL-LHC 25ns [BCMS] ^{see Note 9}	HL-LHC	8b+4e ^{see Note 10}
Beam energy in collision [TeV]	7		7	7	7	7
N_b	1.15×10^{11}		2.2×10^{11}	2.2×10^{11}	2.3×10^{11}	2.3×10^{11}
n_b	2808		2748	2604	1968	1968
Number of collisions in IP1 and IP5 ^{see Note 1}	2808		2736	2592	1960	1960
N_{tot}	3.20×10^{14}		6.00×10^{14}	5.70×10^{14}	4.50×10^{14}	4.50×10^{14}
beam current [A]	0.58		1.09	1.03	0.82	0.82
x-ing angle [μrad] ^{see Note 11}	285		590	590	554 ^{see Note 10}	554 ^{see Note 10}
beam separation [σ] ^{see Note 11}	9.4		12.5	12.5	12.5 ^{see Note 10}	12.5 ^{see Note 10}
β^* [m]	0.55		0.15	0.15	0.15	0.15
ϵ_n [μm]	3.75		2.50	2.50	2.2	2.2
ϵ_L [eVs]	2.5		2.5	2.5	2.5	2.5
r.m.s. energy spread	1.13×10^{-4}		1.13×10^{-4}	1.13×10^{-4}	1.13×10^{-4}	1.13×10^{-4}
r.m.s. bunch length [m]	7.55×10^{-2}		7.55×10^{-2}	7.55×10^{-2}	7.55×10^{-2}	7.55×10^{-2}
IBS horizontal [h]	$80 \rightarrow 106$		18.5	18.5	13.1	13.1
IBS longitudinal [h]	$61 \rightarrow 60$		20.4	20.4	17.6	17.6
Piwiński parameter	0.65		3.14	3.14	3.14	3.14
Total loss factor R_0 without crab cavity	0.836		0.305	0.305	0.304	0.304
Total loss factor R_1 with crab cavity ^{see Note 13}	0.981		0.829	0.829	0.828	0.828
beam-beam / IP without crab cavity	3.10×10^{-3}		3.3×10^{-3}	3.3×10^{-3}	3.90×10^{-3}	3.90×10^{-3}
beam-beam / IP with crab cavity ^{see Note 13}	3.8×10^{-3}		1.10×10^{-2}	1.10×10^{-2}	1.30×10^{-2}	1.30×10^{-2}
Peak Luminosity without crab cavity [$\text{cm}^{-2} \text{s}^{-1}$]	1.00×10^{34}		7.18×10^{34}	6.80×10^{34}	6.38×10^{34}	6.38×10^{34}
Virtual Luminosity with crab cavity: ^{see Note 13}	1.18×10^{34}		1.9×10^{35}	1.85×10^{35}	1.74×10^{35}	1.74×10^{35}
$\mathcal{L}_{\text{peak}} \cdot R_1 / R_0$ [$\text{cm}^{-2} \text{s}^{-1}$]	27		198	198	246	246
Events / crossing without levelling and without crab-cavity	-		5.00×10^{34}	5.00×10^{34}	3.63×10^{34}	3.63×10^{34}
Levelling Luminosity [$\text{cm}^{-2} \text{s}^{-1}$] ^{see Note 5}	-		138	146	140	140
Events / crossing (with levelling and crab cavities for HL-LHC) ^{see Note 5}	27		1.25	1.31	1.28	1.28
Peak line density of pile up event [event/mm] (max over stable beams) ^{see Note 13}	0.21					
Levelling time [h] (assuming no emittance growth) ^{see Notes 8,13}	-		8.3	7.6	9.5	9.5
Number of collisions in IP2/IP8 ^{see Note 7}	2808		2452/2524	2288/2396	1163/1868	1163/1868
N_b at LHC injection ^{see Note 2}	1.20×10^{11}		2.30×10^{11}	2.30×10^{11}	2.40×10^{11}	2.40×10^{11}

Figure A.2: Overview of the LHC beam parameters - part 1 [11]

n_b / injection	288	288	224
N_{tot} / injection	3.46×10^{13}	6.62×10^{13}	5.40×10^{13}
ϵ_n at SPS extraction [μm] <i>see Note 3</i>	3.4	2	≤ 2.00 <i>see Note 6</i>
			1.7

Notes:

- ¹ Assuming one less batch from the PS for machine protection (pilot injection, TL steering with 12 nominal bunches) and non-colliding bunches for experiments (background studies). Note that due to RF beam loading the abort gap length must not exceed the 3 μs design value.
- ² An intensity loss of 5% distributed along the cycle is assumed from SPS extraction to collisions in the LHC.
- ³ A transverse emittance blow-up of 10 to 15% on the average HV emittance in addition to the 15% to 20% expected from intra-beam scattering (IBS) is assumed (to reach the 2.5 μm /3.0 μm of emittance in collision for 25ns/50ns operation).
- ⁴ As of 2012 ALICE collided main bunches against low intensity, satellite bunches (few per-mill of main bunch) produced during the generation of the 50ns beam in the injectors rather than two main bunches, hence the number of collisions is given as zero.
- ⁵ For the design of the HL-LHC systems (collimators, triplet magnets...), a design margin of 50% on the stated peak luminosity was agreed upon.
- ⁶ For the BCMS scheme emittance down to 1.4 μm have already been achieved at LHC injection which might be used to mitigate excessive emittance blowup in the LHC during injection and ramp.
- ⁷ The lower number of collisions in IR2/8 wrt. to the general purpose detectors is a result of the agreed filling scheme, aiming as much as possible at a democratic sharing of collisions between the experiments.
- ⁸ The total number of events/crossing is calculated with an inelastic cross-section of 85 mb (also for nominal), while 100 mb is still assumed for calculating the proton burn off and the resulting levelling time.
- ⁹ BCMS parameters are only considered for injection and as a backup parameter set in case one encounters larger than expected emittance growth in the HL-LHC during injection, ramp and squeeze.
- ¹⁰ The crossing angle for the 8b+4e alternative could be reduced down to about 400 μrad (9 σ) thanks to the lower number of long ranges.
- ¹¹ Minimum normalised long-range beam-beam separation at minimum β^* .
- ¹² The 8b+4e variant represents a back-up scenario for the baseline 25ns operation in case of e-cloud limitations. The parameters are still evolving but are stated for the sake of performance reach comparison.
- ¹³ The current baseline foresees a staged installation of crab-cavity modules in LS3 and LS4, having an initial impact parameters like β^* , crossing angle, virtual luminosity reach and levelling time.

Figure A.3: Overview of the LHC beam parameters - part 2 [11]