Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelor Thesis

Dennis Kirsch

Synchronized Infrastructure for RGB-D Sensor Networks

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Dennis Kirsch

Synchronized Infrastructure for RGB-D Sensor Networks

Bachelor Thesis eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr. Birgit Wendholt

Eingereicht am: 12. Januar 2018

Dennis Kirsch

Thema der Arbeit
Synchronized Infrastructure for RGB-D Sensor Networks

Stichworte
Synchronisation, Verteilte Systeme, Architekturdesign, Echtzeit Oberflächenrekonstruktion

Kurzzusammenfassung

Moderne Sensornetzwerke bestehen aus mehreren Sensoren, die innerhalb einer Deadline ihre gemessenen Werte an eine verarbeitende Instanz senden sollen. Die Sensoren selbst sind da meistens simpel und geben keine Echtzeit-Versprechungen. Zusätzlich ist eine Synchronisation innerhalb des Netzwerks nicht gegeben, welche es ermöglicht von allen Sensoren gleichzeitig Messwerte zu erhalten.

Aus diesem Grund entstehen Synchronisationsprobleme wie sie in Petersen's Arbeit beschrieben sind [1]. Für die Microsoft Kinect, die in Petersen's Versuchaufbau verwendet wird, wurde mit OmniKinect [2] bereits ein Lösungsansatz für das Problem vorgestellt. Allerdings wird durch die Skalierung über Universal Serial Bus (USB) Controller Chips die Bandbreite des south-bridge Controllers von dem Mainboard zum Flaschenhals.

Ziel dieser Bachelorarbeit ist es, das Synchronisationsproblem aus Petersen's Arbeit zu analysieren und eine Lösung vorzuschlagen[1]. Dafür wird ein Sensornetzwerk mit drei Microsoft Kinects aufgebaut. Hierbei wird eine Echtzeitplattform vor jede Kinect geschaltet, welche deren Daten über ein Ethernet Netzwerk an eine verarbeitende Instanz senden. Dieser Aufbau dient als Basis für die Messungen, mit denen die vorgeschlagene Infrastruktur bewertet wird.

[1]Die resultierende Implementierung dieser Infrastruktur ist in folgendem GitHub Repository zur Verfügung gestellt: https://github.com/Cherden/Bachelor2017

Dennis Kirsch

Title of the paper

Synchronized Infrastructure for RGB-D Sensor Networks

Keywords

synchronization, distributed systems, architecture design, real-time surface reconstruction

Abstract

Modern sensor networks consist of multiple sensors, which have to deliver their data within a given deadline to a processing instance. The sensors themselves ar mostly simple and cannot fulfill real-time requirements. A synchronization within the network, which allows to obtain data synchronously from the sensors, is not given as well.

This is the reason why Petersen observed the synchronization problems in his work [1]. In his test setup he uses the Microsoft Kinect. OmniKinect already proposed a solution for synchronization with this sensor [2]. But since their solution scales with Universal Serial Bus (USB) controller chips, the bandwidth of the mainboard's south-bridge controller acts as a bottleneck.

The goal of this bachelor thesis is to analyze the synchronization problems of Petersen's work and propose a solution for it[2]. For the analysis, a sensor network containing three Microsoft Kinects is used. Each Kinect is connected to a real-time plattform, which sends their data over an Ethernet network to a processing instance. The mesaurements to evaluate the proposed infrastructure are based on the described testbed.

[2]The resulting implementation of the proposed solution have been made available on the following GitHub repository: https://github.com/Cherden/Bachelor2017

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Requirements . . . 2

1.2.1 Problems . . . 2

1.2.2 Solutions . . . 3

1.3 Goals . . . 4

1.4 Organization . . . 4

2 Basics **5**

2.1 Middleware . . . 5

2.2 Remote Procedure Call . . . 7

2.3 Networked Control Systems . . . 10

2.4 Clock Synchronization . . . 11

2.4.1 Centralized Synchronization . . . 12

2.4.2 Decentralized Synchronization . . . 13

2.4.3 Discussion . . . 13

3 Related Work **15**

4 Architecture Design **17**

4.1 Testbed . . . 17

4.2 Outer Boundaries . . . 18

4.3 Components . . . 20

4.3.1 Node . . . 20

4.3.2 Aggregator . . . 27

4.3.3 Processing Speed . . . 30

4.4 Communication . . . 31

5 Network Design **34**

5.1 Election . . . 34

5.2 Synchronization . . . 37

5.2.1 Algorithm . . . 37

5.2.2 Test of the Algorithm . . . 39

5.2.3 Resolution . . . 40

5.3 Network Control Communication . . . 41

6 Conclusion **43**

List of Figures

1.1 Illustration of the current steps for reconstruction. . . . 2
1.2 Picture showing the synchronization problem in the reconstruction. Source: Iwer Petersen . . . 2
1.3 Test setup of the old architecture. Source: [1] . . . 3

2.1 RPC with (a) synchronous and (b) asynchronous communication. . . . 8
2.2 Message sequence of a distributed RPC with a forwarding and non forwarding DRPC server. . . . 9
2.3 Typical structure of an NCS. Source: [3] . . . 10
2.4 Measuring delay and offset in NTP. Source: [4] . . . 12
2.5 Sequence for time synchronization using the Berkeley algorithm. . . . 14

4.1 The test setup used for measurements in this thesis. . . . 17
4.2 Black Box model of the infrastructure showing the interfaces to the outer systems. 19
4.3 Component diagram showing the abstract inner components of the infrastructure. 21
4.4 The maximum, minimum and average time it takes to obtain one data packet using the synchronous driver. . . . 22
4.5 The maximum, minimum and average time it takes to serialize the sensor data to a byte stream using (a) the normal setter and (b) a setter with pre-allocated memory. . . . 24
4.6 The maximum, minimum and average time it takes to send one data packet from one node to the processing instance. . . . 25
4.7 The maximum, minimum and average time it takes to process each step. . . . 26
4.8 The maximum, minimum and average time it takes to parse one message from one of the three nodes. . . . 27
4.9 The maximum, minimum and average time it takes to copy a video and depth frame to the function caller. . . . 28
4.10 The maximum, minimum and average time it takes to calculate and return a point cloud. . . . 29
4.11 The maximum, minimum and average process time using three nodes with this architecture. . . . 30
4.12 The maximum, minimum and average time for one node to send its data with three nodes in the network. . . . 31
4.13 Communication model for the presented architecture. . . . 33

5.1 Message sequence of the Bully algorithm. . . . 35
5.2 Message sequence of the modified algorithm. . . . 36

5.3 Message sequence of the implemented synchronization algorithm, based on Berkeley's algorithm. 38
5.4 Message sequence of the implemented synchronization algorithm, based on Berkeley's algorithm. 39

6.1 Picture showing two frames of the recreation algorithm during a fast movement. 43

1 Introduction

After voice over IP and video conferences, the next step to distributed collaboration are virtual workplaces. Additionally to hearing and seeing a communication partner, a virtual room is offered to simulate a more natural interaction. Meetings could be held in such a room, removing travel cost and time. It is also essential for distributed work on a virtual object, e.g. modeling a car. This requires the scan and reconstruction of all communication partners in real time. Petersen developed and compared methods to reconstruct user as an avatar in a digital, three-dimensional space [1].

This thesis' goal is to propose a solution for the problem of synchronization addressed in the master thesis of Iwer Petersen [1]. This chapter gives an overview over the mentioned problem which provides the motivation for this work. Further, the requirements for a solution are discussed. From that the goals and organization of this thesis are presented.

1.1 Motivation

Petersen's reconstruction algorithm uses depth and RGB pictures from three RGB-D sensors to generate models [1]. Each frame therefore consists of two pictures that have to be processed. Further, these frames have to be combined. Since the model should be used for communication and collaboration, real-time processing is a key requirement.

But this process is time consuming, which is shown by the minimum processing time measured by Petersen [1]. With 272 ms per frame, the reconstruction algorithm only outputs three frames per second (fps). This also raises the problem of synchronization. It is necessary to know when the pictures were acquired during the 272 ms time period. In Figure 1.1 the problem is illustrated.

Assuming the last step of combining the frames takes 100 ms, there is a 172 ms time window in which the cameras can capture a picture. Having too much of a delay between each picture results in deformed models and blurry textures. This is shown in a picture taken by Petersen, visible in Figure 1.2. A fast movement results in a model, which is not recognizable anymore.

Total processing time
Process Frame 1
Process Frame 2
Process Frame 3
Combine Frames

Figure 1.1: Illustration of the current steps for reconstruction.

A new architecture can tackle these problems, which's requirements get analyzed in Chapter .

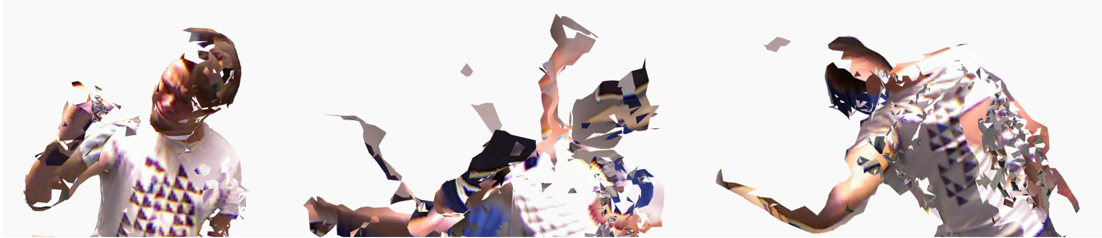

Figure 1.2: Picture showing the synchronization problem in the reconstruction. Source: Iwer Petersen

1.2 Requirements

1.2.1 Problems

Before requirements for a new architecture can be established, the weaknesses of the old one has to be analyzed. The setup is shown in Figure 1.3. There the RGB-D sensors are connected directly to the processing instance, which runs the algorithm. Since the used driver only offers a stream, the frames arrive solely on the criteria of the sensor being able to deliver data [1]. This results in the time capture offset. Because of the one way communication, no synchronization between the cameras can be implemented.

The other problem is the communication medium. All three sensors are connected via the same Universal Serial Bus (USB) to the computer. The RGB-D camera used in his setup is the Microsoft Kinect, which uses the USB 2.0 standard for communication. This standard offers a maximum bandwidth of 280 Mbit/s [5]. Calculating the size of one data package containing a

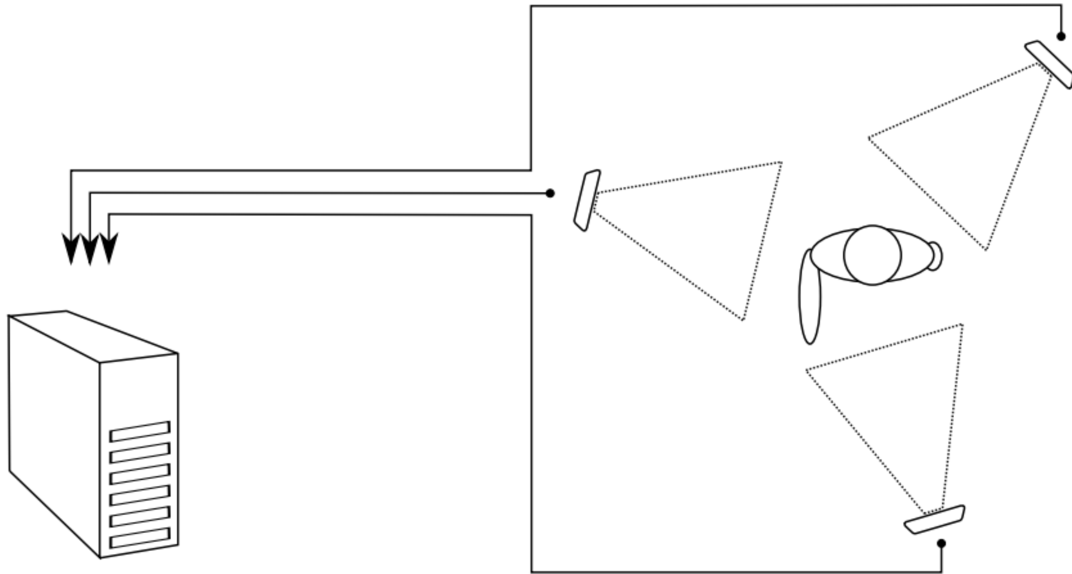

Figure 1.3: Test setup of the old architecture. Source: [1]

video and depth frame results in about 5.8 Mbit per package [6]. With 30 frames per seconds it adds up to a required bandwidth of 175.2 Mbit/s per sensor. Splitting the communication medium for three devices will therefore cut the frame rate immensely. Additionally to the raw data of the frame comes the offset for the RGB format. This transformation is integrated in the driver and therefore increases the required bandwidth. One converted RGB and depth frame is about 12 Mbit big, needing 360 Mbit/s bandwidth to run at 30 FPS.

1.2.2 Solutions

Based on these issues, requirements can be derived in order to build a new infrastructure that addresses them. This includes both, requirements for the software and network. Table 1.1 lists these requirements.

Infrastructure Requirements		
#	**Description**	**Comment**
R1.1	The network should support the required 1.1 Gbit/s.	Since the application cannot process 30 frames per second, a fairly lower bandwidth will suffice.
R1.2	Each node has to be able to obtain 30 frames a second per camera.	-
R1.3	The time difference between each node must not exceed 1 ms.	The time granularity is sufficient for this context.
R1.4	The delay between the capture of each frame has to be below 1 ms.	-
R1.5	The time to make one frame obtainable has to be below 33 ms.	This is necessary in order to achieve a maximum delay of one frame.
R1.6	During the data transmission from nodes to processing instance, no other communication should happen over the network.	Since the high amount of data which will be sent over the network, it should not have to share the bandwidth with other applications.
R1.7	All three frames have to be accessible as one data package through an interface.	This gives better control and also guarantees that the frames are synchronized.
R1.8	The data transmission has to occur on demand.	Receiving a consistent stream of data will use computing power from the processing instance, which could be used for the application.

Table 1.1: Requirements for the new infrastructure, which supports three RGB-D cameras.

1.3 Goals

The goal of this work is to present an infrastructure, that fulfills the requirements presented in Table 1.1. It consists of the software architecture and a concept for the network. The architecture design of the outcome is discussed and its time correctness proven with measurements.

1.4 Organization

Chapter 2 gives an overview over the topics needed for this work. In the next Chapter 3 the outcome of related work is presented. The following Chapter 4 discusses the software architecture design choices. Building on that, Chapter 5 offers solutions to the networking problems. Lastly, Chapter 6 summarizes the results and gives an outlook for future work.

2 Basics

This chapter provide the basics, that are needed for this work. The Section 2.1 explains the idea of middleware architectures in distributed systems. Based on that, remote procedure calls (RPC) are added to the concept in Section 2.2. Networked control systems (NCS) are elucidated in Section 2.3 and the needed clock synchronization for it in Section 2.4.

2.1 Middleware

An application does often not only run local on one computer, but distributed. It has to communicate with other machines to access their services in order to offer its full functionality. A service can be to process given data, store files, or acquire data. In this example the communication participants can be divided in two groups. There is a client, who wants to use the service. Then there are one or multiple servers who provide said services.

The application in this case is a *distributed application*. It is a composition of components which offer a function. The components themselves can not offer the functionality on their own and rely on each other. Besides not having to implement available services, distributing tasks also divides the work load. This can result in faster processing of the main task. [7]

The communication between these components can take place on one machine, in a local network or the Internet. In either case, the application often does not need to know where these components are available. So, communication can be abstracted by offering an interface between the component and application. The software that offers the interfaces to both components is called *middleware*. The middleware's only task is to transfer data between the two. On a local machine, the data transfer over shared memory can be encapsulated, just like communication between multiple machines over sockets. This generates an abstraction, which makes it invisible for the application if a function is executed locally or remote. *Location transparency* is one of the transparencies a middleware can offer. Others are access, and failure transparency. [7]

When implementing a middlware, it is not given that both components who want to communicate use the same processor architecture. One may use a little and the other big endian to represent data in memory. If the middleware would simply copy the data and not regard the endianess it could result in misinterpretation of the data. This would lead to non-deterministic behavior in the application.

To guarantee that the data is interpreted the same way on all machines, it has to be converted to one universal byte order during transfer. After the transmission it has to be converted back to be readable by machine specific architecture. This converting process is called *marshalling* or *serialized. Unmarshalling* or *deserialization* is the process to convert it back.

This process is implemented in different tools. The most known are XML [8], JSON [9], and Google's Protocol Buffers [10]. Benchmarks for the serialization and deserialization time have been made before in the past [11, 12]. Even though the performances may be influenced by the specific hardware specs the test has been run on, Google's protobuf seems to perform the best. Therefore this tool will be used to serialize data in this work.

Protobuf's description language is based on the C++ syntax. In the so called *proto* files, messages get defined. The messages contain fields which can be of different data types, like boolean, integer or character fields. Each field have an identifier assigned, which is needed for the serialization and parsing process. Depending on the syntax used, it has to be specified if a field is required or optional. If a required field is not set, the serialization would return an error. The `proto2` syntax needs this specification, while the newer `proto3` does not. `Proto3` always returns the default value of a data type if a field is not set. An example message in the `proto3` syntax, which is used in this work, is shown in Listing 2.1. [13]

```
1 syntax = "proto3";
2
3 message ExampleMessage {
4         bytes data = 1;
5         uint64 timestamp = 2;
6 }
```

Listing 2.1: Example protobuf message.

Middlewares can be divided in different types. One of them is the message-oriented middleware (MOM). In this concept the data is transfered as messages, to which the communication partner replies.

Other types have a more abstract approach. Some implement the request-response model, which offers access transparency. The client requests data and the server responds. The response can either happen synchronously - after request of the client - or asynchronously, when it is available. [7]

The request-response concept can be found in a RPC middleware, which the next section 2.2 will explain further.

2.2 Remote Procedure Call

As mentioned in Chapter 2.1, a remote procedure call (RPC) middleware relies on the request-response concept. It abstracts the message transportation between two components as a function call. For it, the middleware offers the same function signature to the client, as is implemented by the server. The stub only serializes the parameter list and function name, then sends it to the server. There the middleware calls the function with the received parameters. The return value is then sent back to the client, where the middleware forwards it to the calling function.

There are two modes in which this sequence can be executed. Either the call is synchronous or asynchronous. In the first case the client waits for a return value and the process blocks until the server sends it. A communication sequence of this method is shown in Figure 2.1a. The RPC can also be processed asynchronously. There the application calls the middleware stub method, which does not block. The return value from the server then gets stored in a queue upon receiving. From there the application can access the data when it wants to process it. Figure 2.1b shows a communication sequence of the asynchronous RPC execution.

Building on remote procedure calls are distributed RPCs (DRPC). The idea of it is described in Apache Storm. With this method, the function call from the client gets distributed in a network of nodes. In Figure 2.2 a sequence diagram shows the communication flow with the DRPC pattern. Having the task distributed in a network can increase the speed of the RPC. The idea is, that a network or topology has a DRPC server, which is the only one communicating with the one-to-n clients. It receives the procedure call and distributes it inside the topology. Each node then sends the data back to the DRPC server, which forwards it to the client. One approach to increase performance can be to send the data directly to the client, because for high amounts of data the DRCP server can act as a bottleneck. Then the client has to combine

the sub-results before processing the end result. [14]

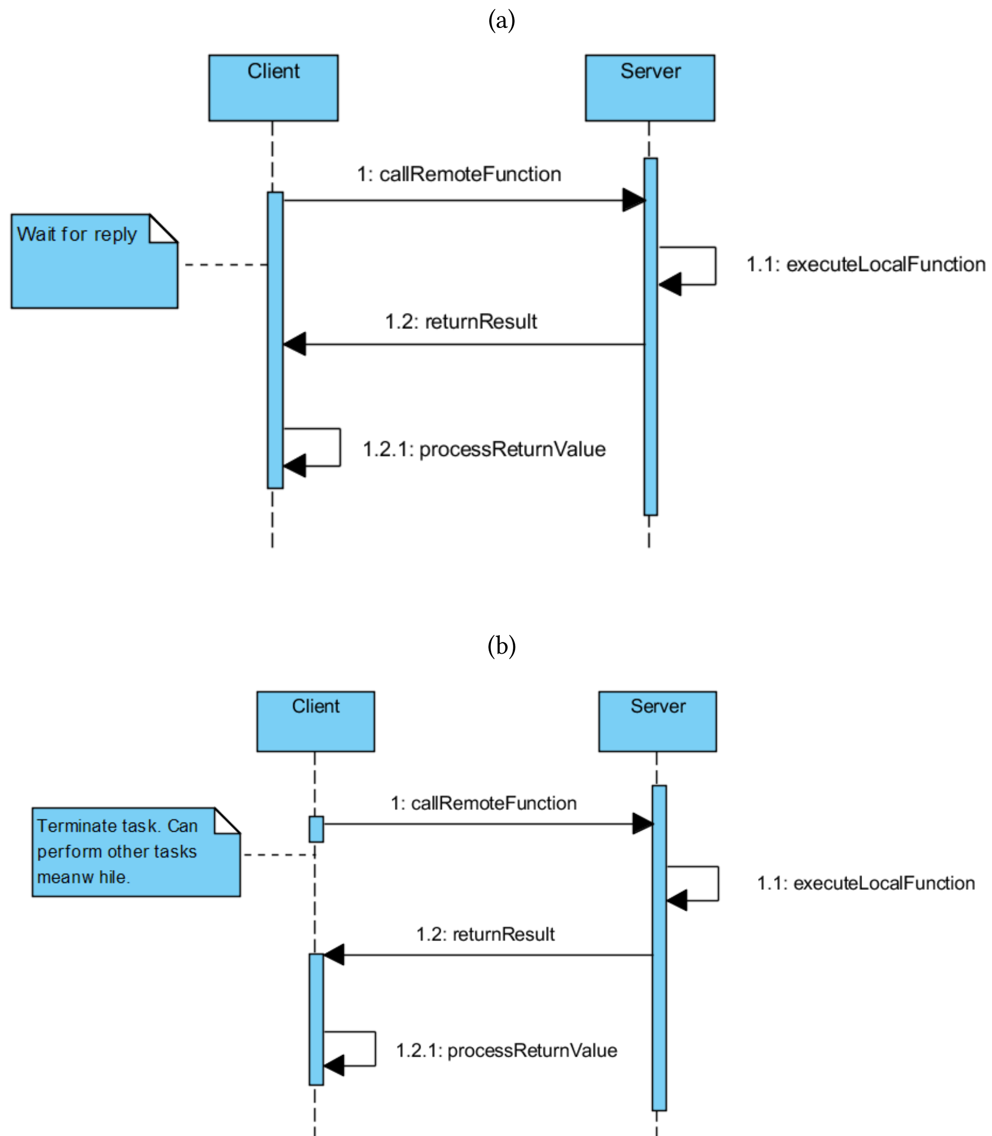

Figure 2.1: RPC with (a) synchronous and (b) asynchronous communication.

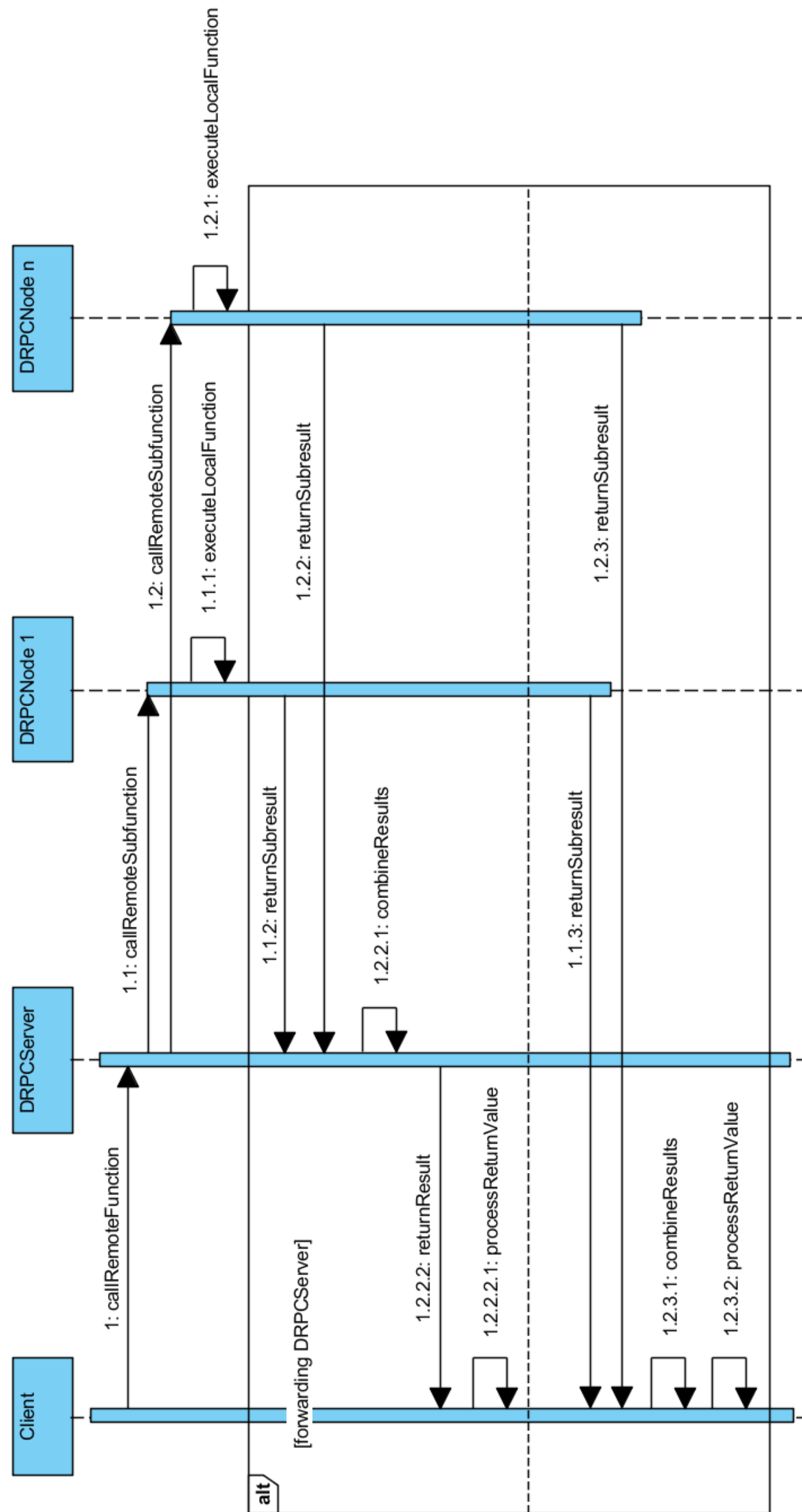

Figure 2.2: Message sequence of a distributed RPC with a forwarding and non forwarding DRPC server.

2.3 Networked Control Systems

Control systems are designed to react to external data, process it, and perform required actions if needed. For example a sensor measures the gas level in the car and sends the information to a controller. The controller processes the data and decides if an action has to be executed. If the gas is low, the controller reacts by turning on an led on the dashboard.

In a networked control system (NCS), the sensors, actuators, and controller are not directly connected. Figure 2.3 shows a model of this approach. They communicate over a network, which can be wired or wireless. This allows physical and quantifiable larger networks. As an exchange the network itself has to be controlled, too. Depending on the control system, specific requirements have to be met. For dynamic NCSs a routing control has to be implemented. Hierarchical NCS designs require protocols which allow the routing. Most control systems also need to fulfill real-time requirements. These specify a deadline, in which the data has to be arrived at the controller. [3]

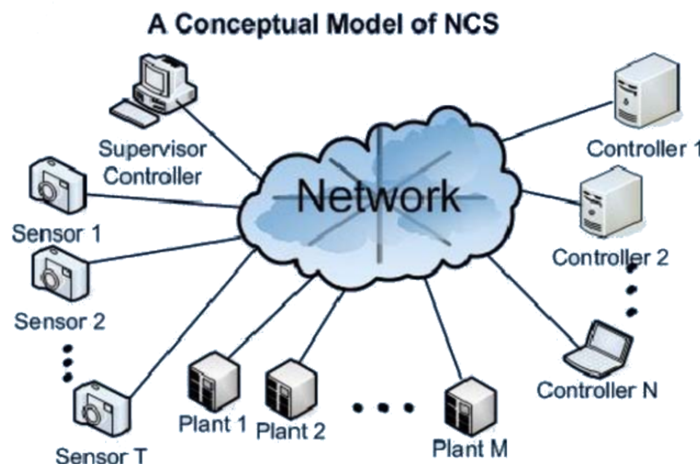

Figure 2.3: Typical structure of an NCS. Source: [3]

Deadlines can be divided in soft and hard deadlines. The former requires data to be there on time, but would not result in failure if it is not met. An example is video streaming, where if the data does not arrive on time it has to load. Hard deadlines on the other hand have to be met. Otherwise the system may harm humans or nature. A hard real-time requirement can be found in the control loop of a nuclear reactor. If the core overheats and the controller does not receive the information, it can result in a meltdown.

To develop an NCS which satisfies real-time requirements, every node within the network has to know the exact time. Otherwise two nodes can not agree on a deadline. Chapter 2.4 explains this problem further.

2.4 Clock Synchronization

Every computer has a quartz, which produces signals, or ticks, in a specific frequency. Out of these signals the computer can deduce when a second passes, if it knows the exact frequency. Kopetz [15] defined this clock as the *local time* with quartz accuracy. In this context it means that the clock is only locally correct. It does not indicate the Unix time [16], but only an approximation of seconds passed since the first measured tick.

It is not exact, because every quartz ticks in a different frequency. Factors for its frequency are the temperature and the accuracy of the quartz itself. Therefore the software that processes the ticks would need to know the exact frequency of the specific quartz in order to portray the time accurately. [17]

But the local time does not meet the requirements for distributed real-time systems. To satisfy them, the nodes within a network of such a system have to synchronize their local time. Contrary to the local clock, the *global time* represents an approximate clock within a network. To create the global time, synchronization algorithms are needed.

Kopetz laid down four rules which these algorithms have to satisfy for distributed real-time systems. The first rule says that the time between the local time of each node must not exceed a known constant. Secondly, the global time should be able to measure small time intervals. Additionally the algorithm has to be fault tolerant and, lastly it must not reduce the performance of the system. [15]

Further, the construct time is in a total order. Therefore, if an event A has a higher timestamp than event B, A occurred later. Synchronization algorithms may have to adjust the time backwards. If it would simply be overwritten, event C, which occurred after B could have a lower timestamp.

To allow the backwards adjustment of time while obeying the statements for a total order, another mechanism called *slewing* can be used. To slew the local time, the amount of ticks from the quartz for one time unit, i.e. virtual frequency, is increased. When the time is fully adjusted, the virtual frequency gets reset to its default value.

Synchronization algorithms can be divided in two categories, centralized and decentralized. Each group will be presented with an example in Chapters 2.4.1 and 2.4.2.

2.4.1 Centralized Synchronization

Centralized synchronization relies on a time server, which the nodes use to synchronize their time with. An example for such an algorithm can be found in the Network Time Protocol (NTP) [4]. Microsoft's Windows Time service, for example, uses this protocol for synchronization [18].

In Figure 2.4, the client A wants to synchronize his time with the server B. T_{i-3} represents the timestamp, at which the client sent the request packet to the server. At T_{i-2} the packet arrived at the server and T_{i-1} marks the timestamp where the response packet was sent. The timestamps T_{i-3} to T_{i-1} are included in the response from the server. With the arrival time T_i and equation 2.1 the client can calculate the true offset θ. [19]

$$\theta = \frac{(T_{\text{i-2}} - T_{\text{i-3}}) + (T_{\text{i-1}} - T_{\text{i}})}{2} \tag{2.1}$$

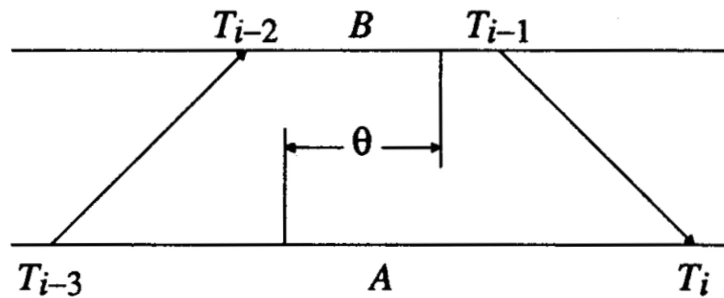

Figure 2.4: Measuring delay and offset in NTP. Source: [4]

2.4.2 Decentralized Synchronization

Decentralized synchronization algorithms tackle that problem. With this approach the nodes inside a network synchronize each other. The Berkeley algorithm [20] is one example of a decentralized synchronization method. Its functionality is similar to a centralized approach. At the beginning, a master node is chosen, which periodically polls the time of each slave node. After receiving an answer, the master node can estimate the round-trip time. Out of the timestamps from all slaves, the average timestamp t_{avg} can be calculated using equation 2.2, with n being the number of slave nodes and t_i being the timestamp of a slave.

$$t_{avg} = \sum_{i=1}^{n} t_i \tag{2.2}$$

The master then replies to each slave node an adjustment time t_adj, calculated by equation 2.3 with t_rtt being the round-trip time for a specific node.

$$t_adj_i = (t_i - t_{avg}) + \frac{t_rtt_i}{2} \tag{2.3}$$

The master then replies t_adj to each slave node. This reduces further inaccuracy through the round-trip time. All nodes adjust their clock by the given amount. An illustration of this algorithm is shown in Figure 2.5. [20]

2.4.3 Discussion

The centralized approach that NTP offers is an already established solution for the synchronization problem. But its biggest disadvantage is the client-server architecture. It offers a single point of failure in the network, which violates Kopetz third rule of fault tolerance [15].

The decentralized approach does not have that problem. If the master node fails, a new master can be elected. But since the algorithm synchronizes the time of all nodes at the same time, it does not scale well in big networks. There it would require a big portion of the bandwidth to send and receive messages at the same time. Therefore it would not fulfill Kopetz fourth rule, which says that the synchronization should not reduce the performance of the system [15]. In small networks, though, the algorithm satisfies all rules.

This infrastructure requires the algorithm to obey all rules laid down by Kopetz. If the synchronization algorithm would fail, the advantage of using this approach is lost. Therefore this infrastructure uses a decentralized over a centralized algorithm.

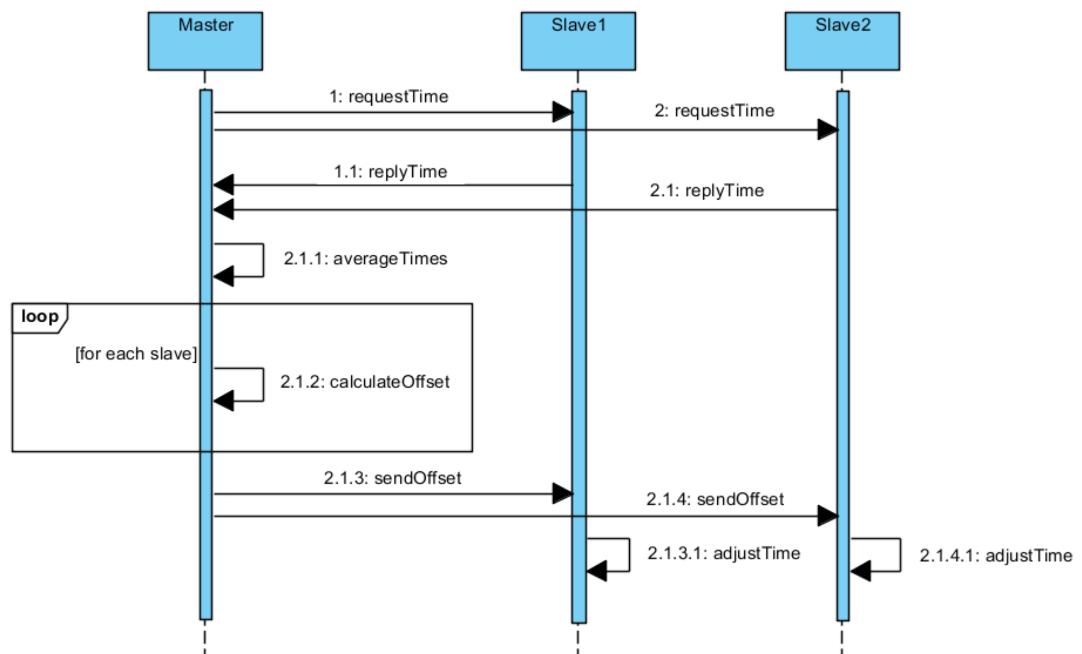

Figure 2.5: Sequence for time synchronization using the Berkeley algorithm.

3 Related Work

Other approaches to reconstruct a user in a digital space have yielded in well-performing results. Omnikinect [2] has presented a solution which scales over USB controllers. Their architecture feeds the processing instance with a stream of data from each camera. That way their system can operate with data from 7 cameras. The bottleneck lies in the bandwidth from the south-bridge of the motherboard.

A synchronization between the frames used for recreation has to be done by the processing instance. Therefore it runs in the same problem as Petersen's approach. Analyzing their videos shows that they did not solve the synchronization problem itself. Instead they implemented an algorithm, that corrects the offsets. Moving the synchronization inside a sensor network can make this algorithm obsolete and therefore improve the performance. As a result the reconstruction can be faster and produced at a higher frame rate.

Another approach has been made by Microsoft with Holoportation [21]. Instead of having one processing instance, they use multiple PCs to distribute the work load. Each PC supports two RGB-D cameras. Their implementation consists of 4 PCs, which results in a total of 8 RGB-D cameras. After those PCs process the data, they send it to another PC over a 10Gbps connection. There the data is then compiled and used to reconstruct the avatar. Instead of reconstructing a new avatar with each frame, they adjust the recreation.

Since only two cameras are connected to one PC, it can be controlled to obtain a synchronized image. In their paper [21] they do not write about a synchronization between the PCs but since their approach adjusts an already existing model, a non-synchronized frame can result in a deformation. The deformation can only be corrected by creating a new model. Derived from that, a synchronization has to happen at some point. But since the information about it is not available, this work tries to recreate their result. Since less expensive hardware is used, achieving similar results can be an improvement to their approach.

Regarding the requirements, most network related ones are located in the context of NCS 1.2.2. More specifically, the synchronization in those systems is of importance. In [22] the method of all-node-based clock synchronization is discussed. Compared to cluster-based or diffusion-based methods it has a scalability problem, because all nodes in the network have to participate in the synchronization process. The prerequisite, that the packet transmission time between all hops is equal is also fulfilled, since all sensors are within one hop accessible.

An algorithm allowing all-node-based synchronization is the Berkeley algorithm [20]. Its functionality is presented in Chapter 2.4.1. Other approaches [23, 24, 25, 26] tackle problems like scalability and finer granularity, which are present in Berkeley's.

With the Precision Time Protocol (PTP) the IEEE defined a standardized clock synchronization protocol in local area networks. It selects a master node, or grandmaster, using the best master clock (BMC) algorithm. The nodes within the network then synchronize their time with the grandmaster. In general, the synchronization mechanism is similar to Berkeleys. But it offers higher configurability and allows synchronization over multiple domains. It also synchronizes the global clock within nanosecond accuracy.

4 Architecture Design

This chapter discusses the architecture design for this infrastructure. It is presented in a top-down approach. Therefore the physical components are identified first and then refined in subcomponents. A critical part in the design choices are the requirements listed on page 4 in Table 1.1.

4.1 Testbed

The physical components for this infrastructure are shown in Figure 4.1. With this setup, i.e. testbed, the time critical requirements get validated through measurements. The components are divided in four groups, which get explained in the following sections.

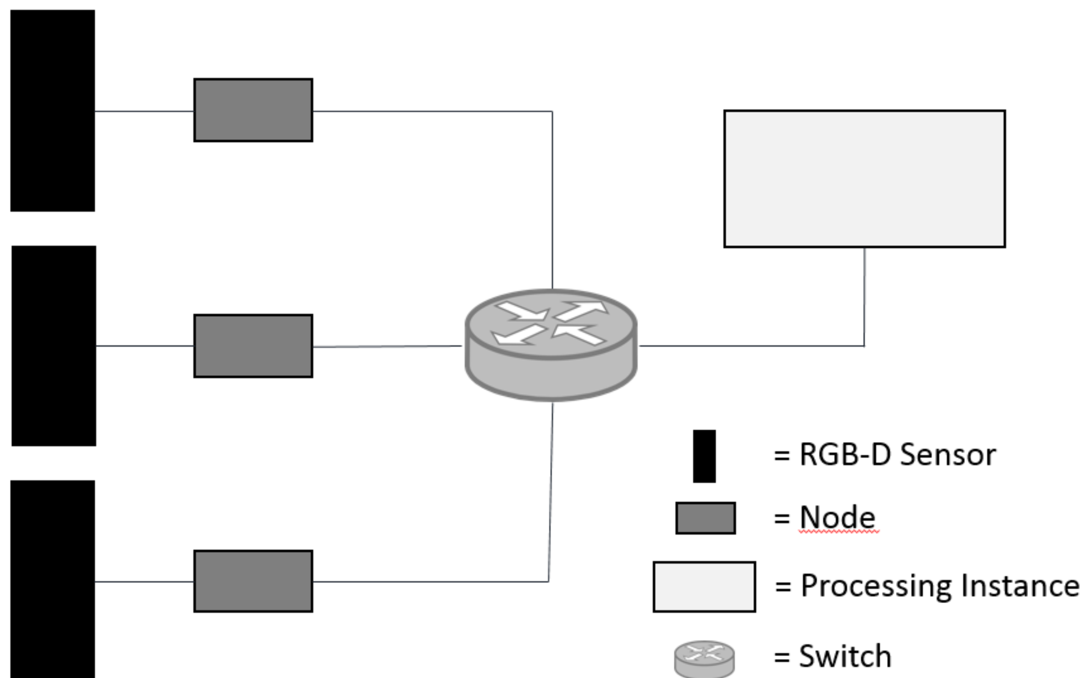

Figure 4.1: The test setup used for measurements in this thesis.

Sensors

As mentioned in the motivation (Chapter 1.1), the sensors used are RGB-D cameras, specifically the Microsoft Kinect v1. The cameras capture 30 RGB and depth frames per second with a 640x480 pixel resolution [6].

Nodes

The nodes have to be able to obtain 30 frames per second to comply with requirement R1.2. They will need to process the data with enough speed ensuring that they do not violate requirement R1.5. In order to comply, the Odroid Xu4 single board computer was used and Ubuntu 16.04 was installed as the operating system.

Switch

To comply with requirement R1.1 an HP 1810 switch was used as it provides a bandwidth of 1 Gbit/s. It offers easy configuration through a web interface and supports jumbo packets of up to 9216 byte frame size. Besides the support of RJ45 copper cables, two slots for fiber connectivity are offered. [27]

Processing Instance

The processing instance is an Intel Core i7 3.4 Ghz CPU, 16 GB of RAM and an Intel Gigabit Ethernet Controller. Ubuntu 17.04 is used as the operating system.

4.2 Outer Boundaries

The goal of this section is to identify the outer boundaries of the system and analyze their interfaces. Out of the physical components, the black box in Figure 4.2 can be derived. It shows the interfaces for input (ISensor) and output (IApp) of the infrastructure. The white components are external and only the infrastructure is in the focus of this work.

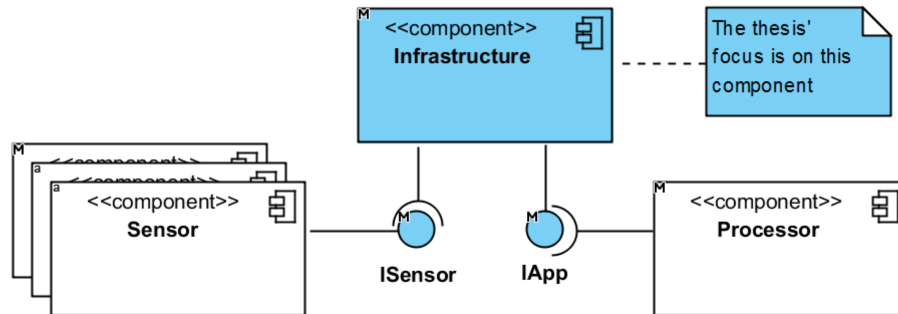

Figure 4.2: Black Box model of the infrastructure showing the interfaces to the outer systems.

ISensor is provided by the driver of the sensor. Microsoft's Kinect for Windows SDK cannot be used in this work, because the nodes run a Linux operating system. Instead an open source driver called libfreenect[1] is used. It also provides the functionality to obtain data synchronously from the sensor. This feature allows the node to obtain a current image when triggered.

The interface to the Processor, IApp, has to be designed regarding the requirements R1.7 and R1.8. Listing 4.1 proposes an approach that fulfills the requirements.

```
1 class IApp{
2 public:
3         int obtainNewData();
4
5         int getVideo(int sensor_id, char* buf, int size);
6
7         int getDepth(int sensor_id, char* buf, int size);
8 };
```

Listing 4.1: Interface to the application.

The function `obtainNewData()` triggers the nodes to acquire new data from the sensors. The trigger based transmission required by R1.8 is fulfilled with this approach. It also gives status about the sensor network with its return value. Possible messages are:

[1]The driver is developed by the project OpenKinect presents at https://openkinect.org/wiki/Main_Page

Not all nodes connected The Processor needs all three nodes to be connected and able to deliver data in order to function. If they are not, the Processor has to wait.

Currently acquiring data When all nodes are connected, this function signals the nodes to acquire data. This process can only be started once at a time. If the nodes are currently obtaining the frames and sending them to the processing instance, it will be announced to the user.

Acquisition succesful On return of this message, the user is notified that the data acquisition was successful and can be obtained.

The data is stored until the function is called again, which complies with requirement R1.7. Functions `getVideo()` and `getDepth()` can be called to get the video and depth frame, respectively. As parameters both expect the `sensor_id`, a buffer (`buf`) to write the data to and the `size` of the buffer. Their return values give information whether the operation is currently executable or not. The main reason for their failure is that no data is available. Either because one of the reasons mentioned in `getNewData()` or the available frame has already been processed.

4.3 Components

Within the black-box component named "Infrastructure" in Figure 4.2, the inner components can be identified. Figure 4.3 shows this diagram. Visible are the two components: Node and Aggregator. They communicate through the interface IAggregator. The following sections provide information about each component.

4.3.1 Node

Each node is connected to one sensor component, communicating through the interface ISensor. Its task is to obtain one frame, prepare it for transfer and then send it to the aggregator. To comply with requirement R1.5, this process can not exceed 33 ms. Optimizing each step reduces the delay between the capture and the portrayal of the reconstructed avatar. In order to validate requirement R1.5 was met, the maximum process time of each task is measured.

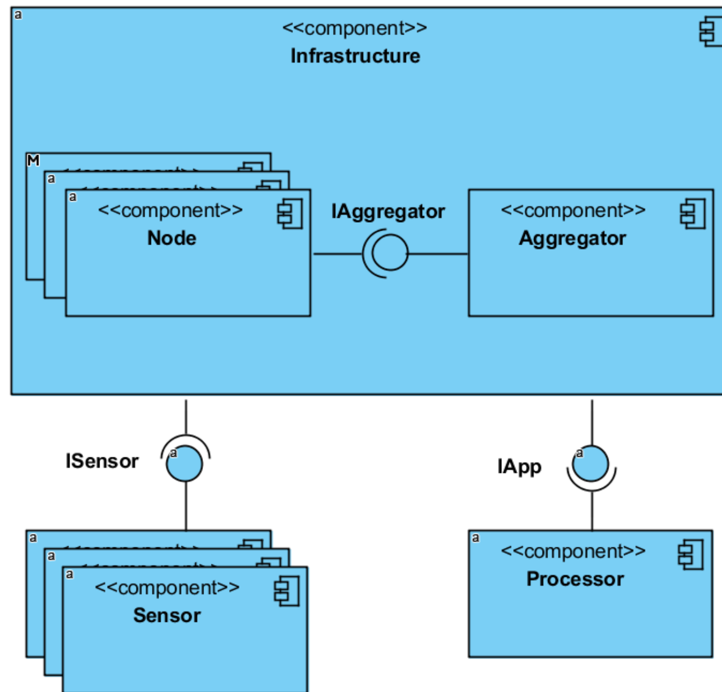

Figure 4.3: Component diagram showing the abstract inner components of the infrastructure.

To illustrate that this architectural setup would meet the time requirements disregarding the network, the measurements were done with only one node active. The results of this architecture utilizing 3 nodes is presented in Chapter 4.3.3. All measurements were captured in a time interval of three minutes, which amounts to a total amount of 5384 measurements. They show the maximum, minimum and average time it took to process each step. From importance is the maximum time to calculate the guaranteed deadline of this infrastructure.

Obtain Frame

This step only consists of the driver calls. The delay of capturing a frame packet synchronously is measured. The functions `freenect_sync_get_video_with_res()` and `freenect_sync_get_depth_with_res()` requires a pointer to allocate memory and save the data on. Reusing the same pointer for each function call saves the step to allocate new memory.

When calling the function with a higher frequency than 30 frames per seconds would allow, it waits until new data can be obtained. To ensure that this delay does not affect the measurements being collected, the processing thread was sent to sleep.

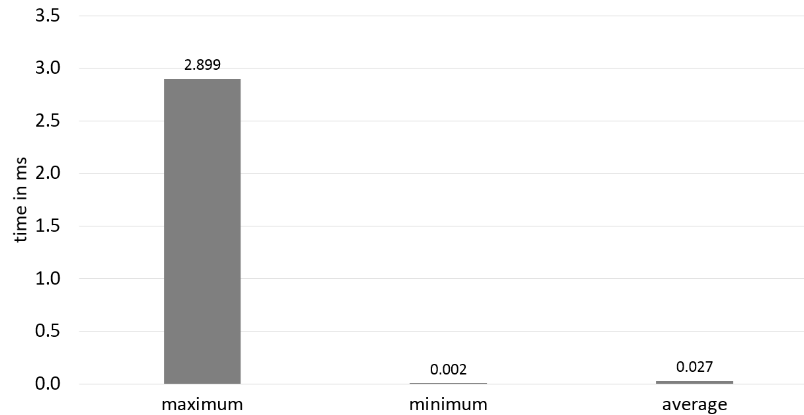

Figure 4.4: The maximum, minimum and average time it takes to obtain one data packet using the synchronous driver.

The measurements shown in Figure 4.4 show, that the processing time for this task can almost be neglected. With an average of 0.027 ms it takes about 0.08% of the 33 ms time window. The maximum of 2.899 ms most likely occurred, because the sleeping thread was awaken too early. It therefore gets treated as an statistical outlier.

Serialization of Data

During this step, the data has to be serialized in order to be transfered over the network. The necessity of this is explained in Chapter 2.1. As mentioned there, the serialization tool used is Google's protobuf [10]. The message used to serialize the data is shown in Listing 4.2. `fvideo_data` and `fdepth_data` contain the video and depth frame. The timestamp is represented in microsecond resolution and taken right after the previous step is completed. It is obtained through the function `gettimeofday()` of the `<sys/time.h>` Linux system library.

```
syntax = "proto3";

message KinectFrameMessage {
        bytes fvideo_data = 1;
        bytes fdepth_data = 2;
        uint64 timestamp = 3;
}
```

Listing 4.2: Protobuf message for node serialization.

As already mentioned in Chapter 1.2.1, this results in an approximately 12 Mbit big data packet. The time it takes to serialize this amount of data is illustrated in Figure 4.5a. This process takes with a maximum of 5.6 ms about 17% of the 33 ms timespan that is available (see R1.5 of Table 1.1 on page 4).

Analyzing the used setter for the protobuf messages shows that new memory is allocated every time the field is set. The API also allows to pass pre-allocated memory, which eliminates this offset. The times measured using the optimized process is shown in Figure 4.5b. The maximum of 1.7 ms is measured during the first serialization process, which includes the initialization of the message. But with an average of 0.9 ms it is more than 3.5 times faster than the method used before.

Send to Aggregator

With the last step, the serialized data is sent to the aggregator over sockets. For this process, there exist two protocols, the Transmission Control Protocol (TCP)[28] or the User Datagram Protocol (UDP)[29]. Video streaming applications normally use UDP as the base protocol. The main reason lies within the use cases of these applications. Packets that get lost on the wire do not have to be retransmitted, since a newer packet will be sent shortly after. The received packets also do not have to be ordered. Only packets with a later send time than the last received packet can be used, the other are discarded.

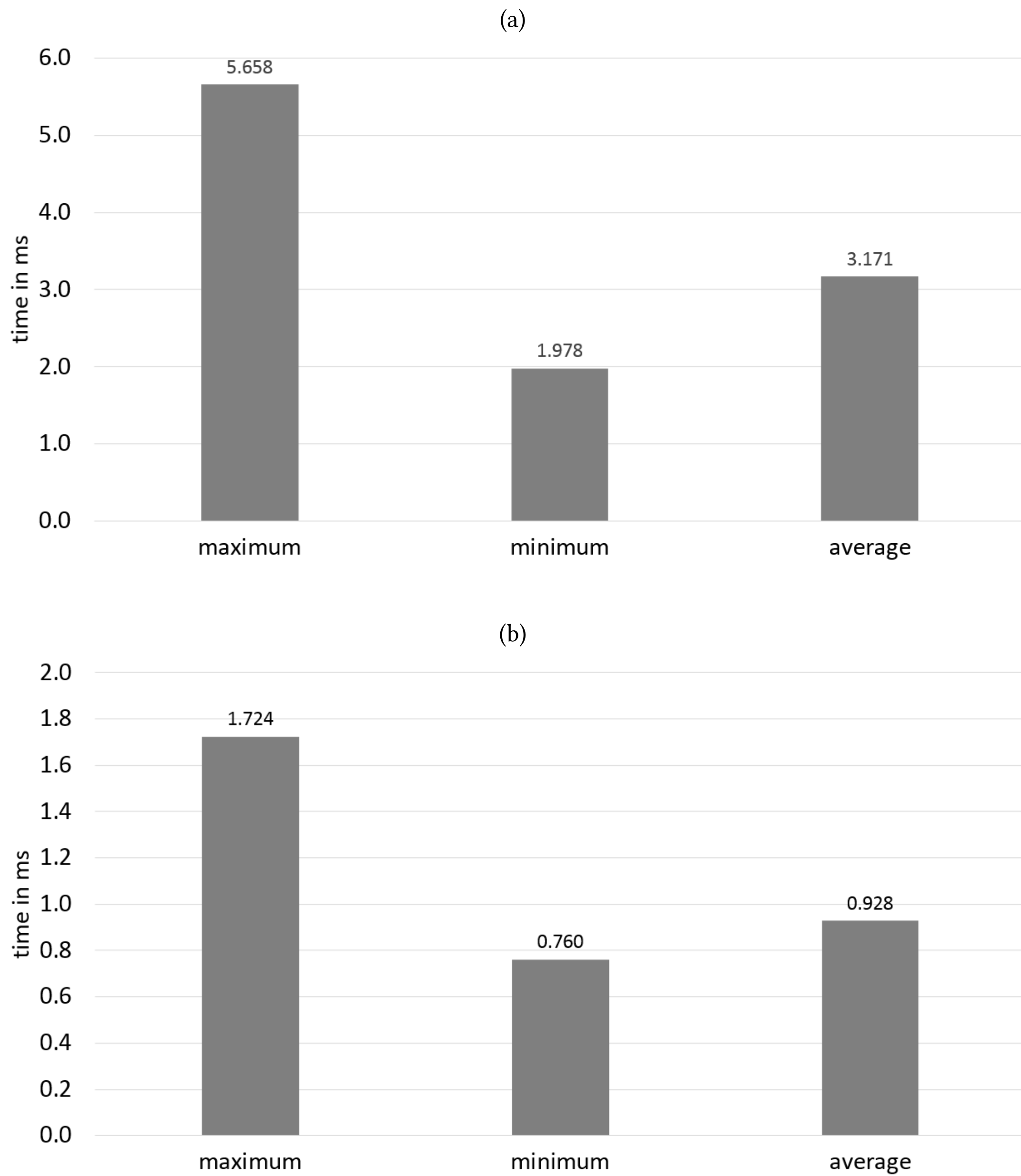

Figure 4.5: The maximum, minimum and average time it takes to serialize the sensor data to a byte stream using (a) the normal setter and (b) a setter with pre-allocated memory.

Chapter 4.1 mentions that the network does not meet what is required by the application. This results in a higher chance of errors through data collision and corruption, a reason which makes UDP unsuitable for this kind of activity. Lastly, the main reason why UDP is not suitable in this context is that one data package is about 12 Mbit big (see Chapter 1.2.1). Since a UDP packet can have a maximum length of about 65 KB [2], the data will be split in about 230 packets.

With these packets it is important, that all of the arrive in order. Therefore TCP is used in this application. The amount of packets is the same, but the protocol provides the mechanism for an ordered transmission of all data. Measuring the time of this process shows that this takes the longest time. As illustrated in Figure 4.6, it takes a maximum of 13.7 ms to complete the process. The big difference between the maximum and minimum process time can be generated by the processing instance. During the runtime of other threads, no packets in the receive buffer can be processed. This step therefore is dependent on the activeness of the processing instance.

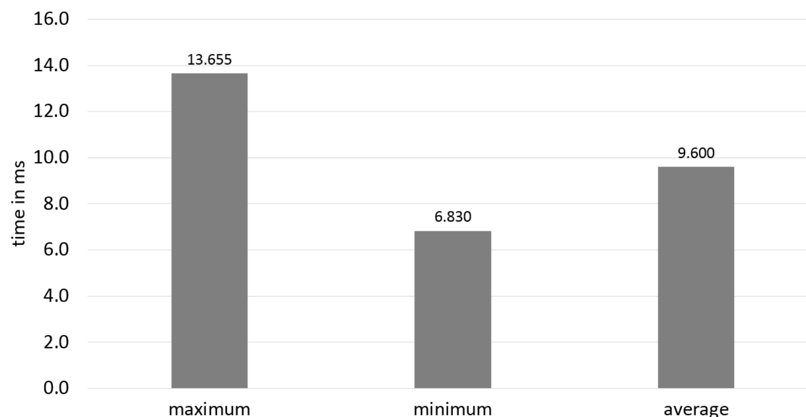

Figure 4.6: The maximum, minimum and average time it takes to send one data packet from one node to the processing instance.

An optimization of this step can be achieved by increasing the maximum transmission unit (MTU). This decreases the amount of overhead through the Internet Protocol (IP)[30] and TCP headers in exchange for higher chance of data corruption. The results in Figure 4.6 where measured with the MTU configured at 9000 bytes instead of the default 1518 bytes of the Ethernet standard[3].

[2]The UDP header's length field is 16 bit long and therefore allows a maximum length of 65535 bytes. [29]

[3]The minimum data length of an Ethernet frame is 1500 bytes [31], the MTU also regards the 14 bytes header and 4 bytes checksum of an Ethernet frame.

Total Process Time

Adding the maximum time of each step results in a maximum process time of about 18.3 ms. As a result the requirement R1.5 of Table 1.1 on page 4 is met.

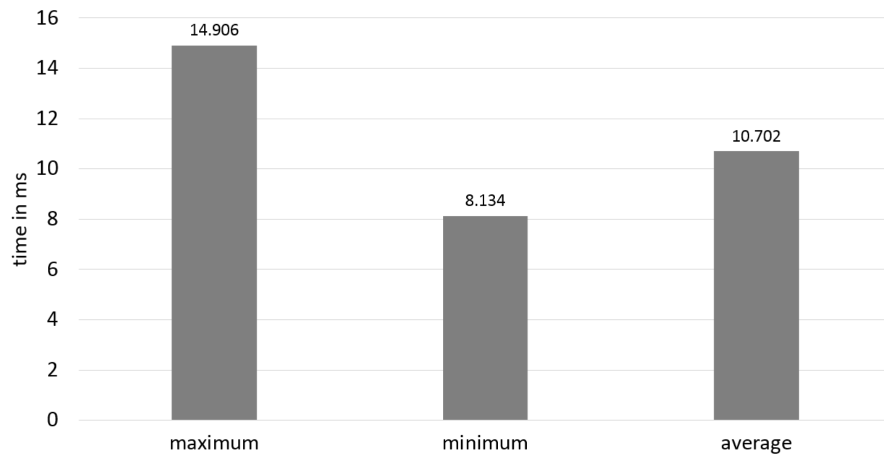

Figure 4.7: The maximum, minimum and average time it takes to process each step.

But as mentioned in each section, some maximum times where measured during the first acquisition process or are statistical outlier. So in order to determine the relative maximum process time, all steps have to be measured back to back. Figure 4.7 shows the results of this measurement. With 14.9 ms the relative maximum process time amounts to less than 50% of the required 33 ms (see R1.5 in Table 1.1 on page 4).

Since each step is optimized in its runtime, the only way to increase the performance is by reducing the data to serialize and send. This can be achieved by compressing the data. But the amount of time this approach wins has to be compared with the processing time of the compression. Since the requirement is fulfilled, which is shown in the next chapter, this step was disregarded. Should the speed of the application accessing the interface increase, this would be the next point of optimization.

As mentioned before, this only shows that the architecture works with one node. The scalability with more nodes is discussed in Chapter 4.3.3.

4.3.2 Aggregator

The aggregator triggers the nodes to obtain data. This is needed to comply with requirement R1.8. After the process described in Section 4.3.1, it receives the data from each node. This data then gets unpacked and can be accessed by the processor. To calculate the process time of the aggregator, the delay generated by unpacking the data and copying it to the processor has to be taken into consideration. Other than the measurements in Chapter 4.3.1, these show the process time for each message from all three nodes. The mentioned delay plus the maximum processing time of the nodes results in the total time needed with this architecture.

The conditions for the measurements are similar to Chapter 4.3.1. The duration is three minutes and since only one node was active as well, a total of 5384 measurements are represented per figure.

Parse Data

Firstly, the serialized byte string, that is received from each node, has to be parsed into readable format. Figure 4.8 shows the maximum, minimum and average time it takes to finish this task. Unlike the serialization process, parsing always operates in a pre-allocated message object. Therefore the average of 0.27 ms is already the optimal time to parse the received data.

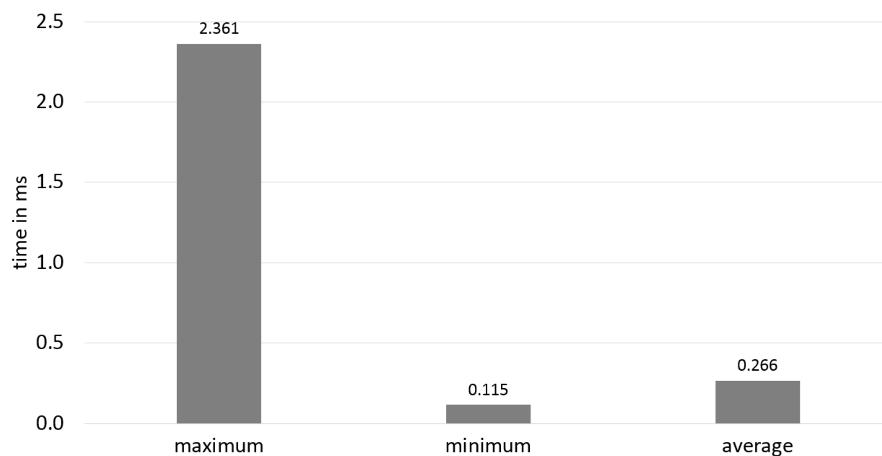

Figure 4.8: The maximum, minimum and average time it takes to parse one message from one of the three nodes.

The maximum time of 2.36 ms can be explained by the way the nodes are implemented. On the processing instance, each node is assigned one thread, which receives and parses the data. While parsing one message, another node may send their data and this thread activates. Depending on the scheduler, the thread that started the parsing process is sent to sleep.

With this explanation and the average case being very close to the minimum, it can be derived that the maximum has no statistical significance. But since this case can still occur, it has to be regarded in order to calculate the worst case scenario.

Copy to Processor

The data parsed then be accessed through the interface IAggregator. Since each node thread on the processing instance already has the parsed message object in memory, the data can be copied to the processor. In order to reduce delay, the function expects pre-allocated memory.

Because this step only consists of one copy instruction, could have been disregarded. But since the same scheduling problems explained in the section describing the parsing process occur, the time has still significance, as shown in Figure 4.9. With a maximum of 2.26 ms and a much lower average of 0.61 ms it is similar to the measurement illustrated in Figure 4.8.

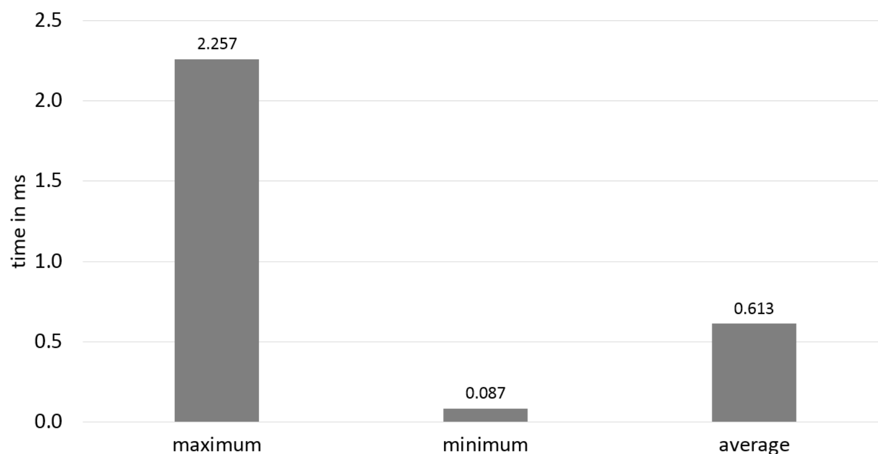

Figure 4.9: The maximum, minimum and average time it takes to copy a video and depth frame to the function caller.

The next step of the processor after acquiring the frames is to calculate the corresponding point cloud out of the depth frame. This process iterates over the copied data and performs a function on each depth point. It was analyzed if doing this step while copying the data will increase the performance. The functionality changes from copying the whole byte array to calculating each cloud point on pre-allocated memory.

Measurements showed, that the first two steps of acquiring the data and then calculating the point cloud takes about 9 ms. With this approach the process time is down to 3.5 ms. In the end it enabled the processor to process two more frames per second than before. As shown in Figure 4.10, it takes an average of 2.18 ms for the point cloud to be calculated and returned to the function caller. Even though it is not part of the architecture, it is a first step to optimize the algorithm.

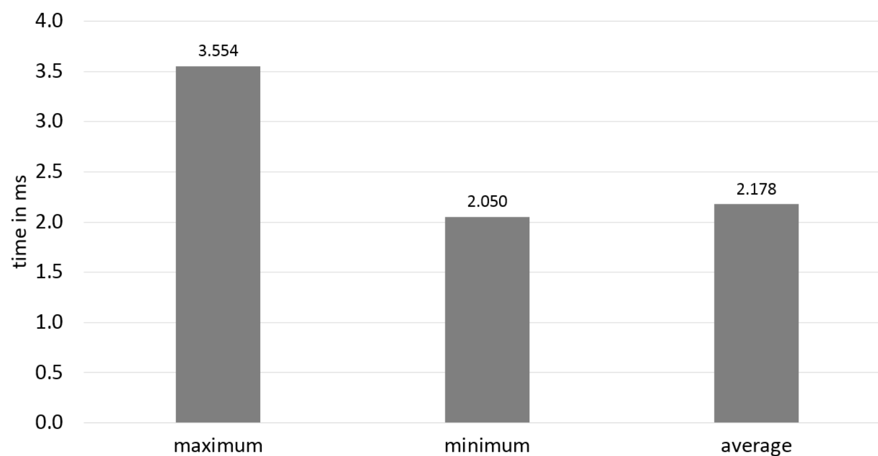

Figure 4.10: The maximum, minimum and average time it takes to calculate and return a point cloud.

The idea of calculating the point clouds distributed on each node was also considered. Before, each would node send the 7.4 Mbit for video plus 4.9 Mbit for the depth frame. With the new approach the depth frame would be converted to a point cloud, which is three time as big[4]. But since the network bandwidth is already exhausted, introducing the extra data to the network would increase the time to send data even more. Therefore this approach was not further contemplated.

[4] A point cloud consists of an x, y and z point. Therefore it's size would be three times 4.9 Mbit, or about 14.7 Mbit.

4.3.3 Processing Speed

The total processing speed of one frame is measured from the point the process is triggered until all data is accessed through the interface IAggregator. As shown in Chapter 4.3.1, the nodes can send their data with a maximum theoretical delay of 18.3 ms, if they are the only sender in the network. The aggregator's delay is 4.6 ms, which can be calculated by adding the maximum measured delays in Chapter 4.3.2. With this results, a soft deadline of 22.9 ms for one node can be guaranteed.

It was already mentioned in the Chapters 4.3.1 and 4.3.2, that even though the statistical outliers have to be considered, the chance of them occurring during at the same time is minimal. In order to determine the real guaranteed delay, the time between the triggering event and obtention through the interface has to be regarded. The measurements also have to include all three nodes to conclude a deadline for this process. The illustrations now represent about 3830 measurements over a span of three minutes. The decline of values within the same timespan can be explained by the higher process time for each frame.

Figure 4.11 shows the result of this measurement. It is visible, that this process takes longer than calculated from each step. With a maximum of about 71.4 ms for one frame, only 14 fps are possible to process. The main reason for these values is the bottleneck of the network bandwidth.

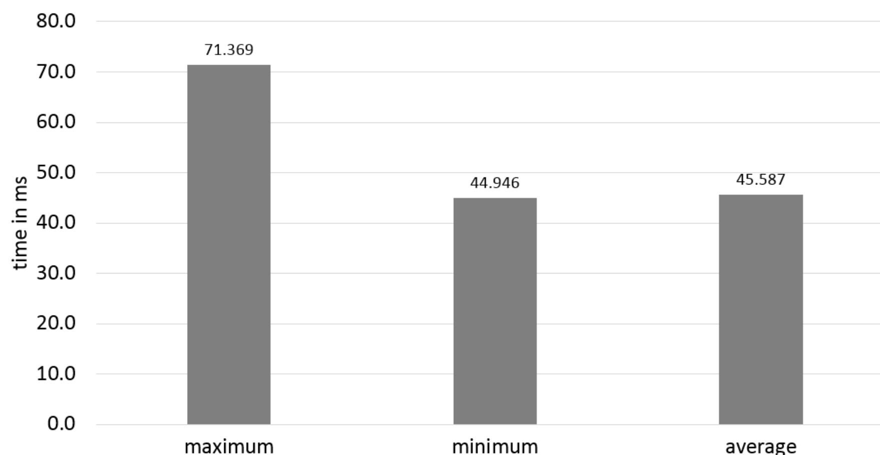

Figure 4.11: The maximum, minimum and average process time using three nodes with this architecture.

In order to analyze it, the time for one node to send its data was measured, when all three nodes are connected. Figure 4.12 illustrates that a delay of 41.9 ms is possible during transfer. Since the processing instance has to receive and parse three messages at once, it also becomes as a bottleneck. The three messages are sent at the same time and are processed in different threads. These threads have to get processing time assigned for the CPU by the scheduler. All these factors play a role in why the time to send/receive one data package is so high with multiple nodes in the network.

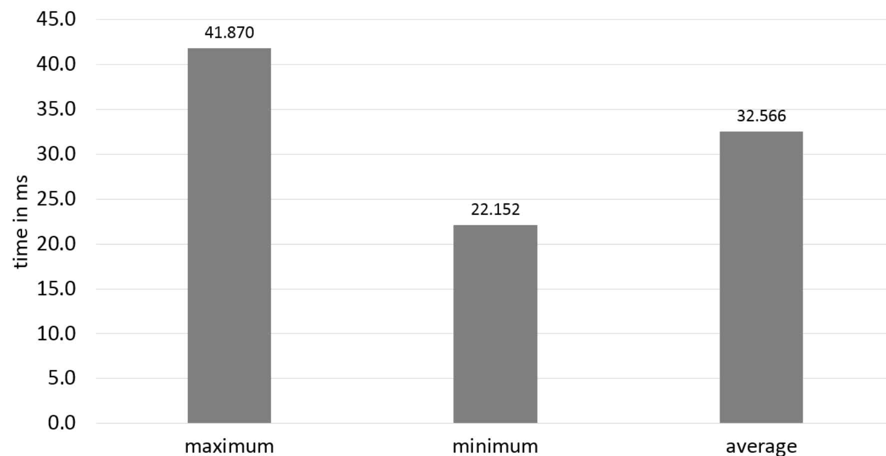

Figure 4.12: The maximum, minimum and average time for one node to send its data with three nodes in the network.

4.4 Communication

This Section explains the pattern used for communication between the aggregator and nodes. As required by R1.8 in Table 1.1 on page 4, the processing instance has to be able to request the data on demand. In other words, the aggregator calls a function on the nodes. These execute the steps described in Chapter 4.3.1 and return the acquired data. This resembles the RPC pattern presented in Chapter 2.2.

As mentioned in the chapter, the communication can either be synchronous or asynchronous. In this context, the synchronous example can be disregarded. With this approach the aggregator would block while one node is acquiring and transferring the data. As measured in the Chapter 4.3.1, this process takes about 14.9 ms. Therefore a delay of this amount would always

exist between the data of each node. In the end, this would result in the same problem shown in Figure 1.1.

Asynchronous communication is because of that the better solution. The aggregator starts the execution of a remote function call for each node and waits until all data is delivered. The delay between each frame is only affected by the delay introduced by the transmission of the trigger message. But since requirement R1.6 presupposes that no other communication takes place over the network during the RPCs, this delay should be minimal.

But the corresponding communication pattern leaves the problem of synchronization between the nodes open. The nodes do not know when the aggregator will acquire new data. Because of that, they do not know if they have enough time to start a synchronization process. As already mentioned, R1.6 in Table 1.1 on page 4 requires the whole bandwidth for the data transfer during the RPCs. If the nodes are currently synchronizing their time, the transfer has to be delayed until this process is finished.

To solve this problem, a distributed RPC can be used. With this pattern, the aggregator sends the trigger message only to one master node, the DRPC server. This node then distributes the message to all other nodes. The nodes then return the data directly to the processing instance. But since the DRPC server knows that a RPC has just been executed, it can trigger the synchronization process. With a confirmation message from each node after the RPC, it also knows that no other messages are currently carried out over the network.

A corresponding communication sequence of the whole process can is shown in Figure 4.13. The steps `executeLocalFunction` and `processResult` are the processes described in Chapter 4.3.1 and Chapter 4.3.2 respectively. The synchronization process is presented in Chapter 5. It also presents the election algorithm used to determine the DRPC server out of the nodes.

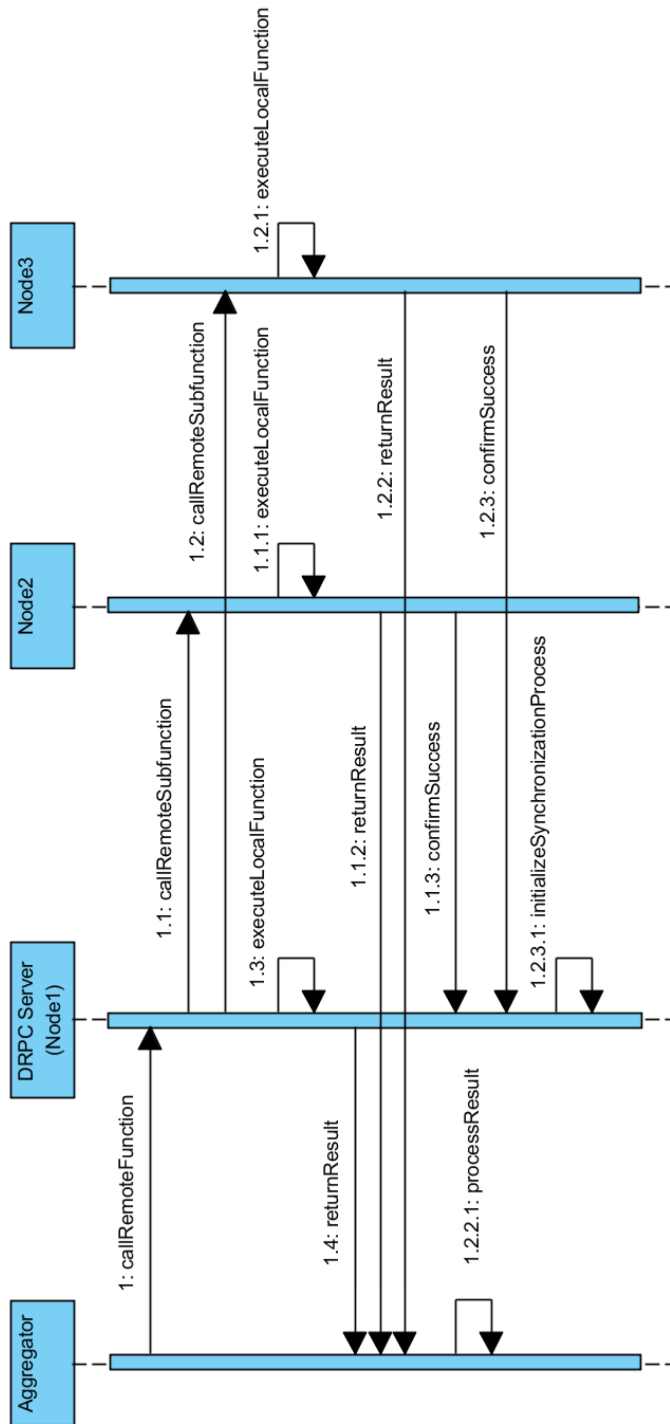

Figure 4.13: Communication model for the presented architecture.

5 Network Design

This chapter presents the underlying network, on which the architecture is based on. Its main focus lies on the synchronization, which includes the election of a DRPC server.

Because of the large bandwidth requirement, configuring the switch and devices correctly also contributes to a better performance. The configuration of the networked devices is explained in Chapter 4.3.1. The most important step is to use jumbo Ethernet frames. Configuring the MTU to 9000 bytes increases the effective data throughput, since the overhead generated through protocol header is reduced.

Other configurations like adjusting the TCP windows size[28] were considered but found to be out of scope for this work. It can increase the performance, but other approaches like compressing the data seem more effective. The comparison between both optimization approaches can be a topic in future works.

5.1 Election

The architecture requires a master node within the network, which triggers the data acquisition and synchronization process. Compared to other distributed applications, this only needs to elect its master once. If a node fails, the application cannot operate and shuts down, therefore a re-election does not have to occur.

This also allows the nodes to elect a master node before the aggregator starts the data acquisition. Because of that the speed of the election process is not of importance. Out of these reasons, an election algorithm that is simple to implement is used. This resulted in a variation of the Bully election algorithm presented by Garcia-Molina [32].

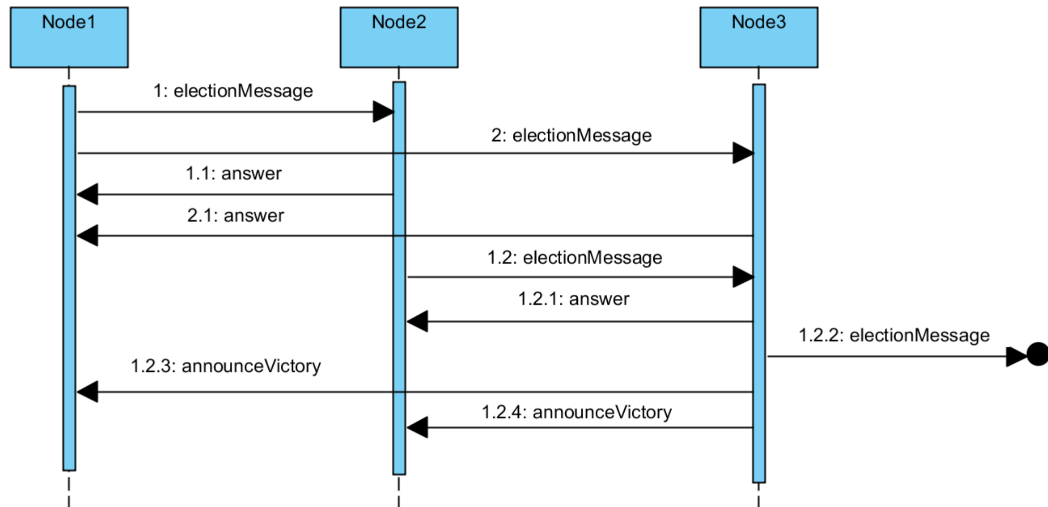

Figure 5.1: Message sequence of the Bully algorithm.

The message flow of the Bully algorithm is shown in Figure 5.1. When a new process, in this case Node1, enters the network, it sends an election Message. If the receiver has a higher ID than the sender, it answers and starts its own election process as shown with messages 1.1 and 1.2. When the node with the highest ID wants to start the election process, it receives no answers as illustrated with message 1.2.2. This node then broadcasts its victory, upon receiving the other nodes memorize him as the master.

Since the nodes have no static ID assigned to them, the modified algorithm is based on the startup time of each node. Another difference is, that the answer already includes the message that a master has been chosen. In terms of scalability, this modification eliminates the victory messages of the Bully algorithm. But since the required bandwidth will not allow more nodes in the network, this is only a small factor of its advantage. If more nodes can be added, it will most likely not exceed ten. The related project OmniKinect [2] uses nine cameras in its setup, which generate a sufficient recreation of a user from all angles. With this small amount of devices in the network, scalability can be neglected as long as it is not exponential.

The algorithm with this variations is illustrated in Figure 5.2. At the startup of each node an election broadcast is sent over the network. The node then waits for an answer from an already started node. The answer message includes the information, whether the replying node is the master or not. The first node to send this broadcast will not receive any answers and elects itself as the master.

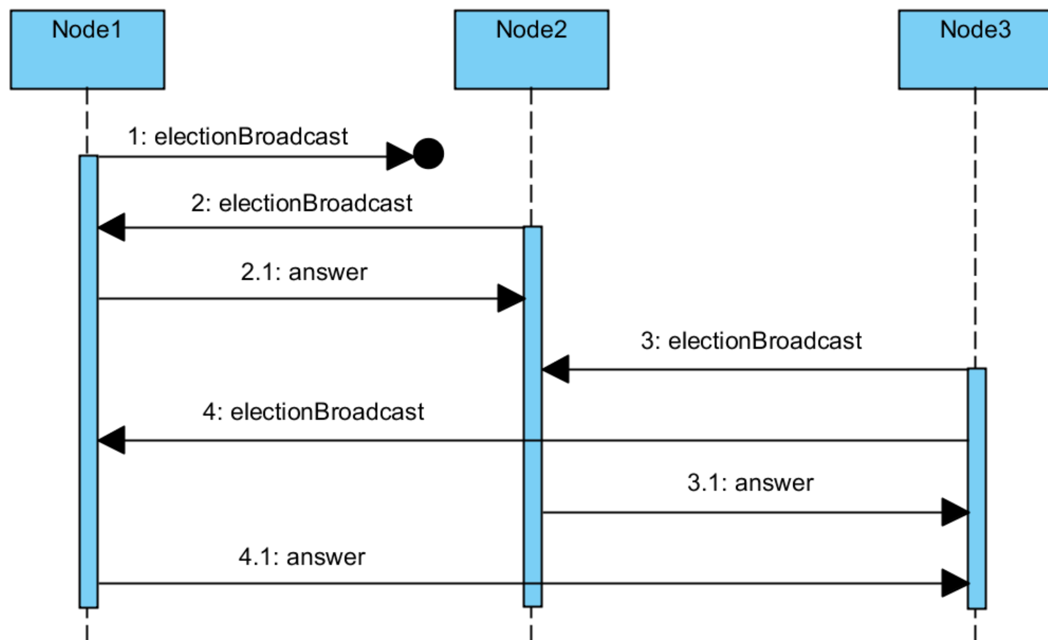

Figure 5.2: Message sequence of the modified algorithm.

As mentioned in Chapter 2.4.3 the fault tolerance of the algorithms used in the network is of importance. The timeout interval, after which the node elects itself, is set to 1 ms. In theory it is possible for two nodes to start up with a delay lower than 1 ms to send each other answers. This would result in a network without an elected master. But since the node applications are started with a deployment script, a built in delay between the start of each node will prevent this problem from occurring.

Another source of error is a packet loss during the election process. The used transport protocol UDP does not offer a retransmission mechanism. If an election broadcast is lost with an already elected master in the network, it can be possible that a second node will elect itself master after the timeout. This is prevented with the answer messages. Should this occur, one node would receive an answer with two nodes responding as master and it triggers a

re-election. The same mechanism helps with the loss of `answer` messages, which results in the same scenario mentioned. In the case of only one dropped `answer` message containing the information of the leader, the node can derive from the other `answer` that another node as already been elected.

5.2 Synchronization

In this chapter the synchronization process used is presented. First the suiting algorithm is determined. Later the implemented algorithm is tested against the requirement from Table 1.1 (see page 4).

5.2.1 Algorithm

In Chapter 3, the Precision Time Protocol (PTP) is described as an already existing solution to this problem. Since it is an established IEEE standard, its guaranteed time synchronization in nanosecond resolution motivates the usage of this approach. With PTP daemon (PTPd) an open source implementation of this standard exists. The problem is, that PTPd is a program which runs in the background on a node. Therefore the architecture has no control when the nodes synchronize their times and requirement R1.6 may not be met.

The other solution is the in Chapter 2.4.2 presented Berkeley algorithm. Compared to PTP it offers a lower resolution, but can be implemented easier because of its simpler message sequence. This allows the architecture to trigger a synchronization process and fulfill requirement R1.6.

Figure 5.3 illustrates the message sequence of this process. First, the master broadcasts the synchronization trigger to all other nodes. While the master waits for the responses containing the timestamp of each node, it saves its own timestamp and starts measuring the round-trip time (RTT). With the RTT it can approximate the time each node captured its time and add the RTT divided by two to its timestamp. This provides a more accurate synchronization as measurements have shown. From all the timestamps, an average is calculated. Each node then receives the offset from the calculated average to its sent timestamp. At the end, all participating devices adjust their time by the received offset.

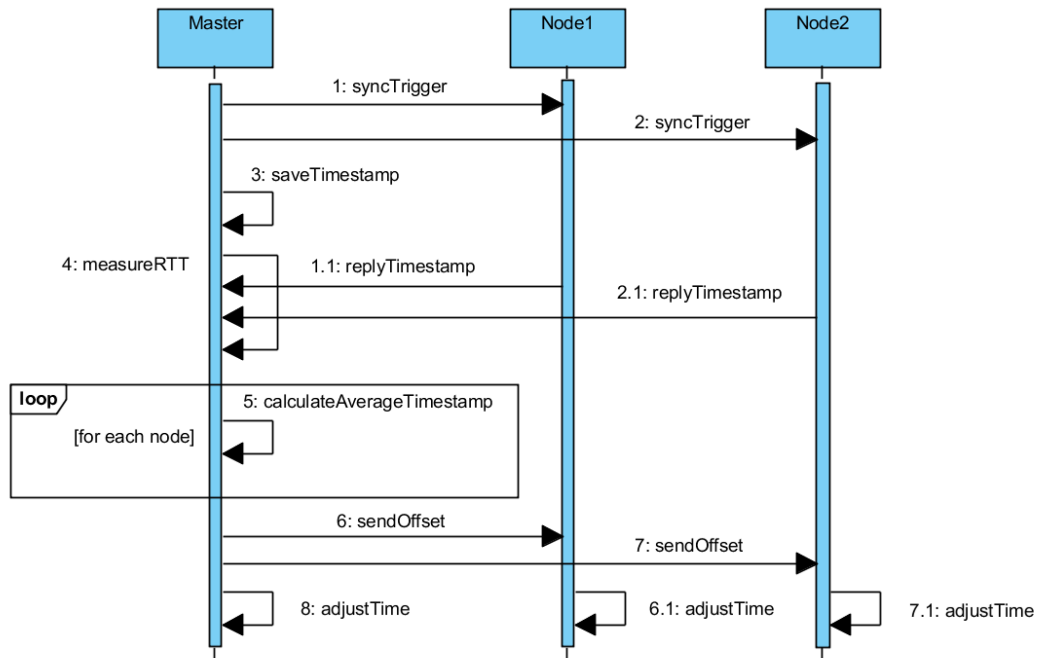

Figure 5.3: Message sequence of the implemented synchronization algorithm, based on Berkeley's algorithm.

As with all other messages sent over the network, Google's protobuf is used to serialize the data. The used message is presented in Chapter 5.3. In the following section the suitability for the illustrated algorithm in this architecture is discussed.

This algorithm consists of two states, timestamp collection and clock adjustment. If the message 1.1 or 2.1 in Figure 5.3 is lost, the master node would be in a deadlock[1]. The same situation can happen on the nodes while waiting for the offset. To prevent this, a timeout has to be used where the process will be set back to the initial state of waiting for the trigger.

Since the synchronization is triggered after the acquisition (see Figure 4.13 on page 33) and the clock drift occurs slowly over time [17], missing it is acceptable.

[1]A process is in a deadlock when it is waiting on an event that will never occur.

5.2.2 Test of the Algorithm

In this section the presented algorithm gets tested against the requirements of Table 1.1 from page 4. Specifically the time to finish one synchronization process and the resolution is of importance.

Processing Time

The process time is important to offer an optimal frame rate. When the processing instance wants to obtain new data, it should not have to wait until the synchronization is finished. Because of requirement R1.6 it would have to, if this is the case.

Figure 5.4 illustrates the measured duration of this process. With about 1.4 ms maximum processing time it will not hinder the application to operate with an optimal frame rate. Combined with the maximum processing time on one node of 14.9 ms (see Chapter 4.3.1), it takes about 16.3 ms to send the obtained data and synchronize the global time.

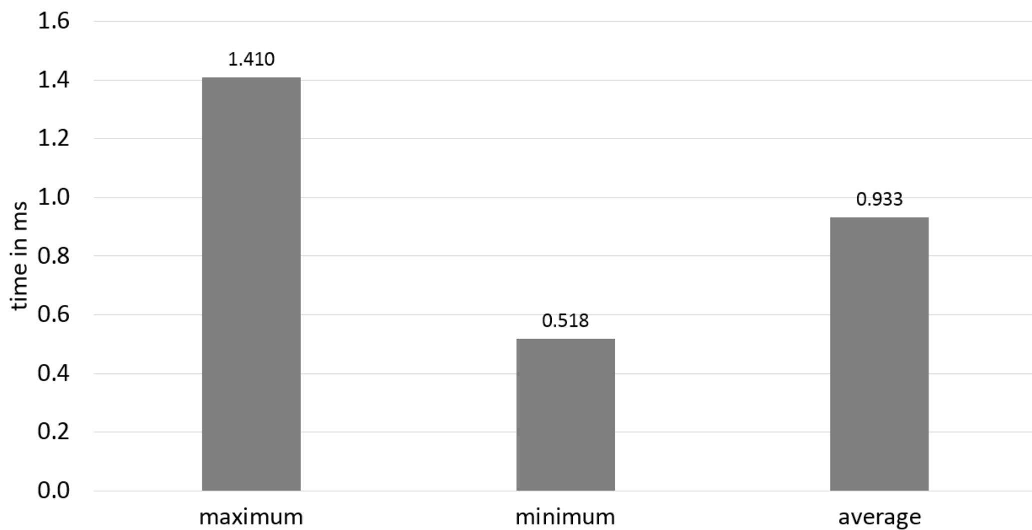

Figure 5.4: Message sequence of the implemented synchronization algorithm, based on Berkeley's algorithm.

One time consuming part during this process is the transmission. As seen in Figure 5.3, this amounts to three one-way delays (OWD). The total amount of what one OWD amounts to could not be measured, since it was too small. Therefore it is less than one microsecond. This is possible because the network is free during this process and con solely use for the synchronization messages.

Another part is the serialization. For each message sent between nodes in Figure 5.3 a protobuf message has to be serialized. This process is more time consuming and amounts to nearly the whole time. Since the nodes operate with the same processor architecture, the step of serialization could be skipped. If this process would ever needed to be optimized, this is where time can be saved.

5.2.3 Resolution

Next the resolution of the synchronization algorithm is analyzed. Requirement R1.3 specifies that the delay between nodes must not exceed 1 ms.

The delay was measured at two points, once with the offset that is sent out during the synchronization process and also at the aggregator with the timestamps. Former will provide information about the minimal offset that occurs during this process. The remaining microseconds is the resolution of this algorithm. The latter gives information about the effective delay between each capture. This helps to check requirements R1.4.

In either case the maximum delay is about 0.1 ms and therefore fulfills both requirements. The delay mainly consists of the approximated RTT. Even though the OWD is relatively small, the synchronization of the messages varies, as seen in Chapter 4.3.1. The OWD is calculated by simply dividing the RTT by two. But the RTT consists of both, transmission and serialization time. With the small OWD, this only calculates about half the serialization and parse time.

5.3 Network Control Communication

Previous chapters showed the control mechanisms that run within the network. During these, it is mentioned that the messages are serialized with Google's protobuf. The message is shown in Listing 5.1.

The message has three headers: ELECTION, SYNC and TRIGGER. The former two are self-explanatory, latter is the trigger message for the data acquisition. The `leader` field is used to reply whether the node is the leader or not during the election process described in Chapter 5.1.

During synchronization the message can have different sub-header, like SEND, REPLY and ADJUST. Former is set during the synchronization broadcast, illustrated as message 1 and 2 in Figure 5.4. The reply messages 1.1 and 2.1 then set the sub-header REPLY, which includes the `timestamp`. Lastly, ADJUST indicates that this message contains the offset in the `offset_usec` field.

The communication, contrary to the data transfer in the architecture, is based on UDP. Since the network bandwidth is free during the transmission of these messages, the probability of data loss or collision is minimal. The messages themselves are also small. The largest ones are the synchronization REPLY or ADJUST with a maximum of 10 bytes[2].

[2]This contains 1 byte for the `type`, 1 byte for the `sync_mode` and 8 bytes for either the `timestamp` or `offset_usec`.

```
syntax = "proto3";

message NetworkMessage {
        enum Type {
                ELECTION = 0;
                TRIGGER = 1;
                SYNC = 2;
        }

        enum SyncMode {
                SEND = 0;
                REPLY = 1;
                ADJUST = 2;
        }

        Type type = 1;

        bool leader = 2;

        // synchronization
        uint64 timestamp = 3;
        SyncMode sync_mode = 4;
        int64 offset_usec = 5;
}
```

Listing 5.1: Protobuf message for network communication between nodes.

6 Conclusion

The goal of this thesis was to present an infrastructure, that supports the user recreation algorithm presented in [1]. In order to measure its functionality, the requirements from page 4 in Table 1.1 have been established. While discussing the architecture and network design, it was shown that all requirements are met with the presented outcome of this work.

R1.1 is not fully met, as it does not offer the full bandwidth, but the application can be operated with the available 1 Gb/s. This also influenced R1.5, but as shown in Chapter 4.3.1, it theoretically supports the required frame rate with higher bandwidth. That the required processing power for the nodes is met can also be found in that chapter. R1.3 and R1.4 are both shown to be fulfilled in Chapter 5.2. The implementation of the last three requirements R1.6, R1.7 and R1.8 are presented in Chapter 4.

Figure 6.1: Picture showing two frames of the recreation algorithm during a fast movement.

In order to fully test if this infrastructure solves the synchronization problem described in [1], the application was rewritten to support it. The result showed clearly, that the problem has been solved. Even fast movements do not provoke a deformation as illustrated in Figure 1.2 on page 2. A similar movement resulted in the recreated models visible in Figure 6.1.

During the discussion of each architecture and network design choice, different approaches for optimization have been addressed. The most promising, which will most likely drastically improve the performance of this application, is to process part of the data distributed. It is mentioned in Chapter 4.3.1 that calculating the point cloud in memory already increased the frame rate.

In order to further increase the performance, distributing the next step of finding triangles within the point cloud, could be distributed. This would most likely require nodes with more processing power. That would result in a setup similar to the one Microsoft uses in its Holoportation project [21].

This approach is planned to be implemented next, in order to further improve on this solution.

Bibliography

[1] Iwer Petersen. Kollaboration im virtuellen Team: Grenzen des Avatar-Realismus bei verteilter Echtzeit-Rekonstruktion. Master's thesis, HAW Hamburg, Germany, 2016.

[2] Bernhard Kainz, Stefan Hauswiesner, Gerhard Reitmayr, Markus Steinberger, Raphael Grasset, Lukas Gruber, Eduardo Veas, Denis Kalkofen, Hartmut Seichter, and Dieter Schmalstieg. Omnikinect: Real-time dense volumetric data acquisition and applications. In *Proceedings of the 18th ACM Symposium on Virtual Reality Software and Technology*, VRST '12, pages 25–32, New York, NY, USA, 2012. ACM.

[3] R. A. Gupta and M. Y. Chow. Networked control system: Overview and research trends. *IEEE Transactions on Industrial Electronics*, 57(7):2527–2535, July 2010.

[4] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, Oct 1991.

[5] Don Anderson and Dave Dzatko. *Universal Serial Bus System Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2001.

[6] Openkinect: Protocol documentation. https://openkinect.org/wiki/Protocol_Documentation#Relevant_Bits_of_Code. Accessed: 2017-11-09.

[7] Bengel G. *Grundkurs Verteilte Systeme: Grundlagen und Praxis des Client-Server und Distributed Computing*. Springer Vieweg, 2014.

[8] w3schools: Introduction to xml. https://openkinect.org/wiki/Protocol_Documentation#Relevant_Bits_of_Code. Accessed: 2017-11-09.

[9] Json: Introducing json. http://www.json.org/. Accessed: 2017-11-09.

[10] Protocol buffers: Developer guide. https://developers.google.com/protocol-buffers/docs/overview. Accessed: 2017-11-09.

[11] The definitive serialization performance guide. https://aloiskraus.wordpress.com/2017/04/23/the-definitive-serialization-performance-guide/. Accessed: 2017-11-09.

[12] Serialization performance comparison. https://maxondev.com/serialization-performance-comparison-c-net-formats-frameworks-xmldatacontractserializer-xmlserializer-binaryformatter-json-newtonsoft-servicestack-text/. Accessed: 2017-11-09.

[13] Protocol buffers: Language guide (proto3). https://developers.google.com/protocol-buffers/docs/proto3. Accessed: 2018-01-11.

[14] Distributed rpc. http://storm.apache.org/releases/1.0.3/Distributed-RPC.html. Accessed: 2017-10-24.

[15] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, C-36(8):933–940, Aug 1987.

[16] Epochconverter: What is epoch time? https://www.epochconverter.com/. Accessed: 2017-11-09.

[17] M. A. Lombardi. The evolution of time measurement, part 2: quartz clocks [recalibration]. *IEEE Instrumentation Measurement Magazine*, 14(5):41–48, October 2011.

[18] Windows time service technical reference. https://technet.microsoft.com/en-us/library/cc773061%28WS.10%29.aspx. Accessed: 2017-11-29.

[19] David L. Mills. *Computer Network Time Synchronization: The Network Time Protocol on Earth and in Space, Second Edition*. CRC Press, Inc., Boca Raton, FL, USA, 2nd edition, 2010.

[20] R. Gusella and S. Zatti. The accuracy of the clock synchronization achieved by tempo in berkeley unix 4.3bsd. *IEEE Transactions on Software Engineering*, 15(7):847–853, Jul 1989.

[21] Sergio Orts-Escolano, Christoph Rhemann, Sean Fanello, Wayne Chang, Adarsh Kowdle, Yury Degtyarev, David Kim, Philip L. Davidson, Sameh Khamis, Mingsong Dou, Vladimir Tankovich, Charles Loop, Qin Cai, Philip A. Chou, Sarah Mennicken, Julien Valentin, Vivek Pradeep, Shenlong Wang, Sing Bing Kang, Pushmeet Kohli, Yuliya Lutchyn, Cem Keskin, and Shahram Izadi. Holoportation: Virtual 3d teleportation in real-time. In *Proceedings*

of the 29th Annual Symposium on User Interface Software and Technology, UIST '16, pages 741–754, New York, NY, USA, 2016. ACM.

[22] Qun Li and D. Rus. Global clock synchronization in sensor networks. *IEEE Transactions on Computers*, 55(2):214–226, Feb 2006.

[23] Terrell R. Bennett, Nicholas Gans, and Roozbeh Jafari. A data-driven synchronization technique for cyber-physical systems. In *Proceedings of the Second International Workshop on the Swarm at the Edge of the Cloud*, SWEC '15, pages 49–54, New York, NY, USA, 2015. ACM.

[24] P. Sommer and R. Wattenhofer. Gradient clock synchronization in wireless sensor networks. In *2009 International Conference on Information Processing in Sensor Networks*, pages 37–48, April 2009.

[25] Christoph Lenzen, Philipp Sommer, and Roger Wattenhofer. Optimal clock synchronization in networks. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 225–238, New York, NY, USA, 2009. ACM.

[26] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. *SIGOPS Oper. Syst. Rev.*, 36(SI):147–163, December 2002.

[27] Hpe officeconnect 1810 switch series. https://h20195.www2.hpe.com/v2/getpdf.aspx/c04164471.pdf. Accessed: 2018-01-11.

[28] Jon Postel. Transmission control protocol. STD 7, RFC Editor, September 1981. http://www.rfc-editor.org/rfc/rfc793.txt.

[29] J. Postel. User datagram protocol. STD 6, RFC Editor, August 1980. http://www.rfc-editor.org/rfc/rfc768.txt.

[30] Jon Postel. Internet protocol. STD 5, RFC Editor, September 1981. http://www.rfc-editor.org/rfc/rfc791.txt.

[31] C. Hornig. A standard for the transmission of ip datagrams over ethernet networks. STD 41, RFC Editor, April 1984.

[32] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):48–59, Jan 1982.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 12. Januar 2018 Dennis Kirsch