Philipp Ferner

Clock synchronization for a real-time communication using Ethernet

Master's Thesis

Graz University of Technology

Institute of Automation and Control
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Martin Horn

Supervisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Daniel Watzenig

Graz, September 2019

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

Computer simulation are a well-known tool for verification and testing in vehicle design. With increasing complexity of vehicles, computer simulation got more sophisticated as well. One solution to cope with this problem is to distribute the task into various subsystems and nodes. This is called co-simulation. The overall simulation model is split into certain components and sub-models. Each of these components simulate their own tasks but may depend on each other. While this can greatly reduce the complexity of the simulation, new challenges arise in the field of communication.

Various Co-Simulation solutions define interfaces and procedure to cope with these new problems. One of the main players in this field is DCP[2] from the *Modelica Association Project*. It is designed for real-time and non-real-time operations. One of these challenges is clock synchronization. However, the current DCP specification does not handle clock synchronization natively but relies on an external mechanism.

This work evaluates the need of clock synchronization for DCP. Thus, the impact of clock divergence is analyzed. Furthermore, the consequences to the simulation result given unsynchronized nodes is evaluated as well. A clock synchronization mechanism compatible with the current DCP draft should be suggested and implemented. Given the flexibility of DCP, a software-based approach is used. PTP is a well-known and widely adapted clock synchronization method satisfying our criteria. The suggested changes are added to the reference DCP implementation DCPLib[3].

The implementation is tested in a simple simulation composed of two DCP slaves, one DCP master and an external environment simulation. It is used to collect the required data and verify the changes. The test environment

[2]https://dcp-standard.org
[3]https://github.com/modelica/DCPLib

features a simple Lane Keep Assist System (LKAS) based on a controller of the “Stanford Racing Team”. Multiple test runs has been conducted to compare simulation results with synchronized and unsynchronized nodes. Various degrees of clock drifting have been measured as well. The simulation is kept as simple as possible but is especially tailored to emphasize errors caused by clocks losing synchronization.

Acknowledgement

I specially want to thank Virtual Vehicle as well as their employees. Virtual Vehicle offered me all the help and tools needed to accomplish this work. This help also includes various hints and suggestions from countless people of the company. Among them I particularly want to give thanks to my supervisor Dipl.-Ing. Martin Krammer. He was a driving force behind of specification of DCP. His knowledge and input around the topic Co-Simulation and scientific writing was always appreciated.

In this context, i also want to highlight the Modelica Association. They are the holding company of DCP and FMI. The work is based on an preview implementation of DCP provided by Modelica.

Finally I also want to thank the Institute of Automation and Control at the Graz University of Technology. They helped during the development of the sample controller and provided crucial information. With thanks to Univ.-Prof. Dipl.-Ing. Dr.techn. Daniel Watzenig the communication between the IRT and the Virtual Vehicle worked flawless.

Zusammenfassung

Computer Simulationen sind wesentlich für die Planung und das Testen von Fahrzeugen. Um mit der steigenden Komplexität von Fahrzeugen mithalten zu können, haben sich auch Simulation Tools weiterentwickelt. Eine dieser Weiterentwicklungen versucht die Komplexität der Simulation auf mehrere und einfachere Komponenten zu verteilen. Dies nennt sich "Co-Simulation". Jede dieser Komponenten simuliert nur einen kleinen Teil, welche jedoch untereinander abhängig sind. Erst durch das Zusammenführen ergibt sich ein fertiges Resultat. Während sich die Simulation dadurch vereinfacht, entstehen neue Herausforderungen im Bereich der Datenübertragung.

Dazu werden von eigenen Co-Simulation Protokollen Modelle, Abläufe und notwendige Schnittstellen definiert. Ein Vorreiter in diesem Bereich ist das DCP Protokoll[4], von der *Modelica Association*. DCP wird für den Echtzeit und Nicht-Echtzeit Betrieb entwickelt. Eine dieser Herausforderungen ist auch die Zeitsynchronisation. Die aktuelle Spezifikation von DCP definiert dazu keinen Mechanismus, sondern verweist auf eine externe Lösung.

Diese Arbeit untersucht die Notwendigkeit der Zeitsynchronisation für DCP. Dazu wird der Einfluss vom "Clock Drift" analysiert. In weiterer Folge wird der Unterschied zwischen Simulationen mit aktiver und inaktiver Zeitsynchronisation gemessen und ausgewertet. Weiters wird eine Synchronisationsmethode implementiert, welche mit der aktuellen DCP Spezifikation kompatibel ist. Aufgrund der hohen Flexibilität von DCP wird dazu eine softwarebasierende Lösung verwendet. PTP ist ein weit verbreiteter Standard, welcher die notwendigen Kriterien erfüllt. Die praktische Umsetzung basiert auf der DCP Referenzimplementierung DCPlib[5].

[4]https://dcp-standard.org
[5]https://github.com/modelica/DCPLib

Zum Testen der Implementierung wird eine einfache Simulation verwendet. Die Simulation besteht aus zwei DCP Slaves, einen DCP Master sowie einer externen Umgebungssimulation. Es wird ein Lane Keep Assist System (LKAS) auf Basis eines Reglers vom “Stanford Racing Team” simuliert. Dabei wurde die Simulation so gestaltet, dass Auswirkungen aufgrund fehlender Zeitsynchronisation besonders sichtbar sind. Es wurden verschiedene Messungen durchgeführt um Ergebnisse mit aktiver und inaktiver Zeitsynchronisation zu vergleichen.

Contents

Abstract iii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem Description . . . 2
- 1.3 Objectives . . . 2

2 Related Work 5
- 2.1 Definitions . . . 5
  - 2.1.1 Clock offset . . . 5
  - 2.1.2 Clock skew and clock drift . . . 6
  - 2.1.3 Clock skew (circuit) . . . 6
  - 2.1.4 Network latency/delay . . . 7
  - 2.1.5 Real-time . . . 8
- 2.2 Open System Interconnection (OSI) . . . 8
  - 2.2.1 7 Layers of the OSI model . . . 9
- 2.3 Clock synchronization . . . 11
  - 2.3.1 Network Time Protocol (NTP) . . . 11
  - 2.3.2 Precision Time Protocol (PTP) . . . 14
- 2.4 Ethernet . . . 17
  - 2.4.1 User Datagram Protocol (UDP) . . . 17
  - 2.4.2 Nondeterministic behavior of Ethernet . . . 18

3 Method 21
- 3.1 Overview of Co-Simulation . . . 21
- 3.2 Distributed Co-Simulation Protocol (DCP) . . . 22
  - 3.2.1 Multichannel support . . . 22
  - 3.2.2 Master-slave architecture . . . 23
  - 3.2.3 Protocol Data Unit (PDU) . . . 24

3.2.4 Operation and state machine . . . 27
3.2.5 Heartbeat mechanism . . . 30
3.3 Clock synchronization for DCP . . . 30
3.3.1 PTP over DCP . . . 32
3.3.2 New PDU types . . . 33

4 Implementation **37**
4.1 Implementation environment . . . 37
4.2 Additions to DCP . . . 38
4.2.1 Additional synchronization PDUs . . . 38
4.2.2 Important DCP states for synchronization . . . 39
4.2.3 Synchronization trigger interval . . . 40
4.2.4 Integration of PTP in DCP master . . . 40
4.2.5 Integration of PTP in DCP Slave . . . 42
4.3 POSIX timestamping API . . . 44
4.3.1 Enabling timestamping API . . . 45
4.3.2 Retrieving timestamps . . . 48
4.4 ASIO modifications . . . 54
4.4.1 Enabling timestamping API . . . 54
4.4.2 Retrieving timestamps . . . 57

5 Test Environment **61**
5.1 Simulation Layout . . . 61
5.2 Used hardware . . . 62
5.3 Simulation properties . . . 63
5.3.1 Virtual road . . . 63
5.3.2 Controller . . . 64
5.4 Test results . . . 66
5.4.1 Effects of over-/undersampling . . . 69
5.4.2 Propagation of error . . . 71

6 Conclusion and Outlook **75**
6.1 Conclusion . . . 75
6.2 Outlook . . . 76

Bibliography **79**

List of Figures

2.1 Clock Offset 5
2.2 Clock Skew 6
2.3 Clock Skew (Circuit) 7
2.4 Network delay and latency 7
2.5 System separation of OSI Layers 9
2.6 Layers of the OSI Model 10
2.7 NTP Stratas with decreasing accuracy 11
2.8 Round-trip-time used in NTP 12
2.9 Synchronization process of PTP 15

3.1 DCP master-slave ethernet architecture 23
3.2 DCP State machine 28
3.3 DCP Heartbeat functionality 31
3.4 PTP Messages 32

4.1 Comparison of step and synchronize interval 44
4.2 Timeline of receiving messages 51
4.3 Timeline for sending messages. 54

5.1 Block Diagram of the test environment 62
5.2 Aerial perspective of the virtual road. 63
5.3 Kinematic model of a automotive. 64
5.4 Clock difference over time 66
5.5 Picture of the simulation rendering 68
5.6 Steering angle during a simulation run. 68
5.7 Difference of steering angles between two runs 69
5.8 Comparison of an ideal clock and a skewed clock 70
5.9 Overlay reference run and steering angle difference 72
5.10 Overlay reference run and steering angle difference 3000 steps 73

List of Tables

2.1 Important NTP Fields 13
2.2 Important PTP Fields 16
2.3 UDP Header 18

3.1 PDU groups with data fields 25
3.2 PTP messages encapsulated in DCP PDUs 34
3.3 Possible values for `sync_op` field 34
3.4 Suggested PDUs encapsulating PTP 35
3.5 Modified `INF_sync` PDU 36

5.1 Steering angle difference between test runs. 71
5.2 Steering angle difference between test runs with limited drift. 73

List of Listings

4.1 Masters sync invocation 40
4.2 Sending sync PDUs 41
4.3 Master receive sync response 41
4.4 Slave receive sync command 42
4.5 Offset applied to correct slaves clock 44
4.6 Signature of POSIX function `ioctl` 45
4.7 Enable hardware timestamping 46
4.8 Signature of POSIX function `setsockopt` 47
4.9 Prepare kernel to collect timestamps in kernel space 47
4.10 Signature of POSIX functions `recv` and `send` 48
4.11 Signature of POSIX function `recvmsg` 49
4.12 Control Message Header object and interface 49
4.13 Retrieval of the *socket time* for incomming data 50
4.14 Retrieval of the *socket time* for outgoing data 52
4.15 ASIOs interface for `ioctrl` access. 55
4.16 Invocation of ASIOs wrapper for POSIXs Input/Output control function 56
4.17 ASIOs interface to set socket options 56
4.18 Invocation of ASIOs wrapper to set socket options 56
4.19 Signature of internal ASIO wrapper functions to receive data 57
4.20 Updated signature of internal ASIO wrapper functions to receive data 57
4.21 Signature of ASIOs receive function 58
4.22 Retreive timestamp info using new ASIO interface 58

List of Acronyms

DCP Distributed Co-Simulation Protocol

CSMA-CD Carrier Sense Multiple Access – Collision Detection

FMI Functional Mock-up Interface

HiL Hardware in the Loop

LKAS Lane Keep Assist System

NTP Network Time Protocol

OSI Open System Interconnection

OWD One-way-delay

RTT Round-trip-time

PDU Protocol Data Unit

PTP Precision Time Protocol

TCP Transmission Control Protocol

UDP User Datagram Protocol

1 Introduction

1.1 Motivation

For vehicle design and development, computer simulation plays a major role. It helps to speed up the process, as well as identify flaws as soon as possible. Simulations are used at the earliest state of the development process till the product launch. However a modern vehicle composes of multiple ECUs and components so that a single simulation module cannot project the real world accurately. Hence, the various components are designed and tested using their own simulation model. These models are focusing on their own tasks and problems which needs to be solved. That is to say on their specific domain. Multiple software emerged to handle the various use cases, however they tend to work poorly outside their domain. In addition, nowadays vehicle are even more complex and many components started to influence and interact with each other. Thus, the challenges to simulate specific components and behaviors increases as well. For example, we cannot rely on a "steering while breaking" simulation without considering an ABS module.

One solution to cope with these problems is to distribute the simulation to multiple but connected devices. Each device works independent of another, but share the same environment and the same simulation state. This is called Co-Simulation. This allows the simulation model to focus on its actual task and domain, as well as consider others models. While this greatly enhances the overall simulation result and model new challenges rises.

For one there is need for standardization of protocols and interfaces. This is required, so that each model can exchange data with each other. Next, some kind of master to control and observe the simulation is needed. Finally, the master and the underlying infrastructure needs to handle the data exchange

and time synchronization. It needs to be guaranteed, that the all nodes share the same information and are in the same simulation step. One such standard is DCP. It defines Protocol Data Unit (PDU) for communication using various channels like Ethernet or CAN. These PDUs allows the master to control its slaves, as well as enables a defined communication between the slaves to exchange simulation data.

1.2 Problem Description

For time synchronization the current DCP draft does not specify or suggest anything. Currently, it only states that the operator must guarantee synchronized nodes if such a behavior is needed. One example where synchronization would be required is a feedback control loop split on two devices. This is especially important in an real-time environment like a HiL scenario. Another challenging aspect of clock synchronization in terms of DCP is that the simulation steps do not need to be fixed for each participant. Depending on the used channel various solutions are applicable. In a fieldbus environment a clock signal can be transmitted on the channel itself e.g. by using the Manchester-Code. Alternatively, a higher level protocol can be implemented to adjust the clocks accordingly. In this case one challenging issue is that clocks have a natural skew. This causes synchronized nodes to shift apart and gradually desynchronize again. Hence, the implementation has to observe and adjust the device clocks if necessary. In addition, another major problem is the (nondeterministic) delay emerging from the OS scheduler resulting into wrong timestamps at application layer. Consequently, we have to observe the node's master-clock as near as possible to the hardware. As a result, a device driver may be required to accurately monitor the node's time.

1.3 Objectives

The basic task is to find applicable solutions of time synchronization methods and provide integration hints for DCP. As reference, a Ethernet UDP

environment is assumed. The use-case is implemented using this DCP communication channel.

The first clock issue we face with DCP is the start criteria. It denotes the start of the simulation based on a timestamp. If the system fails to be synchronized it the system may not even start properly and in worst case may even invoke the error handling procedure. This should be avoided by implementing some time synchronization mechanism.

Based on the use-case scenario, the impact of clock drift should also be evaluated. First and foremost the drift itself may have an impact on a simulation. In this context it is important to find out how clock drift alters the simulation. It may be possible to completely forfeit an interference of clock drift if the simulation follows some basic guidelines. Alternatively it is also possible that the simulation results always differs if the nodes are not clock synchronized. Furthermore, a rough estimation of clock drift over time should be measured. It is important to know how much the clock changes over time.

Last but not least a suitable method to synchronization two or more nodes should be analyzed, suggested and implemented. Currently, there are various options available beginning with a special hardware design via a dedicated synchronization connection to software based approaches. Due to the flexibility and cost efficiency of software based implementations is favored. It goes without saying that this implementation has to be compatible with the current DCP draft. Therefore the implementation has to follow the DCP standard and its design philosophy.

2 Related Work

2.1 Definitions

2.1.1 Clock offset

This work references RFC-2330 [1] to define various clock issues. The offset is specified as the time difference of two nodes at a moment in time. This is often described as the clock model[2, 3].

$$\tau_j(t) := a_j t + b_j \tag{2.1}$$

Consequently, the offset is relative to an given reference clock which is denoted as τ_j. The offset can be minimized with various synchronization protocols like NTP. The offset itself can be harmful or a challenge in case the system is time dependent or time triggered.

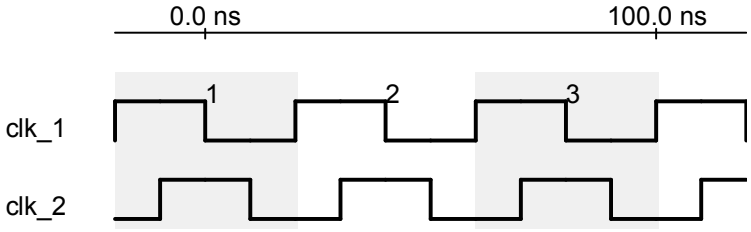

Figure 2.1: clk_2 is offset to clk_1 by half a tick

2.1.2 Clock skew and clock drift

Complying to RFC-2330 we define the clock skew as the first derivative of the clock offset over time. This is the difference of the clock frequency between nodes. The clock skew is a result of physical limitations and will cause the clock drift, which in turn is the second derivative of the offset over time. As the name already suggests, clock drift causes the clock to desynchronize gradually. Accumulating these minor errors results into increasing offset. The only real way to avoid this, would be the use of a single master clock for all nodes. Alternatively, the clock needs to be synchronized regularly.

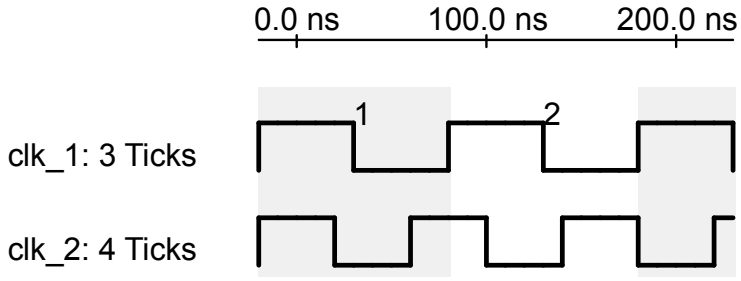

Figure 2.2: clk_2 oscillates faster than clk_1

2.1.3 Clock skew (circuit)

Unlike the clock skew defined in RFC-2330, this clock skew is based on the clock signal rather the device time. It describes the delay of the clock edge and the bound data signal. The clock skew is influenced by two factors: a random (Jitter) and a linear factor. The linear drift is caused from the frequency difference between the communicating devices. The jitter defines the random component of the clock skew. The jitter is a natural occurrence and cannot be eliminated fully.

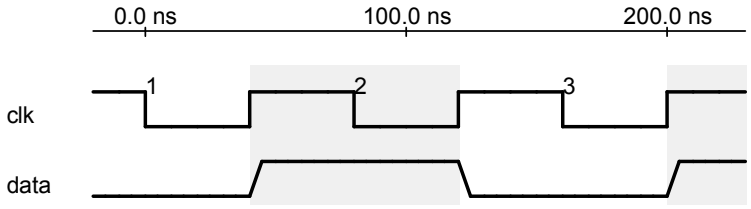

Figure 2.3: Signal responses slowly/delayed to the rising clock edge

2.1.4 Network latency/delay

The network latency is the required time to send and receive data to and from a node. This is also called the Round-trip-time (RTT). On the other hand the network delay measures only the travel-time of a signal in one direction – also called One-way-delay (OWD). Many algorithms estimate the RTT only knowing the network delay. In an optimal case, the OWD equals $\frac{RTT}{2}$ [4]. Besides physical restrictions, low latency can also be caused by the remote node for various reasons. For example the request is considered as low priority at the receiver.

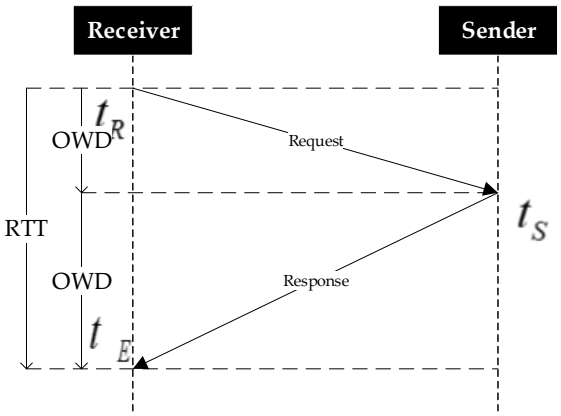

Figure 2.4: RTT and OVD of a Request/Response

2.1.5 Real-time

We call a system real-time capable, if it can guarantee that the response to a request happens at an exactly specified time. Furthermore, this definition is divided into “soft” and “hard” real-time. In a soft real-time environment minor violations of the deadline are permitted [5]. In a hard real-time environment deadlines must be adhered or cause a fatal error.

2.2 Open System Interconnection (OSI)

Since the specification of the Open System Interconnection model[6], it plays a major role in all fields of information and communications technology. It is designed as a standard “open” to all systems and allow integration as wide as possible. This is achieved by splitting each architecture into layers which are vertically stacked; see Figure 2.5. Each layer should add additional logic to the task and thus helps in splitting the overall problems into smaller pieces. Furthermore, it helps on organizing and structuring the design as each layer works independently. That is to say, a layer defines services and functionality by utilizing services of its lower layer and providing its own service to its upper layer. The boundaries of the layers are carefully selected based on historical practice or were the interactions across boundaries/layers is minimal. Next, the service of a layer usually combines similar functionality. Following these rules, the boundaries are often a logical result. As a consequence, changes to the software can be applied throughout the system. This is true as long it stays compatible with its neighboring layers. For example, we can run a Co-Simulation on an network based on Ethernet and UDP/IP on the other hand a UDP/IP frame may also be sent using the EtherCAT fieldbus. In both cases a UDP datagram in an IP network is used, but the actual physical connection differs. Furthermore, OSI also defines that each object at a given layer needs to be uniquely identifiable. To allow correspondence between the upper/lower layer a mapping function (or table) is used. As a consequence, a new connection at a at any given layer requires an established connection at the lower layer.

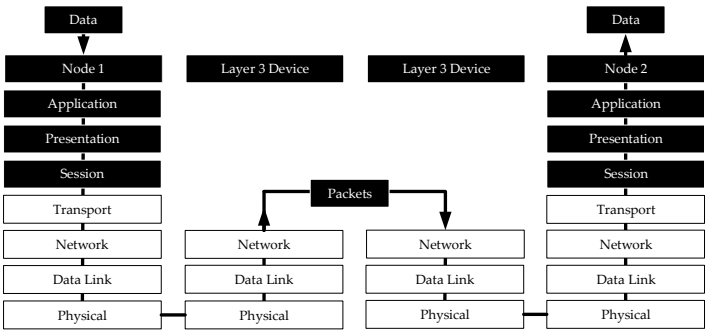

Figure 2.5: Connection of multiple systems with different layers through the OSI layer.

2.2.1 7 Layers of the OSI model

Nowadays the OSI model is well-known for its seven layers depicted in Figure 2.6. The layers are from lowest to highest: Physical, Data Link, Network, Transport, Session, Presentation and Application. The physical layer was chosen for obvious reasons to allow different types of connection. Therefore the next layer – called Data Link layer – should handle the different channels and provide a universal interface for layer 3. The Data Link layer handles error detection or recovery as well as collision handling. In addition, the Data Link layer provides the first addresses for nodes in the network and its data structure is usually called "Frames". Taking the modern internet as example, the layer 2 is usually Ethernet using the MAC addresses to identify nodes in the network. The used physical layer is typically 10GBASE-T.

The next step in the OSI layered model is to allow communication outside the nodes network. That is to say, the Network layer provides functionality to connect multiple networks. It provides a set of functions to support routing across systems. A system may also be only a intermediate node which is only forwarding data to the next system. The data structure in this layer are called "Packets" and again all nodes across the connected network must uniquely identifiable by an given address. In case of the modern internet, IPv4 and IPv6 are the protocols operating on layer 3. For the mapping between the two addresses, a table is used. In case of IPv6 it may be possible to derive the layer 2 address from the IPv6 address,

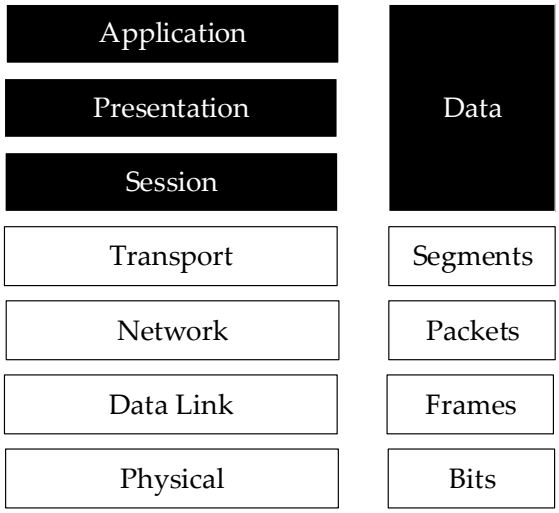

Figure 2.6: The seven layers of the OSI model and how data is represented at each layer.

depending on its configuration.

Finally, the Transport layer handles connection functionality which is only required on an end-to-end basis. Thus, it is the first layer not aware of the intermediate nodes. For the upper layers it provides simple functionality to send or receive data from another system. It relies on the lower layers to find a path or physical connection between the nodes. The well known protocols TCP and UDP are layer 4 implementations. For example TCP ensures that a connection to a destination is established and messages can be exchanged.

Beginning with layer 5 the Session layer is the first application aware layer. It handles the so called session administration service and session dialogue service. The former one is used to bind or unbind two presentation entities (layer 6). Whereas the other one handles data exchange between the upper layer and the transportation layer. This functionality is usually part of the operating system which provides APIs for communications (sockets).

Last but not least, the Presentation and Application layers are both part of the executable and thus can freely be tailored to the application. The Pre-

sentation layer describes the basic structure of the data so it is interpretable for the last layer independent of the environment. As such it has to align the correct byte-order or encrypt/decrypt data if applicable. Finally the Application layer is the highest layer of the architecture. It implements the actual logic and processes the shared information. That is to say the layers one to six are only supporting the Application layer.

2.3 Clock synchronization

2.3.1 NTP

The most basic time synchronization protocol is NTP, specified in RFC-9505 [7]. NTP is hierarchically organized into four stratums. The higher the stratum, the more accurate the time source is. Figure 2.7 shows a possible NTP topology.

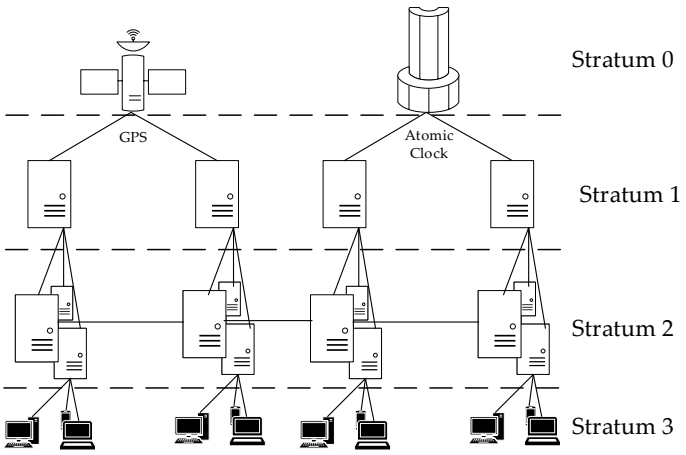

Figure 2.7: NTP is organized into four clock stratas – with decreasing accuracy

Each child polls the NTP server to synchronize its time to its parent time. The NTP server responses to this request with two timestamps τ_2 and τ_3. τ_2

is defined as the reception time of the slave's message. τ_3 on the other hand is the time when the master sends its response. The invoker is in possession of τ_1, the time the procedure was invoked and τ_4, the time the response from the master was received. Furthermore, the timestamps taken at the master were sent to the slave. Using these four values, the offset Θ can be calculated using equation 2.2, the RTT δ using equation 2.3.

$$\Theta = \frac{(\tau_2 - \tau_1) + (\tau_3 - \tau_4)}{2} \tag{2.2}$$

$$\delta = (\tau_4 - \tau_1) + (\tau_3 - \tau_2) \tag{2.3}$$

This procedure of NTP frame exchange is depicted in Figure 2.8. This

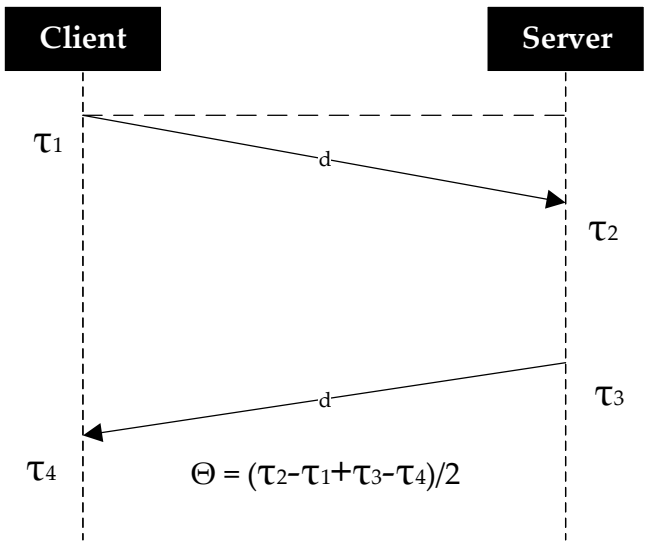

Figure 2.8: Offset and round-trip-time can be calculated using the given timestamps.

approach assumes symmetrical network delay[8]. In a small environment or within the same LAN this is usually applicable. However even in LANs a symmetrical delay cannot be guaranteed. Furthermore, with increasing

Field	Bytes	Description
Version	1	NTP Version
Strat	1	Stratum ID (0-15)
Poll	1	Poll Interval
Reference Timestamp	8	Seconds & Fraction
Originate Timestamp	8	Seconds & Fraction
Receive Timestamp	8	Seconds & Fraction
Transmit Timestamp	8	Seconds & Fraction

Table 2.1: Important NTP Fields

network connectivity this almost impossible to maintain. In the case symmetrical network delay cannot be achieved, the clock offset will slightly be biased.

Frame

A frame is 48 bytes long. NTP operates at the Application layer utilizing UDP for transportation. The important fields are shown in Table 2.1. Each timestamp can be stored in the frame. The timestamps are 8 Bytes long and store the passed seconds as well as the fraction of a second. Unlike Unix timestamps, these timestamps starts at 1. January 1900.

Summary

The accuracy of NTP greatly depends on various influential factors. The most impact is the accuracy of the first stratum (clock source), followed by the network delays and numbers of stratums. In modern systems with a high accurate clock source and a reliable fast network the accuracy of 1ms can be achieved [9]. Furthermore, the stratum depth needs to be taken into account. Each stratum increases overhead and errors. In addition, various solutions were discussed to improve the accuracy. Nowadays these solutions

are essential parts of a state of the art (NTPv4) implementation as described in RFC-5905. This involves clock updates in a phase-locked-loop (PLL) or frequency-locked-loop (FLL) design.

2.3.2 PTP

PTP is specified in IEEE-1588 [10] and it aims to provide a more accurate clock/time synchronization than NTP. Analysis proved that PTP is able to compensate differences up to 200ns[11]. Unlike NTP, PTP is not organized in stratums as each stratum decreases the accuracy. PTP is designed as a master-slave architecture. Consequently, the nodes try to synchronize to the reference clock directly. The reference clock is usually called "grandmaster", while each node is dubbed slave. Grandmaster clocks are usually specially designed network components. Either a high accurate clock is used or depend on GPS signal. Furthermore, their PHY units are designed to support PTP natively without introducing software overhead[12].

In addition, to omitting the stratum architecture, PTP also increases the accuracy by using a slightly more complex algorithm. The algorithm to calculate the clock difference comprises four timestamps and three to four network messages depending on the implementation. These messages are:

- Sync
- Follow Up
- Delay Request
- Delay Response

In contrast to NTP, the synchronization procedure is not triggered by the slaves but rather by the grandmaster. It sends its timestamp to the slave with the first (*Sync*) message and captures the sent time τ_1. Similar to the master, the slave captures τ_2 upon receiving the *Sync* message. In the best case, these timestamps are measured at the PHY unit, but at least in the network driver. The next two messages (*Delay Request* and *Delay Response*) are similar to NTP. These messages are used to retrieve the network delay. The slave sends a message to the master and captures τ_3 as the timestamp when the message was sent. Likewise, the master retrieves τ_4 upon reception of the slave's message. This timestamp is capsulated in the final message

(*Delay Response*) and sent to the slave. Like NTP, this algorithm assumes a symmetrical network delay. Using the four timestamps, the slave can calculate the offset Θ and the the more accurate RTT δ.

$$\delta = \frac{(\tau_4 - \tau_1) - (\tau_3 - \tau_2)}{2} \tag{2.4}$$

$$\begin{aligned} \Theta &= (\tau_2 - \tau_1) - \delta \\ &= \frac{(\tau_2 - \tau_1) + (\tau_3 - \tau_4)}{2} \end{aligned} \tag{2.5}$$

The synchronization procedure is depicted in Figure 2.9.

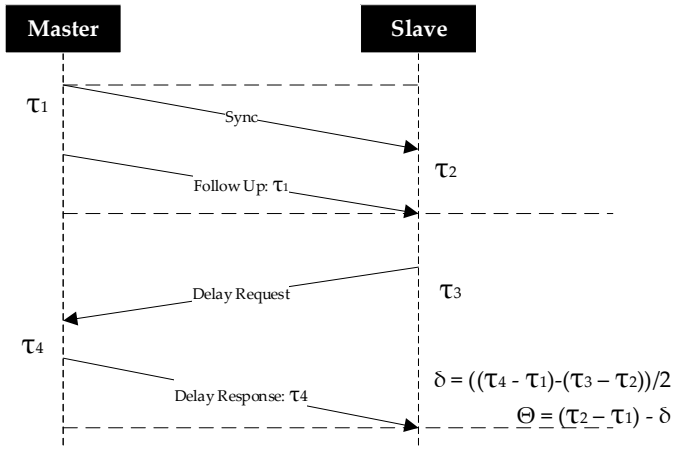

Figure 2.9: Synchronization process of PTP

Frame

A PTP frame is 44 bytes long. All four PTP message relay on the same structure. PTP can be operated at the Network Layer or encapsencapsulated uled inside a UDP package. The first one is possible due to the fact that PTP is designed to run only inside the local network. This enables (special)

Field	Bytes	Description
Length	2	Bytes of the message
Flags	2	Bitmask of various options
Correction	8	
Clock Identity	8	Identity of the master clock
Sequence	2	Running number
Control	1	Message type
Origin Timestamp (s)	4	Unix time in seconds
Origin Timestamp (ns)	6	Nanoseconds offset

Table 2.2: Important PTP Fields

hardware to directly inject the appropriate timestamp into the frame. Table-2.2 shows important fields of the PTP message. The timestamp itself is 10 bytes long. Like NTP, the timestamp has a seconds and a fraction field, with 6 bytes reserved for the seconds and the remaining 4 bytes for the fraction. PTP uses the UNIX time to encode its timestamp. Consequently, it counts the seconds starting at 01–01–1970. The fraction field is populated as an offset to the full second with nanosecond precision. As such the fraction field can vary the timestamp up to 2 seconds in both directions.

Comparison of NTP and PTP

Both implementations try to achieve the same goal, which is clock synchronization. However they vastly differ in their priorities. While NTP is designed with scalability in mind, PTP focuses on accuracy. Thus PTP lacks any scalability related features. It can only synchronize against a single-grandmaster clock within the same network. In this context NTP is vastly more flexible. With the stratum design it is hierarchically organized and can support more clients. Only a handful NTP servers in Stratum 1 synchronize against the high accurate reference clock. It goes without saying that each additional stratum decreases the precision. In this respect, each PTP grand-

master requires a reference clock. From an economic point of view, this has to be considered. NTP synchronization is by far cheaper than PTP. The next main difference between these two protocols is their integration within the OSI layers. While both protocol can be encapsulated in UDP packages, PTP can be operated at the Network-Layer. Thus (software-) overhead and potential delays can be avoided. This also allows PTP to operate more closely at the hardware. Some hardware – especially grandmaster components – even support PTP natively.

2.4 Ethernet

Ethernet – specified in the IEEE-802 family [13] – is well adopted across devices. Thus, its compatibility and simplicity offers a great alternative to fieldbus systems. Hence, its usage is a valid use-case in an automation or simulation environment. Recent trends show that ethernet is on the rise[14, 15]. However, unlike many fieldbus systems, ethernet was not designed with real-time communication in mind. As a result, the ethernet-based protocols like IP and TCP or UDP are not real-time capable as well. The main disadvantageous property of ethernet is its non-deterministic behavior caused by its collision handling system CSMA-CD and interface design.

2.4.1 UDP

UDP – specified as RFC-768[16] – is the preferred protocol to use in a real-time ethernet network[17]. In contrast to Transmission Control Protocol (TCP) it sends only independent packets. As such it does not suffer from indeterminable delay. However, this comes at the cost of missing various properties like ordering and linking related data. Furthermore, UDP cannot guarantee the delivery of the packet which conflicts with the definition of deterministic[18]. Nevertheless, it is the only protocol which can comply to a strict deadline. By design UDP is rather simple. Its 16 bit header is shown in Table 2.3. Considering the header data, the payload of UDP is limited between one byte and 65.507 bytes. The upper size is determined by the underlying IP protocol, which limits the payload of a IP packet to

Field	source_port	destination_port	length	CRC
Bytes	2	2	2	2

Table 2.3: UDP Header with Size in Bytes

64k. A UDP header is concluded with an optional CRC field of two bytes. Both PTP and NTP are used on top of UDP. Although PTP can also be implemented at MAC layer using its own frame. Nevertheless, both of the solutions are practicable and do not suffer any obvious disadvantages nor advantages.

2.4.2 Nondeterministic behavior of Ethernet

As any physical channel is a shared resource and can only be used by one participant for writing at the same time, a mechanism to avoid collisions is required. For ethernet, Carrier Sense Multiple Access – Collision Detection (CSMA-CD) is used. As the name already suggests, CSMA-CD can be broken down into two operation modes. To begin with, the Carrier-Sense Multiple-Access part monitors the channel for an active transmission. As long as the channel is occupied, CSMA will not allow sending data from its own interface. Once CSMA detects no activity on the channel a transmission is allowed. Next, the Collision-Detections plays a major role in terms of predictability. With a growing number of participants, the probability of devices trying to send at the same time increases. Once such a case occurs, interfaces react by jamming the channel for an random time. In terms of deterministic, especially the CD part is problematic. The issue lies with the jamming time, which – by design – cannot be estimated. Next the system does not consider the urgency of messages after the jamming ends. Hence, a low priority message may be transmitted instead of a important one. Summing up, Ethernet is designed to be non deterministic in order to handle shared access to the network. It goes without saying that this violates with real-time constraints. This also has a negative impact on clock synchronization methods relaying on RTT and network delays. This is true for all software based solutions.

Besides a random jamming due to collisions, a message may also be low prioritized and may be delayed indefinably. That is to say, it is possible that a message may starve and miss its deadline. This behavior is strongly influenced by the used network scheduler of the used network devices/-drivers and the workload of the used interface. Various propositions exists to handle these issues on different layers of the OSI model. These solutions are basically categorized in two basic groups. On one hand the MAC (2nd OSI layer) can be modified to achieve a more deterministic behavior. This *can* be achieved in software, by adapting the operating system or the device driver. However for strict tolerance applications a modification of the used hardware is needed. On the other hand higher-level control implementations can reduce the probability of a collision happening significantly. That is to say, all nodes agree on a protocol which controls the nodes I/O operations to avoid collisions. Practical implementations are token-ring or time multiplexing [19, 20]. For these to work probably, accurate clock synchronization is essential.

3 Method

3.1 Overview of Co-Simulation

In modern day vehicle development is getting more difficult each day. With increasing requirements in the many different fields the complexity of a state of the art vehicle has skyrocketed. Nowadays it is required to reduce the consumption and satisfy tight CO_2/NO_x limits, while providing more comfort and security to the driver and passengers. Furthermore, a clear trend towards automated driving can be observed. Many parts in vehicle development need to comply to strict regulations.

This caused, that even the development of a single system is a challenging task itself. As a result many components are developed on its own and provide a set of interfaces to work with each other. The system needs to be thoroughly tested without relaying on other components to be present. This however opposes to the fact, that each system in a car needs to interact with each other to work properly. For example, a Lane Keep Assist System (LKAS) relays on multiple sensors data and indirectly controls the car. Given certain circumstances this in turn may affect an Electronic Stability Control (ESP) system. Thus almost all components are developed in a simulation environment. It is essential for modern verification and testing. With increasing possibilities and knowledge in ICT a trend to distribute the simulation has begun. Such an environment is called Co-simulation. The term describes combining various tools and models into a single environment. In a co-simulation environment a distinction between Software and Hardware models is not required. In other words, it also allows interconnecting software and hardware likewise. Thus, it is possible to connect a system in development to an actual prepared car[21, 22, 22]. One of the pioneers in this field is the Functional Mock-up Interface (FMI)[23]. It describes a

interface over which a set of models called Functional Mock-up Unit (FMU) communicate.

3.2 DCP

DCP provides standard definition for models and interfaces to run a distributed co-simulation. It follows the design principles of FMI. In addition to FMI, DCP is designed with real-time support in mind. This is especially important for Hardware in the Loop (HiL) simulations. Like many other protocols, classification of DCP into the OSI model[6] is only vaguely possible. To begin with DCP runs at application layer and has access to the data. Furthermore, it defines their own messages – named PDU – whose properties span across multiple OSI layers. With the definition of a custom message type, the criteria for presentation layer is met. Next, the DCP and PDU design allows to setup multiple co-simulation scenarios and thus classify for a application at the session layer. Additionally, DCP also defines transportation layer properties like sequence numbering although a separate transportation protocol like UDP or TCP is used[24, 25]. The most important key features are described in the following subsections[26].

3.2.1 Multichannel support

DCP is designed to support multiple communication channels and can easily integrate new ones. For the moment the most prominent ones are CAN and Ethernet via UDP/IP. Both protocols play a major role in modern industrial applications. In automotive engineering CAN is still de facto standard. In simulation CAN and RJ-45 belong to the most common interfaces. While Ethernet is not designed with the safety features of CAN in mind, it still offers a fast and reliable connection suited for most simulations during the development phase. Research showed that Ethernet is on the rise across many industrial fields[14]. Its support also opens up the possibility to incorporate consumer graded hardware. This poses an enormous economy advantage by decreasing computer costs and speed up development time. In the long run proper Ethernet support may also enable

true distributed co-simulation by utilizing TCP. This would open many possibilities like running a simulation distributed over multiple subsidiaries or even across countries. It goes without saying that TCP support faces multiple challenges.In addition to wired communication, a wireless one is a viable option as well. For industrial application, Bluetooth has shown some promise for possible adoption. Thus, in the current DCP draft a Bluetooth driver is specified as well.

3.2.2 Master-slave architecture

A DCP environment is based on the master-slave architecture. Each model inside a simulation is addressed by its unique identifier. While the ID zero is reserved for the DCP master, any other value is assigned to the DCP slaves. The DCP master is responsible for this task and must guarantee the uniqueness of IDs. The master slave communication is used the exchange

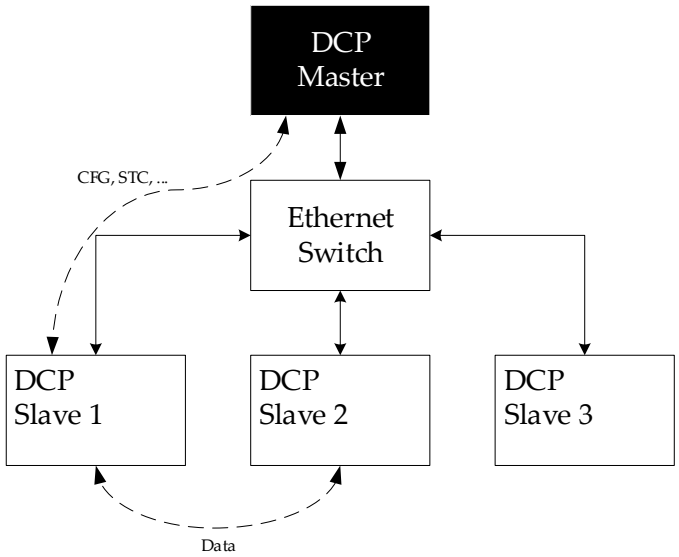

Figure 3.1: DCP on top of ethernet star topology. The dotted lines indicate the PDU communications.

configuration data and commands. Thus, the master is responsible to configure the slaves correctly, start and stop the simulation and handle errors if any occurs. With a heartbeat mechanism the master is also able to monitor the status of any slaves. If a slave diverges from the expected state the master will raise an error and thus stopping the simulation. However, there is also a slave to slave communication. It is used for simulation data, mainly for performance reasons. The data cannot be routed over the master as it would be too much of a bottleneck. The possibilities of message exchange within a DCP environment is shown in Figure 3.1. Note that the actual connection between the nodes depends on the used communication channel. In the depicted figure, a Ethernet connection is assumed. Thus, the messages are actually routed via a switch to either the master or a slave. In an possible CAN environment, all nodes would share the same bus. Obviously the same channel would also be shared if a wireless connection is used.

3.2.3 Protocol Data Unit (PDU)

In DCP exchanges messages are called PDU. All PDUs are fixed in size either by the specification or in some rare cases configured before the simulation starts. Once the simulation starts the PDU sizes need to be fixed and known for all nodes. Thus, the implementation can always allocate the exact amount of memory. The PDUs are categorized into three categories and six groups. Each PDU has its own ID. Depending on its group the ID is within a set range. The range of each PDU group is listed in Table 3.1. In order to acknowledge or reject a PDU, each message provides a sequence id. The subsequent ACK/NACK sets its response sequence id to the given sequence id.

Starting from the bottom, the Data (`DAT`) PDU group is used to exchange simulation data between the DCP slaves. Simulation data is stored inside DCP variables. Variables are addressable by using a variable specific identifier. The ID is usually based upon a counting sequence. Variables can be distinguished between input and output variables. Each variable has a datatype assigned to it and thus vary in size. The `DAT` PDUs operates similar to a streaming buffer. They are exchanged between the slaves after each simulation step. Depending on the variable type, data is either send or received.

PDU Group	type_id	pdu_seq_id	resp_seq_id	sender	receiver	[additional]
CFG_*	0x20-0x2F	y			y	f/c
STC_*	0x01-0x1F	y			y	f
INF_*	0x80-0x8F	y			y	f
RSP_*	0xB0-0xDF		y	y		f
NTF_*	0xE0-0xEF			y		f/c
DAT_*	0xF0-0xFF	y				f/c

Table 3.1: PDU groups with data fields. Legend: *y*: Field is set. *f*: Field depends on a specific PDU. *c*: Field is configured by the DCP master.

Its fields are listed in the basic overview in Table 3.1. The fields required node addressing comprises `ID`, `pdu_seq_id` and a `param_id` or `data_id`. It is possible to store multiple variables inside a single `DAT` PDU. Thus, the actual size of payload is not fixed in the specification but is configured by the master. Obviously it is not possible to combine input and output variables in the same PDU. The variables are access by an offset to the payload buffer.

Next, DCP also provides Notification (`NTF`) PDUs. Currently, two of them are defined in this group. Their basic structure is shown Figure 3.1. The IDs of `NTF` PDUs are within the range of oxEo and oxEF. `NTF_state_change` is as simple as possible. Its payload only stores the State ID. More information on the DCP state-machine is described in the Sub-Section 3.2.4. On the other hand the `NTF_log` is used to log information and data. Due to possible variable values this PDU is variable sized. In DCP each log message is predefined. They are configured similar to formatted printing. A format string with embedded variables is defined. In `NTF_log`, the variables must be ordered accordingly to the format string.

Any other PDUs groups are listed in the *Control* category. It composes of Response (`RSP`), Information (`INF`), State-Change (`STC`) and Configuration (`CFG`) messages. These are used for communication between DCP master

and DCP slaves. A RSP obviously sends a response back to the DCP master. In order to link the request with an response, the response sequence id (rsp_seq_id) field is set according to the requesting sequence id (pdu_seq_id). As a response is tied to its request, the RSP PDUs are explained in the following sections together with their requests. If no data was requested, but a command was send instead, the RSP PDUs are also used to acknowledge or reject the command. This is done by sending a RSP_ack or RSP_nack. Depending on the configuration and implementation an error may be raised consecutively. "Not Acknowledge" PDUs also contain an error_code field, providing additional information why the message was rejected. DCP defines a list of error codes, in which a error code is assigned to an Mnemonic and a description.

INF messages are used to request information from the DCP slaves. Naturally, the DCP slave answers the request with a RSP PDU. In the current draft, three properties – encoded in three PDUs are supported. To begin with the DCP master can retrieve the current state of a DCP slave using the INF_state PDU. The state machine itself is described in more detail in the next sub section. Next, the master can also request the last set error of a slave. As already described, a list of error codes is defined by DCP. In case of an error its ID is reported back to the master. Finally, the INF_log can be used to retrieve something else? The slaves response (RSP) PDUs are: RSP_state_ack, RSP_error_ack and RSP_log_ack.

In addition to requesting information, the master also controls the slaves and the simulation. On one hand, the master configures the slaves utilizing the CFG PDUs, on the other hand it also controls the state machine by sending STC PDUs. For each state change, an individual STC PDU is defined. These PDUs effectively commands a slave to switch to the given state. Some State-Change-PDUs carry additional information like *start time* (STC_run) or *step count* (STC_do_step) in case of non-real-time simulation. A slave acknowledges a state change with RSP_state_ack. It can also reject the change with RSP_nack. This could occur in various situations. For example,an error was raised or the slave could not complete its task of the previously state. If the master violates the state-machine, the slave must reject the change as well.

Finally the CFG group is used to configure a slave. A slave can vastly config-

ured, starting from time resolution (CFG_set_time_res), to custom logging formats (CFG_set_logging). Obviously to achieve great flexibility, it is also essential that the system can define its input and output signals at run-time.

3.2.4 Operation and state machine

As already mentioned, the life-cycle of a DCP environment is defined by a state-machine. Although it is controlled by the DCP master, each DCP slave has to verify the state validity. Any instance in the simulation must raise an error, in case the state-machine is violated. The state-machine is depicted in Figure 3.2. A slave changes state either by receiving any of the STC PDUs, or on its own e.g. once it completes its task for the state. In this case, the slave notifies the master by sending a NTF_state_change PDU.

The entry point of the state machine is the ALIVE state. Once a DCP slave is powered up, it will automatically start in this state. During ALIVE, the slave simply waits till it is registered by receiving a STC_register from a DCP master. A slave can enter the ALIVE state again only if it is deregistered by the same DCP master (STC_deregister). Once deregistered the slave forgets any configuration and is reset to its initial state.

Once a DCP master has taken ownership of a DCP slave, it is in the Configuration state and can be configured using the CFG PDUs. The received commands must be applied before switching to the CONFIGURED state. Any configuration command must be accepted using RSP_ack or rejected using RSP_nack. The master shall decide how to proceeded if a configuration is rejected. The master could try again using a different setup or abort the process. In order to organize the configuration process, a two "state-pairs" are between CONFIGURATION and CONFIGURED.

To begin with, the first configuration states are PREPARING and PREPARED. During PREPARING the slave shall setup its input/output interfaces. The applicable interface is decided by the master using CFG_source_network_information. Once the interfaces are prepared, the slave will signal a state change to PREPARED using SIG_prepared. Next, the master will switch to the second pair named CONFIGURING and CONFIGURED. This is achieved by

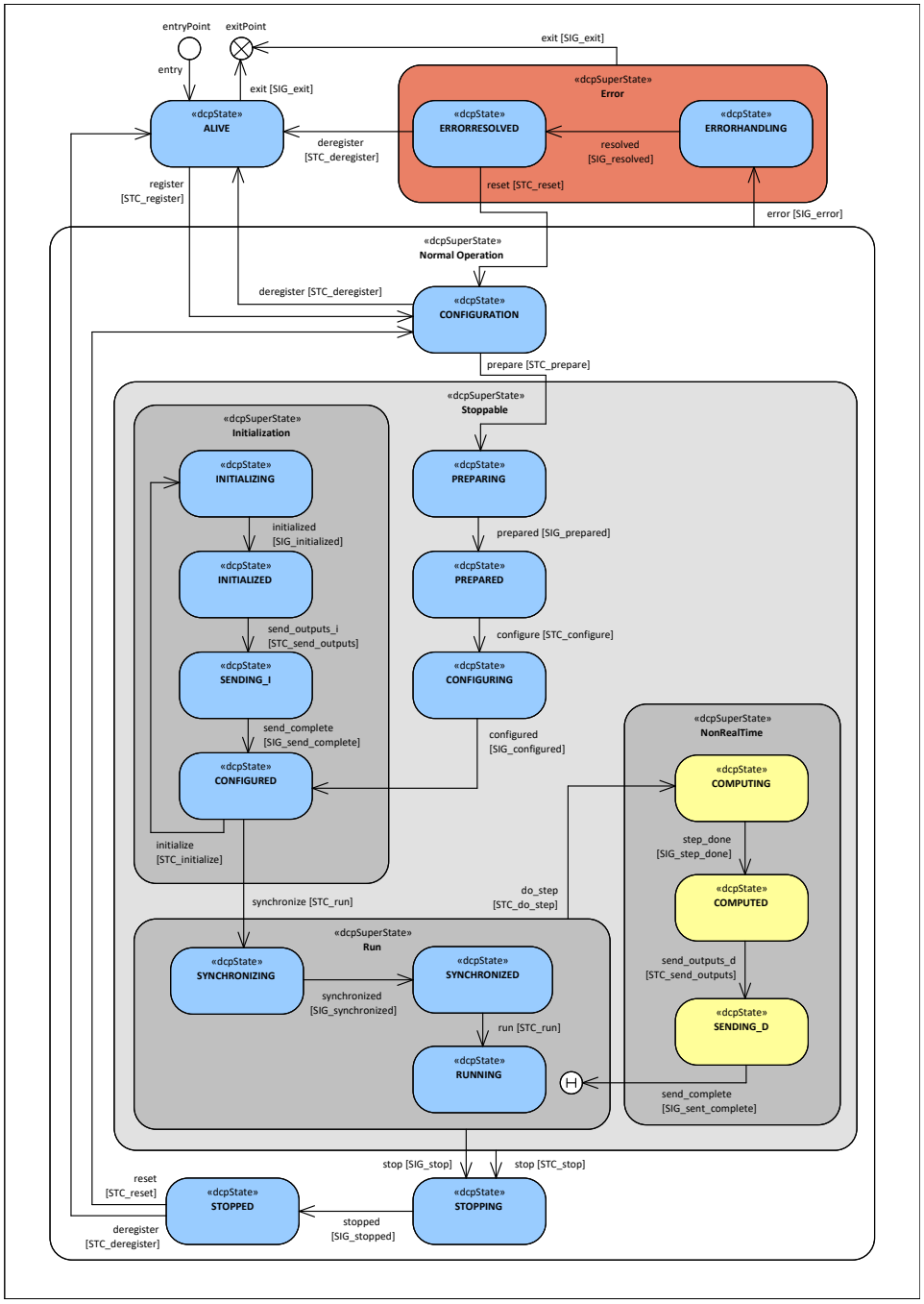

Figure 3.2: DCP State machine with superstates Error, Stoppable and NonRealTime[26].

sending the STC_configure PDU. These states are mainly designed to establish a connection if a connection based interface is used. In addition, a start condition is independent on simulation values is distributed. Once the slave is aware of the starting condition and if required has established the connections, it signals the CONFIGURED state. It is expected that the slave is now aware of how to start the simulation. At this time synchronization is expected.

Upon completing the configuration procedure, the slave must initialize its simulation values. That is to say, the input variables are set to initial values and the first output variables are computed. These steps are executed during the INITIALIZING state and must be concluded before INITIALIZED. The slave signals once INITIALIZED shall be entered. It is expected that the input is constant during this state and a valid output can be calculated. The initializing phase is completed with SENDING_I state, during which the computed outputs are exchanged. Next the slave will switch to CONFIGURED state again and may rerun the initializing phase indefinitely until it starts the simulation by sending STC_run.

The real-time simulation starts once the slave time equals the time stored in the STC_run PDU. However, as some models may require some windup time, the first few simulation steps are executed during the Synchronizing and Synchronized states. The actual RUN state is triggered by sending a second STC_run PDU once every slave is synchronized and in the corresponding state. Note that this SYNCHRONIZATION has nothing to do with time synchronization, but rather with model synchronization.

Alternatively, the simulation can also be executed in non-real-time mode. In this case a small state-machine consisting of the states COMPUTING, COMPUTED and SENDING_D is executed. It goes without saying that COMPUTING is used to compute the task and the slave switches to COMPUTED once it has completed. The simulation data is exchanged in the SENDING_D state. These states are iterated til a exit condition is met.

During RUNNING (super-) state, the simulation is halted by sending a STC_stop message. The slave will enter the STOPPING state and wind down the simulation. Upon finishing the cleanup procedure the slave switches to STOPPED. Here the slave waits for further commands. It may return to CONFIGURATION state or the master may deregister it.

If an error occurs during any state, the master or the slave can enter ERRORHANDLING. The slave will stay in this state until the error is resolved or the slave disconnects. Once the error is resolved, the master can deregister the slave or try to reconfigure it again. In this case it enters the CONFIGURATION state again.

3.2.5 Heartbeat mechanism

Another import aspect of DCP is the ability to monitor the availability of the complete system. While it is only natural, that the DCP master keeps track of the DCP slaves condition, in DCP the DCP slaves also monitor if the DCP master is still functional. This is implementation by a heartbeat mechanism. Based on the configuration, the master requests the slave's state utilizing the INF_state PDU in a periodic manner. The slave must respond with an RSP_state PDU. If either the master or the slave did not receive the expected message within a certain time spawn, the node assumes that the communication partner has died and will thus enter the ERRORHANDLING state. It goes without saying that the time spawn is slightly larger than the expected period in order to compensate slight delays. The heartbeat mechanism and its time values is depicted in Figure 3.3.

3.3 Clock synchronization for DCP

Various time synchronization mechanism were presented in the related work chapter, see Section 2.3. However, mainly for flexibility and scalability the synchronization method should be software based. This effectively rules out every implementation except the NTP and PTP approach. Yet, these two protocols vastly differ in their philosophy. While PTP is designed with precision and high accuracy in mind, NTP tries to satisfy a massive amount of clients. The key difference between NTP and PTP is explained in Section 2.3 as well. In terms of DCP and its goal to support real-time simulation only PTP is a viable option.

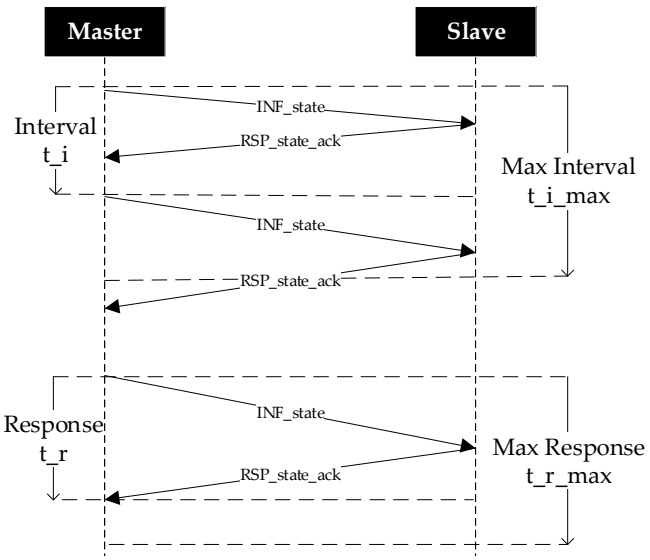

Figure 3.3: DCP Heartbeat functionality

3.3.1 PTP over DCP

PTP was already explained in the Section 2.3 of the related work chapter. The basic procedure of a PTP synchronization process is depicted in Figure 3.4. For PTP, four messages and timestamps are required to calculate the clock difference between two nodes. As these timestamps are not collected at the same node, some of them need to be exchanged. Obviously, as clocks are completely accurate this process needs to be performed in a set interval.

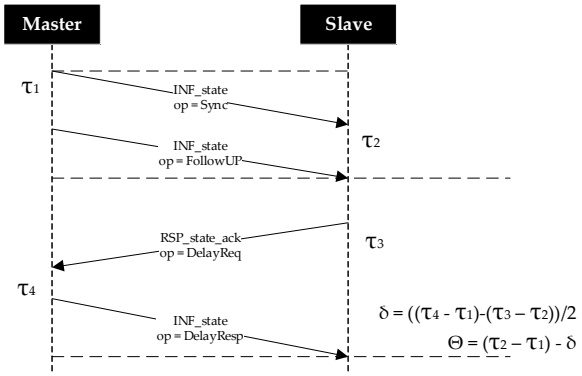

Figure 3.4: PTP Messages

It goes without saying that the simulation can only be started if all nodes are synced. If the DCP slaves are not synched correctly, they may start with their simulation steps too early or too late. Consequently, we have to trigger the synchronization before we are in the "run" superstate and guarantee that the simulation start time is equal to all nodes. Once the simulation started, we need to cope with the drift of the clock. As already mentioned, this is handled by periodically performing the synchronization steps. Thus, DCP also has to run its synchronization procedure in a defined interval. To support a wide range of possibilities, a numerator and denominator similar to the heartbeat interval is added to the slave configuration. The DCP master then shall trigger synchronization every $\frac{numerator}{denominator}$ seconds. While there is no right value for the interval and its range is quite large, it should be within a safe margin. Due to the fact, that at least four additional PDUs

needs to be sent, the underlying bus may get congested if the interval is set to short. On the other hand a very large interval may not be sufficient to counter the drift. In PTPd – a implementation for Unix OS – the smallest (and default) interval is set to one second. With DCP it would be possible to lower this value, but it is not suggested. The upper value greatly depends on the accuracy of the node's clock. The higher the accuracy is, the less offset correction is needed. In doubt a interval around one or two seconds does not harm the system. Detailed results on how clock differences inclines over time are shown in the result section of Chapter 5.

3.3.2 New PDU types

For time synchronization the most important fields are timestamps. The current specification already provides a 64 bit `time` field, though it is designed to store a UNIX timestamp. However, for the synchronization to work a more accurate timestamp is required. The proposed value would also be a 64bit field named `time_us`, capable of storing time in microseconds resolution. Given the Equation 3.1 the possible time spawn is more than sufficient.

$$2^{64}us \approx 584942years \tag{3.1}$$

To synchronize nodes, four new PDUs are required. They are depicted in Table 3.2. The basic procedure follows the PTP steps explained above on a regular interval besides the existing heartbeat monitoring. The heartbeat mechanism is not modified and not removed. It is still valid, as the synchronization may not be used as a heartbeat signal due to its different interval.

As three of the four newly defined PDUs share the same payload, they can easily be merged into a single PDU. The only exception PDU is the response PDU as its sender/receiver field is swapped. It goes without saying, that we still need to distinguish between the current PTP mode. The easiest way to achieve this, is by adding a field `sync_op` storing the current operation. The field is based on the smallest available type, a `uint8`. The possible values for this field are listed in Table 3.3.

	type_id	pdu_seq_id	resp_seq_id	sender	receiver	time_us
INF_sync	0x83	y			y	
INF_follow	0x84	y			y	y
INF_delay	0x85	y			y	
RSP_delay	0xB5		y	y		y

Table 3.2: PTP messages encapsulated in DCP PDUs

Operation	value
Notify	0x00
Sync	0x01
Follow-Up	0x02
Delay-Req	0x04
Delay-Rsp	0x08

Table 3.3: The field sync_op with its possible values to distinguish between PTP operations

	type_id	pdu_seq_id	resp_seq_id	sender	receiver	sync_op	time_us
INF_sync	0x83	y			y	y	y
RSP_sync	0xB5		y	y		y	y

Table 3.4: Suggested PDUs `INF_sync` and `RSP_sync` encapsulating PTP

Summing up, the newly created PDUs `INF_sync` and `RSP_sync` stores the mode in the `sync_op` field and the time in microseconds in the `time_ms` field. The new types and PDUs are depicted in the Table 3.4.

Unlike ordinary PTP, the `sync_op` field also reserves space for *Notify*. This can be used to incorporate the heartbeat mechanism. A value in the first bit indicates the synchronization mechanism. It is expected that the system handles it according to PTP. On the other hand, an empty value for the `sync_op` field simply implements an echo request. The slave shall response with the current time. In the interval of the heartbeat mechanism, a `INF_sync` with an empty `sync_op` is sent. Failing to receive such PDU, the slave can detect an faulty master and enter the `ERRORHANDLING` state. On the other hand, if the slave fails to response in time, the master can raise the error. Thus, a fully operational heartbeat mechanism can be included. In addition to this, each synchronization interval two `INF_sync` PDUs with an set value of *Sync* and *Follow-Up* for `sync_op` are sent. In this case, the slave must perform the PTP implementation and correct the clock difference.

Alternatively to adding new PDUs, an existing PDU can be expanded upon to support time synchronization. As DCP already implements one periodic action, namely the heartbeat mechanism, it is natural that the PDU used for this process would be adapted. In the heartbeat mechanism, the `INF_state` and `RSP_sync` PDUs are used. In order to support time synchronization, these messages would need to carry an additional timestamp and a flag showing which operation currently is performed. Thus, the resulting `INF_state` PDU would also contain the previously described fiels from the *INF_sync* PDU. The changes are the following: Add a new fields `sync_op`

	type_id	pdu_seq_id	resp_seq_id	sender	receiver	state_id	sync_op	time_us
INF_state	0x80	y			y	y	y	y
RSP_state	0xB2		y	y		y	y	y

Table 3.5: Modified INF_sync PDU

and time_us. The possible values for sync_op are listed in Table 3.3. In order not to break the heartbeat mechanism the sync_op field needs a "NO-OP" mode allowing the slave to simply report its current state. This would be the "empty" *Sync* value. On the other hand, if sync_op is set to any value, the PTP sync operation should be executed. As PTP also includes requesting a timestamp from the master during the synchronization operation, the RSP_state PDU needs to be updated as well.

4 Implementation

4.1 Implementation environment

This chapter describes necessary changes to an DCP implementation in order to add time/clock synchronization support. It follows up to the suggested modifications of Chapter 3. To begin with, some clarifications. Firstly the test application is based on the reference DCP implementation on GitHub[1]. It uses standalone ASIO[2] for its network interface. Thus, the used programming language is C++. All used libraries are "header only". While that implies that no compiled library is used, it also results into long compile times. Nevertheless, it also ensures great flexibility and platform independent support. It can basically run on every platform with an modern C++ compiler. Next, it is worth to point out, that the changes are only implemented to work with an Linux based operating system. Mainly due to the fact that Windows provides no PTP interface or any other native workaround. Furthermore, given the nature of the Windows operating system and its lack of real-time support, such a highly precise clock would not have the desired effect anyways[27]. Although Linux is also no RTOS, it can achieve notable results especially when applying the real-time patch. In addition, in embedded systems Linux is the unrivaled operating system. Finally, in this implementation the DCP master is assumed as the PTP grand master clock. In the current draft a pure DCP slave which can act as a pure grand master clock is not realizable.

[1] https://github.com/modelica/DCPLib
[2] https://think-async.com/Asio/

4.2 Additions to DCP

In the previous chapter, three possible PDU modifications were presented. The last solution would alter the `INF_state` PDU and expand it with timestamps and other fields. However these fields have nothing to do with reporting the state of a DCP slave as the name would suggest. The resulting PDU would be some kind of monolithic message. Thus, it is deemed unfitted for the modifications. As a consequence two or more PDUs need to be added. Due to the fact, that DCP only reserves one byte as PDU type, the maximum amount of possible PDUs is rather limited to 256. Thus, this implementations favors the second approach over the first one and introduces a two new PDUs.

4.2.1 Additional synchronization PDUs

These PDUs are named `INF_sync` and `RSP_sync`. Their IDs follow the basic DCP scheme and is 0x83 for `INF_sync` and 0xB5 for `RSP_sync`. Given only two PDUs for four different PTP messages, a special field is used to distinguish between them. This field is called `sync_op` and is based on an `uint8`. It's reserved values are depicted in Table 3.3 from the previous chapter. Note that PTP only requires four types to distinguish between each operation. However, in order to possibly integrate the heartbeat mechanism, the fifth value *Notify* was added. In terms of resources it has no impact but may be useful in the future. The second field of the synchronization PDU payload is essential for a working clock synchronization. It stores a adequate accurate timestamp. In terms of accuracy, the existing DCP field to store date/-times is based on UNIX timestamps and thus has a resolution of seconds. This is not sufficient. For satisfactory clock synchronization, the timestamp need to be at least in microseconds or even nanoseconds. POSIX defines two structs (`timeval` and `timespec`) capable of holding times in such high resolution[28]. Nevertheless, analysis of Linux real-time capabilities showed that context switches delays around 50us[29, 30]. Consequently, microseconds is deemed suitable for the DCP use. Although the POSIX definition implements `timeval` by combining a counter for seconds and milliseconds each, in DCP a single 64 bit sized integer is used. Given Equation 3.1 the

timestamp will not run out of values any time soon. The following Equation 4.1 and 4.2 can be used to convert between the POSIX `timespec` and the DCP `time_us`.

$$time\_us = tv\_sec \cdot 10^6 + tv\_nsec \cdot 10^{-3} \tag{4.1}$$

$$timespec = \begin{cases} tv\_sec = time\_us \cdot 10^{-6} \\ tv\_nsec = time\_us \mod 10^6 \end{cases} \tag{4.2}$$

The resulting PDU (Figure 3.4) is also shown in Section 3.3.2.

4.2.2 Important DCP states for synchronization

It goes without saying that the clocks needs to be synchronized once the simulation starts. However, DCP is not as simple as start, synchronize and run the simulation. Between starting the DCP slave and beginning of the simulation various states are in between. To sum up roughly, the following tasks need to be performed until the slave is ready. Firstly the slaves must setup its network interfaces (`PREPARING`) and configure itself according to the received *Configuration* PDUs from the master (`CONFIGURING`). Next, all simulation data and variables (input/output) are initialized (`INITIALIZING`). Finally, the simulation is winded up in the `RUN` superstate. The `RUN` superstate is entered via the `SYNCHRONIZING` state using the `STC_run` PDU. Note that this does not synchronize the clock, but rather the simulation. Once the simulation is synchronized, the slave signals the `SYNCHRONIZED` state, which in turn will eventually trigger the second `STC_run` PDU causing the simulation to start. Now the slaves are effectively in the `RUNNING` state. As the `STC_run` PDU carries a timestamp when the simulation should begin, it is important that the difference is already known at that point. Consequently, the initial synchronization muss happen before entering the `RUN` superstate. Furthermore, once the clocks are synchronized the clock difference must be kept at a minimum. The only fitting place for clock synchronization is the `CONFIGURING` state. Thus, initial clock correction is triggered together with the `STC_configure` PDU. Once this process has completed, the regular interval for synchronization is started. Thus, the `INF_sync` PDU is valid for all states following the `CONFIGURING` state.

4.2.3 Synchronization trigger interval

As already described, clock differences occur not only during startup, but also after time due to inaccuracies of clocks. This can be compensated by regularly updating one clock. In this solution the DCP master node is assigned as the grand-master role. Thus, all slaves have to update their clocks to match the simulation time of the DCP master. Thus, the DCP master has to invoke the synchronization procedure in a set interval. Current the DCP master already performs a similar operation by sending (and watching) the heartbeat PDUs `INF_state`. The interval for the heartbeat is configured by setting a *numerator* and a *denominator*. The fraction describes the interval in seconds. Its default value is one second. The expected interval time for the synchronization method is about the same. However, as each simulation environment differs and has various demands on the accuracy or stability a independent interval is used. Similar to the heartbeat mechanism the interval for clock synchronization also is defined as a fraction.

4.2.4 Integration of PTP in DCP master

In PTP, the PTP master triggers the synchronization mechanism. Thus, the DCP master must perform these tasks. It is already stated, that correction is triggered during the `CONFIGURING` state. Starting from this point, after each passed interval as well. In order to begin the sequence, the DCP master must send an "empty" `INF_sync` PDU with `sync_op` set to *Sync*, followed by a second `INF_sync` PDU. This time the `sync_op` is set to *Follow-Up*. The timestamp `time_ns` must be set tot the "socket time" of the first message. That is to say, by sending the first `INF_sync` a timestamp is captured and sent to the DCP slave in the so called *Follow-Up* message. Given the following Listing 4.1, the internal `DcpPduSync` (and its derived) objects can store the socket time. This is either the time, the message was sent or received. Looking at the function call in Listing 4.2, this time is passed back and forwarded to the second `INF_sync` PDU.

```
void INF_sync(const uint8_t dcpId,
  const DcpSyncOp syncOp, const timespec& time,
  timespec& socket_time)
```

```
{
  DcpPduSync pdu = { dcpId, getNextSeqNum(dcpId),
    syncOp, time };
  driver.send(pdu);

  socket_time.tv_sec = pdu.socket_time_.tv_sec;
  socket_time.tv_nsec = pdu.socket_time_.tv_nsec;
}
```

Listing 4.1: Masters `INF_sync` implementation. *Socket Time* is retrieved from the *PDU* object

```
timespec time = {0, 0};
INF_sync(get_id(), DcpSyncOp::SYNC, time, time);
INF_sync(get_id(), DcpSyncOp::FOLLOW_UP, time, time);
```

Listing 4.2: Sending `INF_sync` PDUs where the *socket time* is forwarded to the second message

In addition to sending `INF_sync` PDUs of type *Sync* and *Follow-Up*, the DCP master must also respond to the `RSP_sync` message from the DCP slave. Looking at the Listing 4.2 showing the response handler to the `RSP_sync` PDU with `sync_op` equals *Delay-Req* it shows that the DCP master simply sends the retrieved socket time back to the DCP slave.

```
DcpPduSyncAck sync = static_cast<DcpPduSyncAck&>(msg);
if (static_cast<DcpSyncOp>(sync.getSyncOp()) ==
  DcpSyncOp::DELAY_REQ)
{
  DcpPduSync syncRsp = { sync.getSender(),
    sync.getRespSeqId(), DcpSyncOp::DELAY_RSP,
    sync.socket_time_ };
  driver.send(syncRsp);
}
```

Listing 4.3: Master receives a sync response and retrieves *socket time* from PDU object.

This concludes the basic changes to the DCP master implementation. The offset calculation is implemented in the DCP slave, where the clock is adjusted as well.

4.2.5 Integration of PTP in DCP Slave

Unlike the PTP master, the PTP slave does not initialize the synchronization. The PTP slaves must correct their clock difference to the PTP master. Thus, the same applies to the DCP slave. To do that, they must collect all four timestamps. The formula to correct the clock difference Θ is listed in Equation 2.5. The timestamps are numerated τ.

- τ_1: Send time of the *Sync* message. This timestamp is received from the master as payload of the *Follow-Up* message.
- τ_2: Receive time of the *Sync* message. This timestamp is collected from the slave as soon as possible.
- τ_3: Send time of the *Delay-Req* message. This timestamp is collected at the slave once the message is sent to the master.
- τ_4: Receive time of the *Delay-Req* message. This timestamp is retrieved at the master and sent to the slave by utilizing the *Delay-Rsp* message.

Once the `INF_sync` PDU with `sync_op` equals *Delay-Req* is received, the difference can be calculated. The DCP slaves logic retrieve logic is depicted in the following Listing 4.4. Note that the `DcpPdu` and its derived objects are shared between the DCP master's and DCP slave's source code. Thus, the socket time is stored in these objects. Again, the socket time represents the time, the PDU was either sent or received.

```
DcpPduSync& sync = static_cast<DcpPduSync&>(msg);
case DcpSyncOp::SYNC: {
  syncReceived = sync.socket_time_;
  break;
}
case DcpSyncOp::FOLLOW_UP: {
  syncSend = sync.getTimespec();

  DcpPduSyncAck syncAck = { sync.getReceiver(),
    sync.getPduSeqId(), DcpSyncOp::DELAY_REQ, {0, 0} };
  driver.send(syncAck);
  delayReqSend = syncAck.socket_time_;
  break;
}
```

```
case DcpSyncOp::DELAY_RSP: {
  delayReqReceived = sync.getTimespec();

  // calculate Offset:
  std::chrono::microseconds t1, t2, t3, t4, newOffset;
  t1 = DcpPduSync::fromTimespec(syncSend);
  t2 = DcpPduSync::fromTimespec(syncReceived);
  t3 = DcpPduSync::fromTimespec(delayReqSend);
  t4 = DcpPduSync::fromTimespec(delayReqReceived);

  newOffset = (t2 - t1 - t4 + t3) / 2;
  dSyncOffset = syncOffset - newOffset;
  syncOffset = newOffset;
  break;
}
```

Listing 4.4: Slave handles `INF_sync` reception and retrieves all required timestamps to calculate the clock difference `newOffset`.

Rapid changes to the clock may result into loosing synchronization between the DCP slaves and the global simulation clock. This would have caused the opposite effect of what the solution tries to solve. Thus, the calculated difference is gradually applied to the DCP slaves clock. It is worth to mention, that PTP even enforces such behavior. That is to say, the DCP slave tries to approximate the DCP master's clock by small changes. This is achieved, by splitting the total difference over the expected steps during the next synchronization interval. The actual correction per step θ is calculated using Equation 4.3.

$$\begin{aligned} \theta &= \frac{\Theta}{\frac{numerator_{sync}}{denominator_{sync}} / \frac{numerator_{step}}{denominator_{step}}} \\ &= \frac{\Theta}{interval_{sync} / interval_{step}} \end{aligned} \tag{4.3}$$

In this oversimplified depiction, the synchronization interval is defined to be 10 times the step interval. Thus, the clock difference per step $\theta = \frac{\Theta}{10}$. The resulting value is added to the *Next Communication* timestamp,

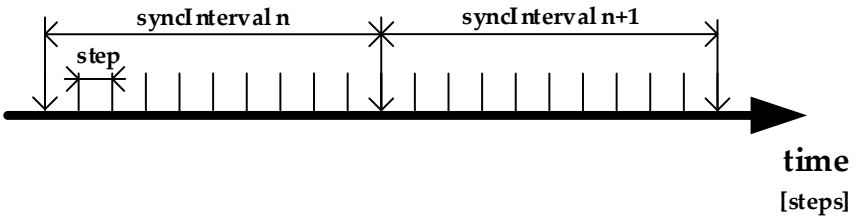

Figure 4.1: Comparison of step and synchronize interval

which denotes the start time of the next simulation step. See Listing 4.5 for implementation of Equation 4.3.

```
int64_t stepInterval = (int64_t) (
  (double) numerator / (double) denominator *
  (double)steps * 1000000.0);
nextCommunication += stepInterval;

if (CapabilityFlags.canHandleTimeOffset) {
  int64_t syncInterval = (int64_t) (
    (double) syncNumerator / (double) syncDenominator
    * 1000000.0);
  nextCommunication -= dSyncOffset /
    (syncInterval / stepInterval);
}
```

Listing 4.5: Offset is applied gradually after each simulation step.

4.3 POSIX timestamping API

In the previous section, the term "*socket time*" was mentioned a few times. The *socket time* describes the time when the message is as close as possible to the physical unit. This way the captured timestamp does not include various unpredictable software overhead. This overhead is caused for example by

program code, locking mechanism and message queues. Especially due to the semaphores and mutexes and given the fact that neither Windows nor Linux is an real-time operating system this overhead is unpredictable. As a consequence timestamps taken at the program logic suffer uncertainties and thus inaccurate correction values. In optimal case it is the exact time, the message was received or sent. In reality, however this is impossible to achieve. The highest accuracy is realizable using special hardware. This comes at a price, as PTP enabled hardware is not cheap. In addition to that, there is also a practical software based solution. It requires PTP enabled drivers and support from the operating system.

Linux is designed after the principle *Everything is a file*. The API provides a simple functions for each files. These are for example `open`, `close`, `read` and `write` only to name a few. The *file* for network operations is called sockets. Sockets are quite complex structures. Thus, they also come with a lot of additional functionality and configurations. Sockets can be broken down into different API levels, each describing an layer in the OSI model. The next sections describe which options need to be enabled to enable timestamping support.

Nowadays almost any networking driver comes with PTP support enabled. In terms of operating system, only POSIX compatible systems offer an interface. The "*timestamping*" interface[3] offers userspace access to these timestamps. The accuracy of this interface depends on the available resources. It will utilize the hardware if possible or fallback to the software based approach. If neither the hardware nor the driver based implementation is available, the interface is disabled.

4.3.1 Enabling timestamping API

In order to utilize the timestamps, they must be enabled. In Linux the syscall `ioctl`[4] is used to manipulate device parameters[28].

```
int ioctl(int fd, unsigned long request, ...);
```

[3]https://www.kernel.org/doc/Documentation/networking/timestamping.txt
[4]http://pubs.opengroup.org/onlinepubs/9699919799/functions/ioctl.html

Listing 4.6: Signature of POSIX function `ioctl`

Using this syscall on the physical network device like "`eth0`" the socket timestamping can be enabled. As many Linux based syscalls, `ioctl` takes a open file descriptor as its first argument. The second argument represents a request code, individual for each device group. It defines which device parameter should accessed, weather the access is read or write and how large the provided buffer must be. A pointer to this buffer is the third argument passed to `ioctl`. The buffer must meet the conditions defined to the request code. Usually a struct is defined alongside which can be used. This ensures that the access is readable and that enough memory is available. Should the syscall fail, the return value will be `-1` and the global `errno` is set appropriate. Summing up, the kernel interface to manipulate device parameters is quite flexible.

It is also worth to mention, that `ioctl` usually requires the binary to be executed with higher privileges. Failing to do so will abort the function call with the following error message: "Operation not permitted". Depending on the application use-case this may be a massive issue.

The request code to enable hardware timestamping is `SIOCSHWTSTAMP`. It expects a struct of type `ifreq`. In order to work, two fields of the `ifreq` struct must be set. `ifr_name` should store the network device name and `ifr_data` should point to an valid `hwstamp_config` object. The `hwstamp_config` object contains additional data required to modify the hardware timestamping. Setting `tx_type` and `rx_filter` can be used to enable or disable the feature for any communication direction. Listing 4.7 shows how hardware timestamping is enabled.

```
ifreq device();
hwtstamp_config hwconfig();

strcpy(device.ifr_name, "etho");
device.ifr_data = &hwconfig;
hwconfig.tx_type = HWTSTAMP_TX_ON;          // ..._OFF;
hwconfig.rx_filter = HWTSTAMP_FILTER_ALL; // ..._NONE;
```

```
int err = ioctl(socketFd, SIOCSHWTSTAMP, &device);
```

Listing 4.7: Enable hardware timestamping using `ioctl`

Besides enabling hardware timestamping, the Kernel must also be notified. Otherwise it will discard the timestamp and not forward it to the userspace. However sockets are a bit more complex. On one side, they communicate with the application via an API. On the other side, sockets simply hand over byte streams to the network device. Recalling the OSI model, many layers are in between.Sockets also handles various networking protocol features like checksums, retransmissions (in case of streaming sockets), and many more. Similar to devices, sockets can be manipulated by utilizing a syscall. For sockets this function is `setsockopt`[5][28].

```
int setsockopt(int sockfd, int level,
  int optname, const void *optval, socklen_t optlen);
```

Listing 4.8: Signature of POSIX function `setsockopt`

Again a file descriptor is used to select the correct object. The `level` parameter, is used to define at which implementation level the changes should be applied. As described above, operations of a socket can span over multiple OSI layers. Using this function it is possible to configure properties of layer 4 (TCP, UDP) and layer 3 (IPv4, IPv6). In addition, various API level changes can be applied, by passing the `SOL_SOCKET` constant. The remaining three parameters are similar to `ioctl`. `optname` is similar to the request code. `optval` and `optlen` are simple pointers to an valid object. Unlike `ioctl` the size of the object is passed as well. Again the object type is linked to its object code. However, for socket manipulations, most objects are simple integers. Thus, using bitwise or it is possible to set multiple options using only one function call. Like many other Linux syscall functions, the return value indicates success or failure. In the case of an error, the global `errno` field is set appropriate. For timestamping support, options for `SO_TIMESTAMPING` must be set. `SO_TIMESTAMPING` is located at the API level. See the Listing 4.9 for implementation details.

```
int options = SOF_TIMESTAMPING_RAW_HARDWARE
            | SOF_TIMESTAMPING_TX_HARDWARE
```

[5]http://pubs.opengroup.org/onlinepubs/9699919799/functions/setsockopt.html

```
                | SOF_TIMESTAMPING_RX_HARDWARE;
int err = setsockopt(socketFd, SOL_SOCKET,
  SO_TIMESTAMPING, &options, sizeof(options));
```

Listing 4.9: Prepare kernel to collect timestamps for receiving and sending datagrams.

This concludes the prerequisites to retrieve socket timestamps. The passed arguments vary depending on the used hardware. The code in Listing 4.9 assume that PTP hardware is available. If this is not the case, some of functionality may not work. This is especially true for the `ioctl` configurations. Although hardware timestamping may not be available in this case, the operating system and the socket may be configured to use software timestamping. It goes without saying that this is less accurate, but still achieves better results than pure userspace implementations. The constants `SOF_TIMESTAMPING_(TX/RX)_SOFTWARE` should be used in this case. Anyways, a successful `setsockopt` function call is required for socket timestamping to work.

4.3.2 Retrieving timestamps

Once at least the `SO_TIMESTAMPING` for the given socket is enabled, timestamps can be retrieved. The interface works independent on the used timestamping mode. That is to say, hardware support is beneficial but not strictly required. Timestamps are retrieved using the sockets read function. Sockets use functions of the family `recv`[6] to receive data and `send`[7] to send data. As a socket can be broken down to a file, these functions are basically wrappers around `read` and `write`[28].

```
ssize_t recv(int sockfd, void *buf, size_t len,
  int flags);
ssize_t send(int sockfd, const void *buf, size_t len,
  int flags);
```

Listing 4.10: Signature of POSIX functions `recv` and `send`

[6] http://pubs.opengroup.org/onlinepubs/9699919799/functions/recv.html
[7] pubs.opengroup.org/onlinepubs/9699919799/functions/send.html

The `recv` functions are used to load the received data from the socket into the userspace application. If not defined otherwise, these functions usually block the execution and wait until data is available. Their return value indicates the number of received bytes. If an error was raised the return value is `-1` and `errno` is set. In addition, some of them also support access to *ancillary data*. Internally *Ancillary data* are described as *control messages*. They are not part of the actual payload but include additional metadata and information over all socket *levels*. If enabled, timestamp information is stored as a *control message* at level `SOL_SOCKET`. Given a separate buffer in addition to the payload buffer the kernel will store the metadata automatically upon reception. *Control message* can be retrieved utilizing the `recvmsg` function. Naturally, this is a derived form of `recv`. This function must be used in conjunction with the `msghdr` struct. In order to minimize arguments and improve code readability, many parameters are combined in this struct. Using `msghdr`, buffers for payload, remote endpoint names and control messages can be passed.

```
ssize_t recvmsg(int sockfd, struct msghdr *msg,
  int flags);
```

Listing 4.11: Signature of POSIX function `recvmsg`

While endpoint information and payloads can easily be retrieved directly or using vectored-I/O[8], the *control message* buffer can store several *control messages*. Multiple ancillary data are stored successively in the *control message* buffer. In order to access a single message, the *cmsg* C-Macros as well as the `cmsghdr` struct should be used. The macros define a simple interface allowing to iterate over the *control message* buffer. See Listing 4.12 for the interface and members of the control headers.

```
struct cmsghdr* CMSG_FIRSTHDR(struct msghdr* msgh);
struct cmsghdr* CMSG_NXTHDR(struct msghdr* msgh,
  struct cmsghdr* cmsg);

unsigned char* CMSG_DATA(struct cmsghdr* cmsg);

struct cmsghdr {
```

[8] pubs.opengroup.org/onlinepubs/9699919799/functions/readv.html

```
  size_t cmsg_len;    // Data byte count
  int    cmsg_level;  // Originating protocol
  int    cmsg_type;   // Protocol-specific type
  /* followed by
   * unsigned char cmsg_data[];
   */
};
```

Listing 4.12: Interface and object definition of the Control Message Header

Ancillary data is categorized into API levels (`cmsg_level`) of the socket, depending on their origin. As sockets span multiple layer of the OSI model, ancillary data from multiple protocols can be retrieved. In addition to API level, the messages can be distinguished by the `cmsg_type` field. Multiple values are defined, and they describe the size and the memory layout of the data buffer. The buffer itself is accesses as an `unsigned char` pointer using the `CMSG_DATA` macro. Typically, it must be casted to the expected type. As the data is stores successively in the buffer, the length of the *control message* is important.

For timestamp information, the API level is `SOL_SOCKET` and the type is `SO_TIMESTAMPING`. Obviously this is equivalent to the parameters passed to `setsockopt`. The data is of type `struct timespec` and thus has a resolution of nanoseconds. The Listing 4.13 below shows how the *socket time* can be extracted from the *control message*.

```
msghdr msg();           // Msg Header Struct
char control[128];      // CtrlHeader buffer

// Set msg fields accordingly
// ...
// Pass control header buffer
msg.msg_control = control;
msg.msg_controllen = sizeof(control);
int bytes = recvmsg(socketFd, &msg, 0);

for (cmsghdr* cmsg = CMSG_FIRSTHDR(&msg);
     cmsg; cmsg = CMSG_NXTHDR(&msg, cmsg))
```

```
{
  if (cmsg->cmsg_level == SOL_SOCKET &&
      cmsg->cmsg_type == SO_TIMESTAMPING)
  {
    timespec* timestamp = (timespec*) CMSG_DATA(cmsg);
  }
}
```

Listing 4.13: Extraction of the collected timestamp from the control message headers from incoming data.

The timeflow for receiving messages and retrieving the payload as well as the *control message* data to the userspace is depicted in Figure 4.2. It shows the sequence of important events and at which protection level they are executed. Upon reception of a message, the timestamp is captured and the

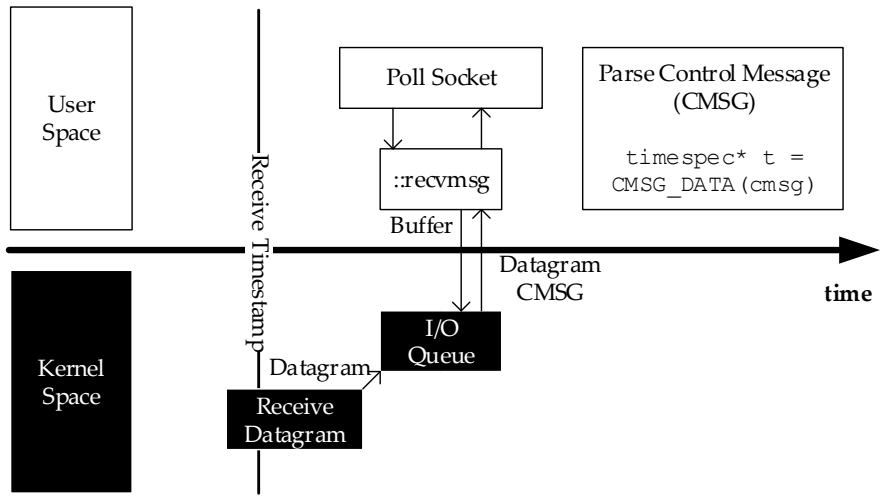

Figure 4.2: Timeline of receiving messages.

data is processed. The payload itself is pushed into the kernel internal *I/O Queue*. This queue is consumed by the `recvmsg` function in the userspace via a syscall. Together with the payload, the timestamp is queried as well. The extraction of the timestamp from the *control message* buffer is performed in

the userspace. This image also shows the importance of *socket time* and how early it can be captured. Using a timestamp when the payload is available at the userspace would also imply a huge overhead. The overhead is caused by internal task and I/O queueing, possible polling interval as well as more computation time. In total this overhead would be unpredictable and thus introduce errors.

In contract to receiving, retrieving timestamps using `write` syscalls (or their socket counterparts) is not possible. The `send` function does not provide the possibility to return data back to the callee. This is true for timestamps or any other `control message` data. In addition errors raised in lower layers (socket levels) cannot be reported in detail. In order to cope with this problem, these errors are stored in a special queue called `MSG_ERRQUEUE`. As this queue is essentially the same as any other kernel queue, `recv` can be used to load data from it by setting `flag` to `MSG_ERRQUEUE`. Consequently, it is also possible to load ancillary data from it using `recvmsg`. Once data is loaded from this queue, it can be parsed similar to data received via transmission. The loaded payload is the data which caused the error and the *control message* buffer contains ancillary data including detail information on the error. For the *control message* containing the error, the *level* and *type* is set accordingly. If enabled, timestamp information is also stored in the ancillary data. Summing up, in order to retrieve the ancillary data, the `MSG_ERRQUEUE` must be queried after sending the payload. This is also true, if no error occurred. That is to say, the `MSG_ERRQUEUE` is used as a feedback queue in any case despite the name error in it. Unfortunately in some cases polling is required, as sending data could take some time. As a result the `MSG_ERRQUEUE` cannot be filled. Alternatvely, `send` could be used in blocking mode, this guarantees that the transmission has completed upon function return.

In order to load the “send” timestamp, the message must simply be transmitted using the function `send`. After this call is completed, a consequent call of `recvmsg` with `flag &= MSG_ERRQUEUE` is required. Unlike during the receiving operation, the payload is not important while sending. Thus, the payload vector can be `NULL`. Only the `msg_control` field must point to a valid buffer. Parsing the content of this buffer is identical to receiving payload. The important code lines are in Listing 4.14.

```
msghdr msg();
char control[128];
const char* buffer = "Sample";

int bytes = send(socketFd, buffer, strlen(buffer));

// Set msg fields accordingly
msg.msg_control = control;
msg.msg_controllen = sizeof(control);
bytes = recvmsg(socketFd, &msg, MSG_ERRQUEUE);

for (cmsghdr* cmsg = CMSG_FIRSTHDR(&msg);
      cmsg; cmsg = CMSG_NXTHDR(&msg, cmsg))
{
  if (cmsg->cmsg_level == SOL_SOCKET &&
      cmsg->cmsg_type == SO_TIMESTAMPING)
  {
    timespec* timestamp = (timespec*) CMSG_DATA(cmsg);
  }
}
```

Listing 4.14: Extraction of the collected timestamp from the control message headers in case of sending data.

As the implementation, the timeflow for sending data and retrieving the send timestamp is a bit more complex. The timeflow is shown in Figure 4.3. Again it shows the succession of important events and in which protection level they take place. Data is sent using the `send` syscall. In kernel mode, the payload is pushed into the I/O queue where it resides until it is scheduled for send. The *socket time* is captured shortly after sending. The timestamp is stored in the *control message* alongside varios other ancillary data. If an error occurred, it is also stored as an ancillary data. The ancillary data is then stored in the error queue, where it can be queried using `recvmsg`. Finally, the extraction of the timestamp from the *control message* buffer can be performed in the userspace. Like its receive counterpart, this image also shows the importance of capturing the timestamp in the kernel (or hardware). Many operations with unpredictable execution time are carried

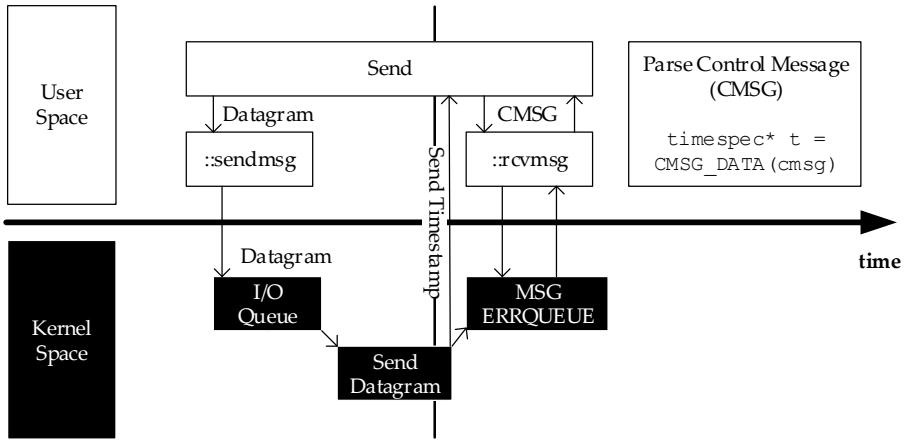

Figure 4.3: Timeline for sending messages.

out in the userspace as well as in the kernelspace.

4.4 ASIO modifications

The modelica reference implementation relies upon ASIO for platform independent socket access. The latest version of ASIO does not support socket timestamping. Thus, some changes were applied to the ASIO source so that the DCP library can utilize it. Although ASIO is platform independent, socket timestamping is only available on POSIX based systems. Consequently, the changes were only applied to the POSIX subsystem, while the interfaces were adjusted globaly to ensure a successful build. For the *winsocket* subsystem the interface changes are simply ignored.

4.4.1 Enabling timestamping API

ASIO provides wrapper methods for `ioctl` and `setsockopt`. These wrappers are `io_control` and `set_option` of ASIOs `basic_socket` class. Consequently,

enabling the SO_TIMESTAMPING feature is possible without changing any ASIO sourcefiles.

To begin with, ioctl is quite flexible. Consequently, ASIOs wrapper needs to be flexible as well. The POSIX function supports arbitrary objects passed depending on the passed *request code*. In order not to limit the functionality, ASIOs io_control method takes an object which implements the IoControlCommand interface. An optional second parameter is a C++ error object std::error_code. ASIO automatically parses the errno if required.

```
template <typename IoControlCommand>
void basic_socket::io_control(
  IoControlCommand& command, asio::error_code& ec);

class IoControlCommand
{
public:
  int name() const:

  void set(bool value);
  bool get() const;
  ioctl_arg_type* data();
  const ioctl_arg_type* data() const;
};
```

Listing 4.15: ASIOs interface for ioctrl access.

Each method is linked to a parameter of ioctl. The name method represents the *request code*. The passed object pointer is retrieved using the data methods. A typical implementation holds the object internally. This also ensures that the object gets properly cleaned by relying on RAII. The internal state should be modifiable using the set method. The ASIOs interface to call ioctrl is in Listing 4.15 and Listing 4.16 shows how to invoke the function.

For timestamping the *request code* – method name – is SIOCSHWTSTAMP. The internal object must be struct ifreq and struct hwtstamp_config. Using set, hwtstamp_config's tx_type and rx_filter is modified.

```
std::error_code err;
io_hwstamp_command command;
socket->io_control(command, err);
```

Listing 4.16: ASIOs wrapper of `ioctrl` invoked to enable hardware timestamping.

Next the socket must be configured to support timestamping. As already described, the `basic_socket` class provides a `set_option` method. While in theory the POSIX function `setsockopt` is as flexible as `ioctl`, it only uses integers as objects. Thus the wrapper is drastically simpler and does not require custom objects.

```
template <typename SettableSocketOption>
void basic_socket::set_option(const
  SettableSocketOption& option, asio::error_code& ec);
```

Listing 4.17: ASIOs interface to set socket options

ASIO provides a few predefined template classes which can be used in conjunction to this method. These classes are defined in the `socket_option` namespace. Their interface is similar to the one for `io_control`, having methods for `level`, `name` and `data`. Socket *level* and *request code* are passed at compile time as template parameters. Their value can simply be assigned using an overloaded operator. For error handling, an optional second parameter may be used. In case, ASIO parses the `errno` value and assigns it to an standard C++ error object.

In terms of timestamping support, the `socket_option::integer` class is used. The level is `SOL_SOCKET`, with *request code* `SO_TIMESTAMPING`. The values are simply assigned. The flags are designed so that bitwise or operations can combine multiple values. Listing 4.18 shows how `SO_TIMESTAMPING` can be enabled using ASIOs interface.

```
std::error_code err;
asio::detail::socket_option
    ::integer<SOL_SOCKET, SO_TIMESTAMPING>
  timeStampFlags;
timeStampFlags = SOF_TIMESTAMPING_RAW_HARDWARE
               | SOF_TIMESTAMPING_TX_HARDWARE
```

```
                | SOF_TIMESTAMPING_RX_HARDWARE;
socket->set_option(timeStampFlags, err);
```

Listing 4.18: Enable SO_TIMESTAMPING using ASIOs wrapper functions.

4.4.2 Retrieving timestamps

Unfortunately ASIO does not retrieve the *control message* headers. Although it uses `recvmsg` internally, it omits the buffer for the ancillary data. Consequently, ASIO must be modified from the lowest function back to the library interface, without breaking existing code or interfaces. The POSIX functions are accessed using wrapper functions defined in `asio::detail::socket_ops` namespace. There is a wrapper for each function of the `recv` family.

```
signed_size_type recvmsg(
  socket_type s, buf* bufs, size_t count,
  int in_flags, int& out_flags,
  asio::error_code& ec);
```

Listing 4.19: Signature of internal ASIO wrapper functions to receive data

To load ancillary data using this wrapper, the signature must be changed. Buffers for *control message* data must be added. See Listing 4.19 and Listing 4.20 for the old and updated signature.

```
signed_size_type recvmsg(
    socket_type s, buf* bufs, size_t count,
    int in_flags, int& out_flags,
    void* control, size_t controllen,
    asio::error_code& ec);
```

Listing 4.20: Updated signature of internal ASIO wrapper functions to receive data

The implementation checks whether the passed arguments are valid and forwards them to the syscall. The windows implementation simply ignores the parameters. These wrapper functions are access through varios other implementation functions. The first class accessing these wrapper functions is `reactive_socket_service`. As the name already suggests it is a base class

for ASIO sockets. The relevant functions take an *endpoint* pointer, a buffer and flags.

```
template <typename MutableBufferSequence>
size_t receive_from(implementation_type& impl,
  const MutableBufferSequence& buffers,
  endpoint_type& sender_endpoint,
  socket_base::message_flags flags,
  asio::error_code& ec);
```

Listing 4.21: Signature of ASIOs receive function

The ASIO endpoint is a class storing varios information on the connection. Most noteable it is used to store the remote's IP address and port. It is expanded to support storage for the *control messages* as well. A class – `control_header` – to encapsulate the raw buffer and provide some helper functions around the C-Macros is used. The endpoint class offers and API to control a private instance of the `control_header` class. Using this API the buffer size can be set and thus enabling the mechanism. Once the socket consumes the endpoint, it will also check for a valid *control message* buffer. If this is the case, it will also be used during the `recvmsg` call. Upon reception the endpoint *control message* buffer can be queried.

```
lastAccess.control_size(128);
socket->async_receive_from(
  asio::buffer(data + 4, maxLength),
  lastAccess,
  std::bind(&Socket::handle_receive, this,
    std::placeholders::_1,
    std::placeholders::_2));

void handle_receive(const std::error_code &error,
  std::size_t bytes_transferred)
{
  DcpPdu *pdu = makeDcpPdu(data, bytes_transferred);
  for (control_header& chdr : lastAccess.control())
  {
    DcpPduSync &sync = static_cast<DcpPduSync &>(dcp);
    if (chdr.level() == SOL_SOCKET &&
```

```
        chdr.type() == SO_TIMESTAMPING)
      {
        sync.socket_time_ = *chdr.data<timespec*>();
      }
    }
}
```

Listing 4.22: Memory for control buffer is allocated and evaluated upon data reception.

The snippet in Listing 4.22 allocates 128 bytes for the *control message* buffer and starts an async receive. That is to say, the execution continues and once data is received, the given callback function `handle_receive` is triggered. In this function, the ancillary data can be retrieved from the endpoint. Like the POSIX implementation in Listing 4.13, the *control message headers* are iterated until the correct data is found. It is identified by `SOL_SOCKET` as *level*/`name` and `SO_TIMESTAMPING` as *request code*/`type`.

This concludes all necessary changes to implement PTP support for DCP. ASIO was slightly modified to add support for ancillary data. With these changes DCP is able to retrieve timestamps which are captured shortly after reception and before sending. This greatly enhances the accuracy of the clock difference. Finally, DCP was modified to apply the calculated clock difference by adapting the real-time step size.

5 Test Environment

5.1 Simulation Layout

In order to verify a working implementation of node synchronization, a test environment is required. It goes without saying that this setup should practically utilize DCP, synchronize clocks and execute a Co-Simulation in (soft-) real-time mode. In the previous chapter, *DCPLib* from modelicas Github[1] page was expanded upon clock synchronization support. Consequently, this modified version is also used for the test bench setup. Naturally, the modified ASIO is also used.

The main demonstrator should simulate a Lane Keep Assist System (LKAS) for a simple vehicle on the road. In a simulation, the system handles the steering of a car and tries to keep it on the center lane. It goes without saying, that the simulation should be build in a distributed manner. The demonstrator comprise of a sensor model, a controller model and an environment simulation, each on an individual node. In addition, a DCP master is required to control the demonstrator.

The LKAS system is implemented in the DCP scope, distributed on two slaves. It is implemented as an closed loop controller and calculates the *steering angle* of the car based on various input values. The model is based on the simple kinematic model of an automobile[31]. One slave – the *controller module* – handles the calculation of the parameters. The second slave acts as a bridge between the DCP scope and the environment simulation. It "simulates" the sensor input and thus is named *sensor module*.

Finally, the environment simulation runs outside of the DCP scope. It communicates through the *sensor module* with DCP. The environment simulation

[1] https://github.com/modelica/DCPLib

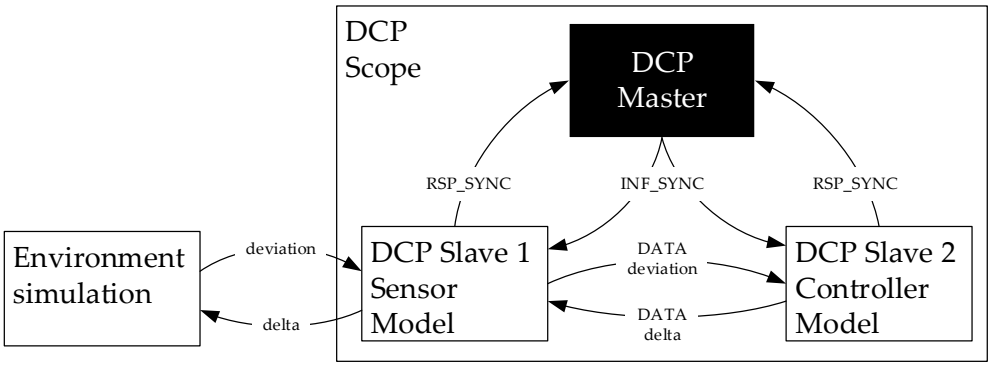

Figure 5.1: Block Diagram of the test environment

is basically a CarMaker/Simulink[2] project. Once started, it runs the simulation in constant loop until a stop criteria is met. This could be a time limit or a distance.

5.2 Used hardware

The DCP nodes are running on three identical BeagleBone Black[3]. The boards a powered by an AM3358 Cortex-A8 CPU at 1GHz from Texas Instruments. They can support any Linux 32 bit ARM support. In this example, the OS was Debian Linux. Obviously they are connected via Ethernet to each other. Thus the simulation uses UDP/IPv4 as its communication channel. The used Ethernet driver supports PTP natively.

The environment simulation runs on a standard PC, connected to the same network as the DCP nodes. For the communication between the *sensor model* and the *environment* a UDP connection was used as well. This connection is independent to the DCP communication.

[2]https://ipg-automotive.com/de/produkte-services/simulation-software/carmaker/

[3]https://beagleboard.org/black

5.3 Simulation properties

To begin with, the environment was designed to be as constant as possible. The LKAS – which is simulated in the DCP scope – runs on a simple passenger car with a constant speed of 50 km/h. The DCP scope runs in real-time mode with a step size of 20ms.

5.3.1 Virtual road

The virtual road is based on a lemniscate shape. A lemniscate is not to complicated and thus keeps the simulation simple. It also offers every possible steering direction. The track has two straights combined with one right and one left turn. The straight parts between the two turns also helps the system to stabilize. The total length of one lap at the center line is one kilometer. The car started at the intersection of the lemniscate, and at the center of the road. This ensures that the simulation is not falsified during the initialization phase, in which the car accelerates to the target speed. The shape of the virtual road is captured in Figure 5.2.

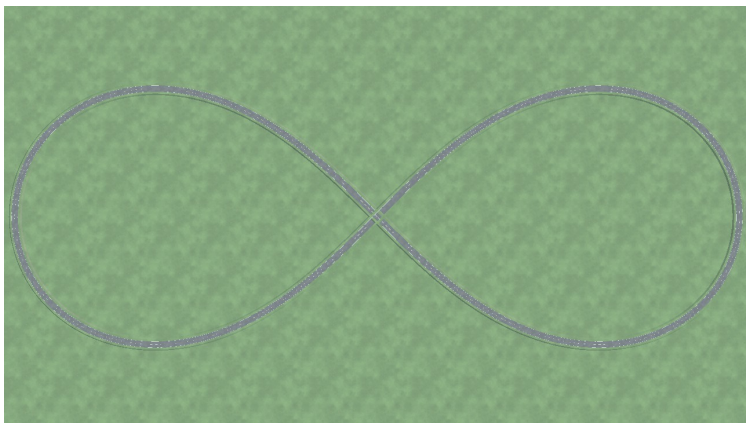

Figure 5.2: Aerial perspective of the virtual road.

5.3.2 Controller

The controller is solely implemented in the DCP scope. It is an "Stanley" controller of the Stanford Racing Team. They participated with this controller at the "DARPA Grand Challenge" in 2005[31]. They provided both a longitudinal and a lateral controller. The longitudinal controller is responsible for the vehicles speed. To simplify this model, the longitudinal controller is omitted. The speed is fixed to 50 km/h. The lateral controller handles the steering of the vehicle. It is based on the kinematic equation of motion. The basic formula for the *steering angle* $\delta(t)$ is described in Equation 5.1.

$$\begin{aligned}\delta(t) = \psi(t) - \psi_{ss}(t) + arctan\frac{k \cdot e(t)}{k_{soft} + v(t)} \\ + k_{d,yaw} \cdot (r_{trj}(t) - r(t)) \\ + k_{d,ste} \cdot (\psi_{i-1} - \psi_i)\end{aligned} \tag{5.1}$$

The parameters of Equation 5.1 are described in Figure 5.3. To begin with

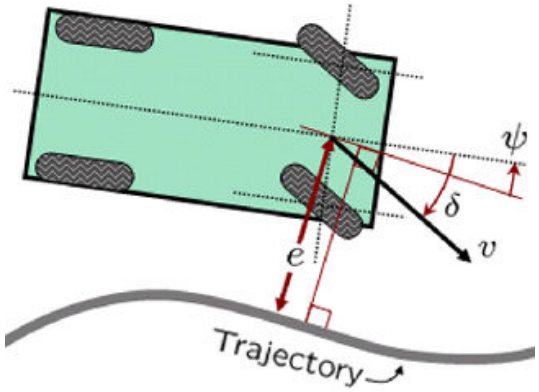

Figure 5.3: Kinematic model of a automotive as defined in[31].

the "cross track error" $e(t)$ basically describes the distance between the car and the road. It is defined as the shortest distance between the cars current position and the nearest point on the road's trajectory. Given this point on the trajectory, the yaw angle $psi(t)$ is calculated. The yaw angle is the deviation angle between the road's trajectory and the vehicle's heading. Finally, the yaw rate is denoted as $r(t)$. The yaw rate is defined as the

angular velocity of the rotation. Meaning it is the first derivative of the yaw angle. In addition to the cars yaw rate, the controller also relies on the yaw rate of the deviation angle $\psi(t)$. This yaw rate r_{trj} is described in Equation 5.2.

$$r_{trj} = \frac{v(t)sin(\psi(t))}{a+b} \tag{5.2}$$

The basic reference for the controller is the yaw angle $\psi(t)$. It is slightly offset by $\psi_{ss}(t)$ which depends on the yaw rate of the trajectory and the vehicle's velocity $v(t)$. It basically improves stability in case the car is exactly following the trajectory. In this case the error is zero but the required steering angle is not. Thus a set value of zero would introduce a slight oscillating behavior. The *arctan* function is used to correct an existing error. It depends on the "cross track error" $e(t)$ and the vehicle's velocity $v(t)$. With increasing error, the controller will turn towards the road more aggressively. This is essentially the proportional part of the controller and is fine tuned by setting the constant k and k_{soft}. The later one helps to limit sensitivity during low speed sections and avoid a division by zero.

The controller so far only calculates the steering angle δ. However, with increasing vehicle speed the impact of sideway velocity increases as well. Consequently, the output of the controller – the steering angle – may not reflect the actual movements of the car. In order to correct the resulting offset, a damping effect on the yaw rate is introduced to the equation. It corrects the difference between the cars yaw rate $r(t)$ and the yaw rate of the trajectory r_{trj}. Using k_{yaw} the impact of this part is fine-tuned. Finally, to compensate time delays due to latency and resulting overshoots and possible instability the steering angle is measured and normalized. The current steering angle is compared to the retrieved value from the previous step.

The actual equation used in the DARPA Challenge also considers the (physical) limit of the *steering angle*. It limits the calculated value to the appropriate min/max values. However, as the test vehicle in this simulation starts exactly at the center of the road it must not *find* the road and perform an u-turn. Furthermore, the road is designed to ensure that the trajectory stays within the cars limitations. Consequently, it will never saturate the *steering angle*.

As the model has no longitudinal controller, it cannot control the vehicles speed. Consequently, it cannot handle tighter corners as they require reduced speed in order to traverse them. As a result, the possible angle of a turn is further limited, but sufficient for the clock synchronization test model.

5.4 Test results

Firstly the effect of clock skew was measured. This was accomplished by two nodes exchanging the PTP messages. However, the calculated offset was not applied but stored in a list. As a consequence the clock difference was saved over time and is visualized in Figure 5.4. The test was conducted over nearly seven minutes. During that period, the calculation of the clock difference was performed every five seconds. In total 76 calculations were recorded. A clear incline of the clock difference can be seen. Although, some

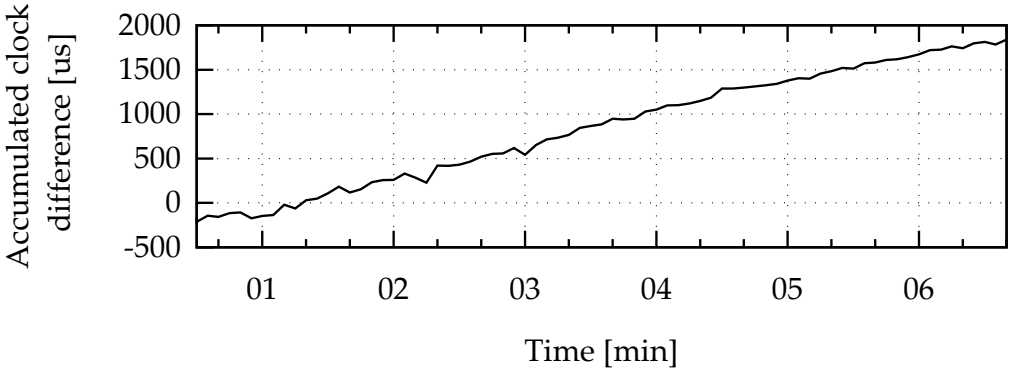

Figure 5.4: Measured Clock difference over time

outliers show that the difference is not only increasing but also decreasing. However these changes are only for a short period of time. In a global context, the difference is increasing in a more or less linear fashion. It is worth to mention, that the difference can be negative as well. This is seen at the beginning of the recording. This simply indicates that the one node surpassed the other node in time. For example, after startup node A was

ahead of B, however B's clock runs faster so that it will eventually catch up to A and pass it. The system started at $\tau_0 = -154us$ and ended at $\tau_{75} = 1839us$. That is to say, the clocks drifted apart a total of $1993us$ during the measure time of 380 seconds. The following values can be estimated: After one hour, the clock difference is already $18881us \approx 18ms$. After a whole day, the difference is approximately $453145us \approx 0.45s$. It is important to mention, that these values are greatly device dependent. Values can be vastly different even if another device of the same product is used. In practice this is no issue, as the clock synchronization is performed regularly.

Combining this values with the real-time step size, the impact on the system can be estimated as well.The clock-difference should never exceed half of the step size, otherwise it will result into oversampling (negative clock skew) or undersampling (positive clock skew) of variables[32]. This behavior was analyzed using the following tests. These tests were conducted using two simulation runs. The first run acted as a reference. It's synchronization properties were enabled. On the other hand, the next run had no sort of synchronization. To see the oversampling effect more clearly, some test cases run with various forms of *artificial clock drift*. That is to say, the clock at the *controller model* was actively disturbed. All measures were taken at the *sensor model*. Consequently, this node was synchronized to the DCP master all the time. This guarantees the equal amount of samples independent of the test run and its *artificial clock*.

The tests were conducted in the environment described above. The car started the center of the road during a straight part. It accelerated to 50 km/h and kept that speed. At each test run, the simulation lasted 120 seconds. During that time, the car drove about 1.7 km, depending on the stability of the steering. During each simulation step, the *deviation distance* σ was retrieved and a matching *steering angle* δ was calculated. The following Figure 5.6 shows the *steering angle* of the reference run. The *steering angle* matches a sine wave due to the roads shape. During a single lap, the car has to turn left $\delta > 0$, orientate itself and finally, turn right $\delta < 0$. The measured value is in radiant. For comparison, at highest point the steering wheel was turned $0.616rad = 35°$ to the left. The only point where the car was steering straightforwards was around 60 seconds, exact the time the car passes the intersection at the center.

Figure 5.5: Picture of the simulation render comparing the shaded reference car and a test run car with artificial drift of 5ms.

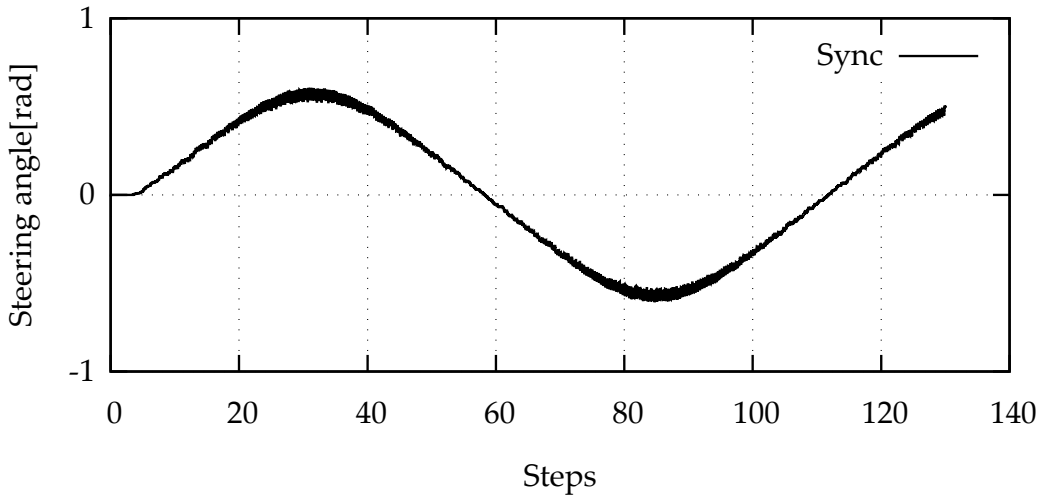

Figure 5.6: Steering angle during a simulation run.

Comparing the reference output to test runs with *artificial clock drift* a difference per step can be calculated. This is visually represented in Figure 5.7. Various information can be derived from this picture. To begin with, the

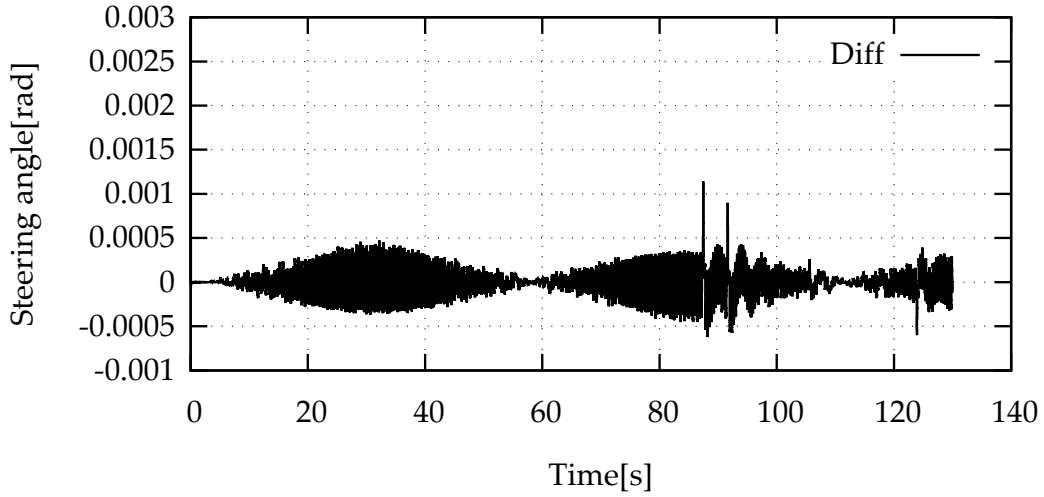

Figure 5.7: Difference of steering angle between reference run and run width *artificial clock drift*

difference is only detected when the car is in a turn. More precisely, the error even increases towards the turn apex and reduces subsequently. Next, the deviation seems to oscillate between positive and negative values. This indicates that the car is slightly oscillating. In other words, the controller is aggressively correcting the measured deviation. This behavior is caused by the difference in sampling due to the lack of time synchronization.

5.4.1 Effects of over-/undersampling

The main difference can be observed when the car approaches a turn until the apex. During this period the controller should increase the *steering angle*. It also must counter the sideways velocity, which is at its greatest once the car passes the apex. In order to counter to sideway velocity, the controller applies a negative dampening based on the vehicles yaw rate. The yaw rate is the first derivative of the yaw angle and thus is slightly time dependent.

The difference of the ideal clock and the artificially created skew clock is

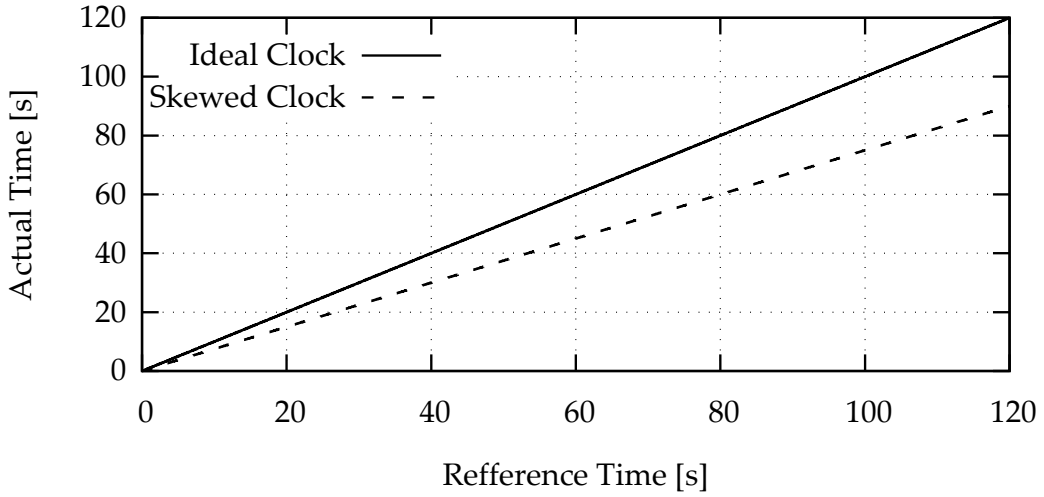

Figure 5.8: Comparison of an ideal clock and a skewed clock

shown in Figure 5.8. Over time the gab increases, which is only natural, as the skewed clock runs slightly faster each step. The ideal clock represents the DCP master. Yet, the controller is quite flexible and counters the timing issues. Nevertheless, sideway velocities depends on the vehicles speed. A higher speed may have more impact on the simulation result. Furthermore it is not guaranteed that controllers can correct themselves. It is possible that the slight error will have more impact later in the simulation and thus propagate. If the error exceed safety margins it may destabilize the controller. On one hand the steering controller may react aggressively if the error is overestimated. On the other hand underestimating the error, will cause the car to adjust its *steering angle* only slightly. In any case, the *deviation distance* will eventually be worsened. This will amplify the problem even more. In general the Table 5.1 indicates that with increasing oversampling the overall accuracy decreases. High values for the mean error key figures – MAE and RMSD – point out that the system is incorrect over long runs. These key figures incline with the applied offset. Next, the impact of offset can also be derived using the highest observed error for each test run. The *max error* is stated together with the simulation time when it occurred. By increasing the offset, the highest error is observed earlier. Exception to this are the highest

Offset	max error	MAE	RMSD
100us	0.507 (87s)	0.048	0.076
1 000us	0.427 (80s)	0.049	0.076
5 000us	0.564 (26s)	0.064	0.090
7 000us	0.571 (25s)	0.074	0.100
10 000us	2.544 (89s)	0.386	0.557
20 000us	8.763 (64s)	2.876	3.441

Table 5.1: Steering angle difference (*error*) between test runs. Max error indicates the highest difference which occurred. Further key figures are *Mean absolute error* (MAE) and *Root-mean-squared-deviation* (RMSD).

offsets of 10 000us and 20 000us. This is caused by an unstable controller as its mean errors are extremely high. These test runs already exceed the error of the first test run (100us – $\max(e) = 0.507$) after 23 seconds (10 000us) or 5 seconds (20 000us) respectively. Past this point, the steering angle of these test runs starts to oscillate. Thus the max error and the mean errors are high. On the other hand, below half of the step size, the error is steadily increasing, but the controller is able to correct itself. Consequently, the mean values stay below or around 0.1 error. The road layout helps the controller to recenter the car during the straight sections. This is visually represented in Figure 5.9. The difference between the *steering angle* increases during turns, while it stabilize during the straight sections.

5.4.2 Propagation of error

The second statement was that previously introduced error is propagated through the whole simulation. This is a well-known issue with computer simulation in general[33, 34]. Typically, these issues arise of insufficient knowledge of the model itself, e.g. weather prediction. Yet the effect is the same, independent how the (first) error was introduced.

To test this, the clock drift was only simulated through the first part of the simulation. After a set time, the clocks stayed synchronized til the end of the simulation. It is expected that an error will still be traceable after enabling

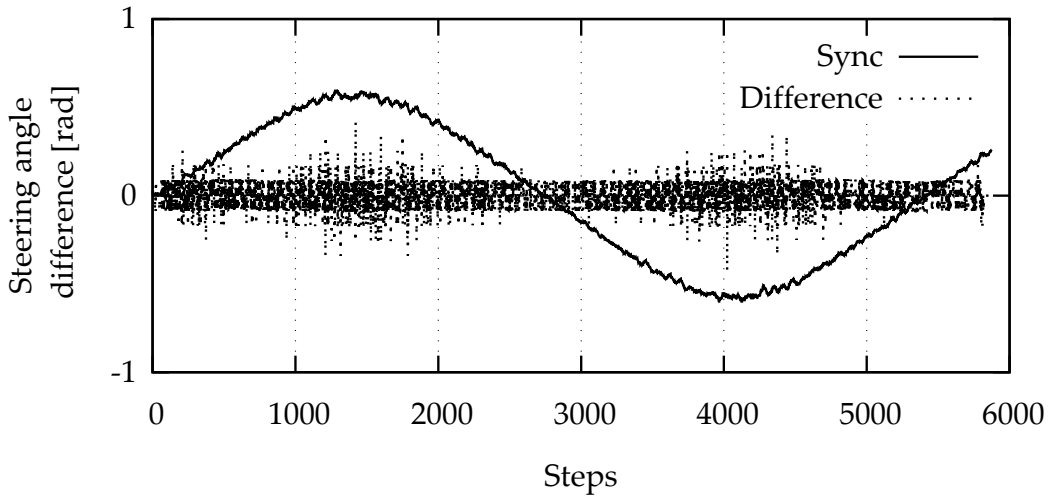

Figure 5.9: Overlay of reference test run and the difference of the steering angle between reference and 1 ooous offset test runs

synchronization. Reason for that is, that the car exited the previous turn at a complete different position or time than the reference car. As a consequence the car entered the second turn at a different position/angle as well. Thus the controller reacts from a completely new situation. It is only natural that the result will differ from the reference controller.

During the first half of the total 6 ooo steps, the clock was artificially drifted by a 5ms. After 3 ooo steps, this intervention was stopped and the clocks were synchronized. Consequently, no new error should be introduced. However as expected, the calculated *steering angle* still differed from the reference values during the second turn. Figure 5.10 shows the result of this test run. This is investigated by comparing the various error key figures between the test run above and the ordinary 5ms drift test run. The calculated *errors* are listed in Table 5.2. The values are split between the overall simulation, the first and the second half of the simulation period. In total, the test run with restored synchronization achieved minimal less average error. On one hand, the average error during the first 3 ooo steps is equal. On the other hand, it declined between 3 ooo and 6 ooo steps once the synchronization was restored. Contrary to this the other test run showed, that the *mean average error* (MAE) is fairly balanced between the

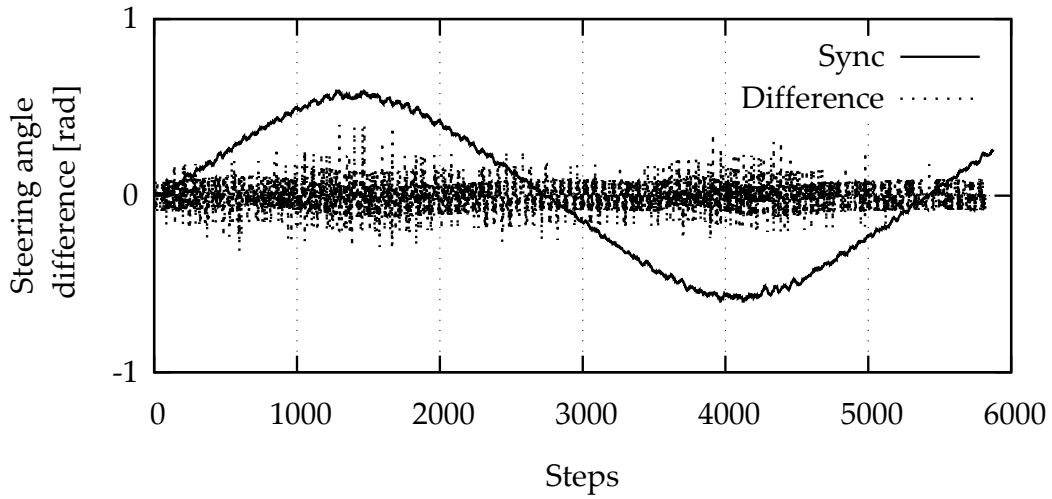

Figure 5.10: Overlay of reference test run and the difference of the steering angle between reference and offset test runs. After 3000 steps, the clocks stayed synchronized.

Drift 5ms for 6 000 steps			
Steps	max error	MAE	RMSD
0-6000	0.564 (26s)	0.064	0.090
0-3000	0.564 (26s)	0.064	0.088
3000-6000	0.480 (78s)	0.065	0.092
Drift 5ms for 3 000 steps			
0-6000	0.560 (26s)	0.058	0.079
0-3000	0.560 (26s)	0.064	0.084
3000-6000	0.352 (92s)	0.051	0.074

Table 5.2: Comparison of test runs with clock drift 5ms over whole simulation and only half of the simulation.

first and the second half. The RMSD even inclines. This behavior is expected as the error during the first half contributes to the upcoming simulation result. Consequently, the error introduced during the first 3 000 steps were traceable during the complete simulation. Surprisingly, the error declined once the synchronization was enabled again. This is due to the fact, that the main task of the controller is error minimization. Eventually it will follow the same line as the reference car again. Only the time offset caused by driving a longer distance during unstable operation cannot be corrected. Thus a constant error in the difference calculation will always be present.

6 Conclusion and Outlook

6.1 Conclusion

The effects of diverging clock difference due to drift is analyzed. It is worth to mention, that clock difference itself is no issue, as long as the step size can be guaranteed. Obviously this is only true as long as the system operates with no absolute clock, but in relation to the step size. In case absolute times are used, the difference must be corrected. Otherwise, the nodes will wait differently long. Yet this issue can easily be resolved and usually must happen only once. On the other hand the system starts to fail, if the simulation step size can no longer be guaranteed. The probability that this is poses an issue increases with reduction of step size and worsening of clock drift. That is to say, if the clock drift alters the difference more than half of the defined step size, sampling points will be missed. Consequently, the simulation is over- or undersampling. The amount of clock drift depends on the quality of the used hardware and may even vary between devices of the same product. The measurements taken for the test setup used two BeagleBones. A drift of approximately 2ms over the measure time of 360 seconds were recorded. In order to counter the clock drift, the DCP library was adapted.

The DCP changes are based on PTP, a clock synchronization algorithm. It utilizes Linux Kernel APIs to achieve a highly accurate correction rate in sub microseconds[11]. As clock drifts over time, the correction is executed at a set interval. Naturally the interval depends on the used hardware. For the test runs a interval of 5 seconds was used.

A test environment based on a carmaker simulation is used to collect various data and verify the implemented PTP algorithm. The simulation was split into three simulation nodes and kept as simple as possible. A

discrete controller based on the kinematic model of automotive was used to control a cars *steering angle*. Over- and undersampling of the simulation was tested, by operating the controller at various alternating sampling rates. Depending on the impact, the controller acts sensitive to these changes. As a consequence, the output differed from the reference car. In addition to that, the error is propagated through the simulation. An error will always affect the upcoming calculations, independent how the error was introduced in the first place. Earlier an error happens, the more uncertainty is introduced to the final result.

The test case emphasizes these issues individually. That is to say, the effects on the simulation greatly depends on used model as well. The controller is sensitive to deviations of the sampling rate. A different solution or problem may not be affected at all. Once introduced, the controller calculates a wrong value for the *steering angle*. This in turn will propagate through the simulation. Yet if the steering controller works correctly, it is able to revise the error. The road layout of a lemniscate helps the controller in this task. The road allows indefinite runs and provides two straight sections allowing the car to approximate the reference car lap by lap. Only this in conjunction with a closed loop controller was able to reduce the error, in a different application the error may be constant or even increase further.

6.2 Outlook

The existing implementation only works in the local network and can only utilize UDP. These limitations are inherited from the Kernel API and PTP. PTP is designed to work only within the same network. Furthermore, the timestamping API supports only UDP. A streaming protocol like TCP is currently not supported.

However, for future applications Co-Simulation environments may span multiple companies and should work across the globe. This implies that the simulation runs on multiple networks as well. Currently, only non-real-time operation is supported in this scenario. Obviously the synchronization task for such an environment is challenging. The Round-trip-time (RTT) cannot be assumed as symmetric anymore. Furthermore, reliability is much more

an issue than in local networks. While package drops are less likely to occur, packet ordering poses a massive problem[18]. In this context, wireless connection are also vulnerable. Especially in an Hardware in the Loop (HiL) setup with an physical moving object a wireless network is the only way the object can be interconnected. Due to the low reliability it is uncertain if the PTP algorithm is working as indented. It may even be possible that the simulation is harmed by adjusting the clock wrongly.

Besides UDP no other communication channel is supported by the implementation suggested in this work. The current DCP draft defines interfaces for CAN, TCP and Bluetooth as well. While CAN theoretically can synchronize the clock utilizing the bus, the other two interfaces must relay on external synchronization methods. Another limitation of the current implementation is, that only the DCP master is used as the clock reference. Minor improvements can be achieved by integrating a dedicated PTP grand master into the simulation network. This way all DCP nodes, including the DCP master, synchronize against the reference clock.

Summing up, with the current PTP integration into DCP, the clocks can be corrected to avoid clock differences. However, as long as the step-size can be adhered and no absolute time is required during the simulation, no issues will arise. Currently the implementation only supports UDP due to API limitations, but can be expanded upon in the future. In this case changes to the kernel and drivers may be required. Alternatively to PTP a hardware based approach may be viable in the future. Ethernet gains much attraction in the fields of industry. With Synchronous Ethernet (SyncE) a promising standard is in development allowing clock synchronization at a hardware level[35]. It utilizes ethernet's physical connection for the clock signal. Naturally special hardware is required. However, unlike the current industrial ethernet standards, SyncE is intended to work with the standard topology.

Bibliography

[1] V Paxon, G Almes, J Mahdavi, and M Mathis. Framework for IP Performance Metrics. 1998.

[2] Nikolaos M. Freris, Scott R. Graham, and P. R. Kumar. Fundamental Limits on Synchronizing Clocks Over Networks. *IEEE Transactions on Automatic Control*, 56(6):1352–1364, jun 2011.

[3] Martin Levesque and David Tipper. A Survey of Clock Synchronization Over Packet-Switched Networks. *IEEE Communications Surveys & Tutorials*, 18(4):2926–2947, 2016.

[4] Abdelrahman Abdou, Ashraf Matrawy, and Paul C. Van Oorschot. Accurate one-way delay estimation with reduced client trustworthiness. *IEEE Communications Letters*, 19(5):735–738, 2015.

[5] Peter Danielis, Jan Skodzik, Vlado Altmann, Eike Bjoern Schweissguth, Frank Golatowski, Dirk Timmermann, and Joerg Schacht. Survey on real-time communication via ethernet in industrial automation environments. In *19th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2014*, pages 1–8. IEEE, sep 2014.

[6] Hubert Zimmermann. OSI reference model–The ISO model of architecture for open systems interconnection. *IEEE Transactions on communications*, 28(4):425–432, 1980.

[7] David Mills, U Delaware, J Martin, Burbank J, and W Kasch. RFC Standard for NTP. Technical report, 2010.

[8] T. Gotoh, K. Imamura, and A. Kaneko. Improvement of NTP time offset under the asymmetric network with double packets method. In *Conference Digest Conference on Precision Electromagnetic Measurements*, pages 448–449. IEEE.

[9] D.L. Mills. Improved algorithms for synchronizing computer network clocks. *IEEE/ACM Transactions on Networking*, 3(3):245–254, jun 1995.

[10] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. Technical Report July, IEEE, 2008.

[11] Zhaoqing Liu, Dongxing Zhao, Min Huang, Yigang Zhang, and A Scheme Selection. A universal method for implementing IEEE 1588 with the 1000M ethernet interface. In *Ieee Autotestcon*, pages 1–7. IEEE, sep 2016.

[12] Igor Freire, Ilan Sousa, Aldebaro Klautau, Igor Almeida, Chenguang Lu, and Miguel Berg. Analysis and evaluation of end-to-end PTP synchronization for ethernet-based fronthaul. *2016 IEEE Global Communications Conference, GLOBECOM 2016 - Proceedings*, 2016.

[13] IEEE Standard for Ethernet. *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012)*, pages 1–4017, 2016.

[14] Pbsi. Ethernet adoption in process automation to double by 2016.

[15] Peter C Evans and Marco Annunziata. Industrial Internet: Pushing the Boundaries of Minds and Machines. *General Electric*, page 37, jun 2012.

[16] J Postel. RFC Standard for UDP. 1980.

[17] Guangjin Pan and Hanhan Xue. Real time analysis of current transport protocols in high loss networks. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, pages 2743–2746. IEEE, dec 2011.

[18] Suk Kim Chin and R. Braun. A survey of UDP packet loss characteristics. In *Conference Record of Thirty-Fifth Asilomar Conference on Signals, Systems and Computers (Cat.No.01CH37256)*, pages 200–204 vol.1. IEEE, 2001.

[19] Raimundo Viegas, Ricardo Valentim, Daniel Texeira, and Luiz Guedes. Analysis of Protocols to Ethernet Automation Networks. In *2006 SICE-ICASE International Joint Conference*, pages 4981–4985. IEEE, 2006.

[20] Abhay K. Parekh and Robert G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[21] Peter Baumann, Martin Krammer, Mario Driussi, Lars Mikelsons, Josef Zehetner, Werner Mair, and Dieter Schramm. Using the Distributed Co-Simulation Protocol for a Mixed Real-Virtual Prototype. In *2019 IEEE International Conference on Mechatronics (ICM)*, pages 440–445. IEEE, mar 2019.

[22] Martin Krammer, Nadja Marko, and Martin Benedikt. Interfacing Real-Time Systems for Advanced Co-Simulation - The ACOSAR Approach. In Catherine Dubois, Francesco Parisi-Presicce, Dimitris Kolovos, and Nicholas Matragkas, editors, *STAF 2016 Doctoral Symposium and Projects Showcase*, pages 32–39, Vienna, Austria, 2016. Dubois, Catherine Parisi-Presicce, Francesco Kolovos, Dimitris Matragkas, Nicholas.

[23] Modelisar Consortium and "FMI" Modelica Association Project. Functional Mock-up Interface for Model Exchange and Co-Simulation, Version 2.0, 2014.

[24] Martin Krammer, Martin Benedikt, Torsten Blochwitz, Khaled Alekeish, Nicolas Amringer, Christian Kater, Stefan Materne, Roberto Ruvalcaba, Klaus Schuch, Josef Zehetner, Micha Damm-Norwig, Viktor Schreiber, Natarajan Nagarajan, Isidro Corral, Tommy Sparber, Serge Klein, and Jakob Andert. The Distributed Co-Simulation Protocol for the Integration of Real-Time Systems and Simulation Environments. In *Proceedings of the 50th Computer Simulation Conference*. Society for Modeling and Simulation International (SCS), 2018.

[25] Martin Krammer, Klaus Schuch, Christian Kater, Khaled Alekeish, Torsten Blochwitz, Stefan Materne, Andreas Soppa, and Martin Benedikt. Standardized Integration of Real-Time and Non-Real-Time Systems: The Distributed Co-Simulation Protocol. In *Proceedings of the 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019*, volume 157, pages 87–96, Regensburg, Germany, feb 2019. Modelica Association.

[26] *DCP Specification Document, Version 1.0*. Linköping, Sweden, 2019.

[27] Lifang Wang, Xing She Zhou, Ze Jun Jiang, and Aihua Zhang. A Real-Time Process Scheduling Policy in Windows. In *2012 International Conference on Computer Science and Service System*, pages 22–24. IEEE, aug 2012.

[28] IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7. *Society*, (Jan):1–3951, 2018.

[29] Zhang Yanyan and Ran Xiangjin. Analysis of Linux Kernel's Real-Time Performance. In *2018 International Conference on Smart Grid and Electrical Automation (ICSGEA)*, pages 191–194. IEEE, jun 2018.

[30] Jianfeng He, Yufeng Li, Wei Zhang, Fang Fang, and Hongkun Xu. Real-Time Optimization and Application of the Embedded ARM-Linux Scheduling Policy. In *2011 International Conference of Information Technology, Computer Engineering and Management Sciences*, pages 134–138. IEEE, sep 2011.

[31] Gabriel M. Hoffmann, Claire J. Tomlin, Michael Montemerlo, and Sebastian Thrun. Autonomous Automobile Trajectory Tracking for Off-Road Driving: Controller Design, Experimental Validation and Racing. In *2007 American Control Conference*, pages 2296–2301. IEEE, jul 2007.

[32] H. Nyquist. Certain Topics in Telegraph Transmission Theory. *Transactions of the American Institute of Electrical Engineers*, 47(2):617–644, apr 1928.

[33] William L Oberkampf, Sharon M DeLand, Brian M Rutherford, Kathleen V Diegert, and Kenneth F Alvin. Error and uncertainty in modeling and simulation. *Reliability Engineering & System Safety*, 75(3):333–357, 2002.

[34] Barry Croke. Representing uncertainty in objective functions: extension to include the influence of serial correlation. 2009.

[35] ITU-T. Timing characteristics of a synchronous Ethernet equipment slave clock. page 44, 2018.