Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Simulation of Time-synchronized Networks using IEEE 1588-2008

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Wolfgang Wallner
Matrikelnummer 0725458

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Armin Wasicek

Wien, April 12, 2016

(Unterschrift Verfasser)

(Unterschrift Betreuung)

Simulation of Time-synchronized Networks using IEEE 1588-2008

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Wolfgang Wallner
Registration Number 0725458

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner
Assistance: Univ.Ass. Dipl.-Ing. Dr.techn. Armin Wasicek

Vienna, April 12, 2016

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Wolfgang Wallner
Pischelsdorf 81, 5233 Pischelsdorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)

(Unterschrift Verfasser)

The purpose of computing is insight, not numbers.

Richard Hamming

Acknowledgements

I would like to thank my advisors for this thesis, Prof. Dr. Peter Puschner and Univ.Ass. Dipl.-Ing. Dr.techn. Armin Wasicek.

The focus of this thesis is on the simulation of the Precision Time Protocol (PTP). Thus it is a great honor for me that John C. Eidson, Ph.D., the chair of the PTP standards committee, took the time and read a draft of my thesis. I am particularly grateful for his helpful comments and suggestions.

My long-term friend, former classmate and fellow student, Oliver Hechinger, MSc contributed to this thesis by giving constructive suggestions and technical proofreading.

I am deeply grateful to my parents, for continuously encouraging and supporting me throughout the years.

Abstract

A global time base is the foundation for any distributed Real-Time System (RTS). As clock devices suffer from various noise effects, some kind of time synchronization is necessary to establish a global time base of reasonable precision. Different technologies have been developed to solve this task. The Precision Time Protocol (PTP), which is standardized in [IEE08], presents one candidate that is widely deployed in the various industries. Commercial off-the-shelf (COTS) devices start to have hardware support for PTP. Having hardware timestamping available, PTP promises to be a good compromise between costs and precision. Low-cost, high-precision hardware makes PTP an interesting option for system designers and enables new distributed RTS applications.

When using PTP in an application, it is of interest to justify the expected precision in advance. Especially as there are many options in how PTP can be implemented and configured. The goal of this thesis was to develop a simulation framework for PTP, which helps system designers carrying out Design Space Exploration (DSE).

To reach this goal, this thesis presents two software projects:

- *LibPLN*: a portable, efficient software library to generate realistic oscillator noise
- *LibPTP*: a simulation framework for PTP devices

Both of these components are implemented as generic, portable C++ libraries. This enables users can customize them and to integrate them in their own design tool chains in order to evaluate their hyotheses on the system under consideration.

Kurzfassung

Eine globale Zeitbasis ist die Grundlage für verteilte Echtzeitsyteme. Da einzelne Uhren durch verschiedene Rauscheffekte voneinander abweichen, benötigt man eine Art der Synchronisierung um eine globale Zeitbasis aufzubauen. Verschiedene Technologien wurden entwickelt um diese Aufgabe zu lösen. Mit dem Precision Time Protocol (PTP), welches in [IEE08] standardisiert wurde, steht ein weiterer Kandidat zur Verfügung der in verschiedenen Industrien eingesetzt wird. Da mittlerweile auch normale Verbrauchergeräte anfangen Hardwareunterstützung für PTP zu enthalten, wird das Protokoll mehr und mehr zu einem interessanten Kompromiss zwischen Kosten und Präzision. Die Verfügbarkeit von kostengünstiger, präziser Hardware macht PTP zu einer interessanten Option für Systemdesigner and ermöglicht neue Applikationen verteilter Echtzeitsysteme.

Wenn man PTP in einer Applikation einsetzen möchte, ist es von Interesse die zu erwartende Präzision der Synchronisierung im Vorhinein abschätzen zu können. Im Speziellen da PTP zahlreiche Optionen zur Verfügungen stellt wie es implementiert und konfiguriert werden kann. Das Ziel dieser Arbeit ist es ein Simulations-Framework für PTP zu entwickeln, welches einen Systemdesigner bei diesen Designentscheidungen unterstützt.

Um dieses Ziel zu erreichen, wurden für diese Arbeit zwei Softwareprojekte entwickelt:

- *LibPLN*: eine portable, effiziente Softwarebibliothek zum erstellen von realistischem Oszillatorrauschen
- *LibPTP*: ein Simulations-Framework für PTP-Geräte

Beide Komponenten wurden als generische, portable C++ Bibliotheken implementiert. Das ermöglicht es Anwendern sie für ihre Zwecke anzupassen und in eigenen Entwicklungswerkzeuge zu integrerieren, um damit die jeweils untersuchten Systeme simulieren zu können.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 1
- 1.3 Focus of this Thesis . . . 2
- 1.4 Methodological Approach . . . 3
- 1.5 Related Work . . . 3
- 1.6 Structure of this Thesis . . . 4
- 1.7 Typographic Conventions . . . 4

2 Basic Terms and Concepts **5**
- 2.1 General Terminology . . . 5
- 2.2 IEEE 1588 - Precision Time Protocol . . . 6
- 2.3 Clocks and Clock Noise . . . 16
- 2.4 Simulation Environment OMNeT++ . . . 29

3 Implementation **31**
- 3.1 Efficient PLN generation library . . . 31
- 3.2 OMNeT++ simulation models . . . 50

4 Evaluation **63**
- 4.1 Assumptions . . . 63
- 4.2 Experiments . . . 66
- 4.3 Result discussion . . . 75

5 Conclusion **77**
- 5.1 Contributions of this Work . . . 77
- 5.2 Future Work . . . 78

A Acronyms **79**

Bibliography **81**

CHAPTER 1

Introduction

This chapter will give a short overview over the purpose of this thesis.

1.1 Motivation

The implementation of distributed RTSs requires that the individual nodes of a network establish a global time base of known precision. The Precision Time Protocol (PTP) as specified in [IEE02] and later [IEE08] aims to provide high precision time synchronization in packet switched networks. While PTP could be implemented as a pure software solution, the addition of hardware support improves the reachable precision by several orders of magnitude. Given that hardware support for PTP starts to becomes available in COTS devices, it can be expected that the financial effort to establish a high precision global time base with PTP will decrease in the coming years. This would enable the use of PTP in domains where the economic pressure on individual nodes is high, e.g. in industrial automation where networks often contain large numbers of low performance nodes.

While these properties would make PTP an attractive technology for domains like industrial automation, it is hard to predict how PTP behaves in large networks. As especially the prices for PTP-aware network switches are currently quite high ($\geqslant 1000$ \$), it is not feasible to test the behavior of large PTP networks. Additionally, PTP provides a large range of optional features, and it might not be clear what the best options for a specific implementation are. Therefore it would be useful to have a simulation environment with PTP-aware network nodes.

1.2 Problem Statement

The aim of this thesis is the development and implementation of a simulation framework for PTP networks. This problem can be divided into two main tasks:

- Simulation of oscillator noise
- Simulation of PTP to counter this oscillator noise

The resulting simulation should be able to provide realistic answers to research questions about the influence of different parameters of a PTP network. Examples of parameters of interest include:

- Synchronization interval
- Oscillator quality
- Line length
- PTP clock types (e.g Boundary Clocks (BCs)/Transparent Clocks (TCs))
- Delay mechanism

1.3 Focus of this Thesis

The simulation framework which is developed for this thesis should implement the following features:

- A *realistic* and *efficient* model for oscillators with support for the inherent stochastic noise processes
- A PTP stack with support for the most common IEEE 1588-2008 features, especially
    - All PTP clock types:
        - Ordinary Clock (OC)
        - Boundary Clock (BC)
        - Transparent Clock (TC)
    - Both PTP delay mechanisms:
        - End-to-End (E2E)
        - Peer-to-Peer (P2P)
- Models for PTP transport over Ethernet (as specified in Annex F of [IEE08])
- A servo control for slave clocks based on a proportional-integral (PI) control model

The focus of this thesis is limited, and it only deals with a subsection of the overall problem. Other areas of interest (e.g. temperature influence on oscillators) have been spared out. However, the solutions that where developed for this thesis have been kept flexible, so that the work can be extended and improved in future projects. Such extensions would include:

- external influences on the oscillators (e.g. temperature, pressure, . . .)
- different servo control models
- different lower layers for packet transmission (e.g. User Datagram Protocol (UDP))

The simulation of the PTP specific models is implemented in Objective Modular Network Testbed in C++ (OMNeT++)[1]. While there are several different simulation environments that might be suitable for this task, the reasons for selecting OMNeT++ are the following:

---

[1] `http://www.omnetpp.org/`

- OMNeT++ is simple (easy to learn) yet very expressive.
- It is available with an academic license free of charge.
- It is portable, and available for the usual desktop Operating Systems (OSs).
- The source code is available, which is important for debugging.
- OMNeT++ can be extended with additional models via libraries. One of the available libraries is the *INET* library, which already contains a lot of useful network models.
- There is already existing literature that deals with the simulation of PTP in OMNeT++ (see section 1.5 for details).

1.4 Methodological Approach

The following work items have been carried out for this thesis:

- Literature research on the relevant topics (simulation of oscillator noise, PTP clock servo design, . . .)
- Study of available open-source PTP implementations (PTPd[2], LinuxPTP[3])
- Design and development of a a library for clock noise generation
- Design and development of a PTP simulation framework
- Study of data sheets to base hardware assumptions on realistic values
- Setup of simulated experiments to verify the plausibility of the implemented components
- Analysis of the resulting data

1.5 Related Work

1.5.1 Simulation of PTP

A basic simulation of PTP in OMNeT++ has been described by Steinhauser in [Ste12]. Gaderer et al. have discussed the implementation of PTP and related topics such as oscillator noise in OMNeT++ in several papers (e.g. [PGGS07], [GLN+07], [GNLK08], [RGNL10], [GNLS11]). Another paper dealing with PTP simulation in OMNeT++ is that by Liu and Yang[LY11]. A paper by Depari et al.[DFF+07] deals with the simulation of PTP in another simulation framework.

1.5.2 Simulation of oscillator noise

The simulation of oscillator noise has been an ongoing research topic for decades, and it is well covered in the academic literature. The papers that were most relevant for this thesis are those by Kasdin and Walter[KW92], [Wal94], [Kas95] and those by Gaderer et al (as listed above, especially [GNLS11]).

---

[2] `http://ptpd.sourceforge.net/`

[3] `http://linuxptp.sourceforge.net/`

1.6 Structure of this Thesis

To avoid ambiguity, chapter 2 introduces the basic terms and concepts which will be used later. After that, chapter 3 will describe the developed simulation framework as well as related tools in detail. Chapter 4 will discuss the experiments that were carried out with the simulation framework, and their results. Finally, chapter 5 will give an overview of the contribution of this thesis, and an outlook on possible future extensions.

1.7 Typographic Conventions

Quotations Exact quotes will be emphasized as follows: “Within a domain, an ordinary or boundary clock with a port in the Slave state shall synchronize to its master in the synchronization hierarchy established by the best master clock algorithm.”[IEE08]

Definitions Special *terms* will be emphasized when they are first used.

Source code A source code example[4] is given in Listing 1.1.

```
int getRandomNumber( void )
{
    return 4;    // chosen by fair dice roll.
                 // guaranted to be random.
}
```

Listing 1.1: A sample source code listing which implements a random number generator.

Keywords Keywords like variable names, states, etc. will be formated in a monospace font: The function `getRandomNumber()` returns a truly random number.

Numbers Numbers are per default given in base 10. Hexadecimal numbers will be prefixed with `0x`, binary numbers with `0b`. Exceptions are numerical IDs with a special format, like e.g. Media Access Control (MAC) addresses (these consist of 6 pairs of each 2 hexadecimal digits, separated by colons).

Examples: 1337, `0b101010`, `0xC0FFEE`, 00:DE:AD:BE:EF:00

---

[4] Source code was taken from `http://xkcd.com/221`, which is licensed under CC-BY-NC 2.5

CHAPTER 2

Basic Terms and Concepts

This chapter introduces the terminology and the theoretical concepts that are used in the rest of this thesis. Several definitions are introduced, and an overview is given over the Precision Time Protocol (PTP) and Powerlaw Noise (PLN) processes. The goal is that a reader with a technical background, but who is not an expert in these fields, is able to follow later discussions.

Section 2.1 introduces general terminology. A short introduction to PTP is given in section 2.2. The clock model (especially the noise model) that is used for this thesis is covered in section 2.3. Finally, section 2.4 gives a quick overview over the OMNeT++ simulation environment that is used later.

2.1 General Terminology

The topic of this thesis deals with time synchronization for a global time base, and thus two important fields are Real-Time Systems (RTSs) and Frequency Stability Analysis (FSA). The terms used in this thesis follow the established semantics given in the respective literature of these fields: Terms relevant in the domain of RTSs follow the definitions given by Kopetz [Kop11], terms used in the domain of FSA follow the definitions of IEEE 1139 [IEE09], and terms relevant for PTP follow the definitions of IEEE 1588 [IEE08].

Global time As in [Kop11], the concept of *time* in this thesis does not take into account any relativistic effects. We assume that all nodes of our systems are subject to the same progress of time. Thus, we may refer to this time as the single *global time* (or *real-time*). For theoretical discussions, we might imagine an omniscient external observer who knows the exact time at any instant, and with infinite precision. However, none of the systems that we will model later will be assumed to know the actual value of real-time, but just their local estimation.

Clock As in [Kop11]: “A (digital physical) clock is a device for measuring time. It contains a counter and a physical oscillation mechanism that periodically generates an event to increase the counter.” Only digital clocks are in the focus of this thesis.

Tick, tick length, offset[1] A *tick* or *clock tick* refers to the increment of a clock’s counter. For a perfect clock, these ticks occur at equal time intervals. We will refer to this nominal time interval between two ticks as the *tick length* of a clock, or its *granularity*. The difference of one clock’s counter value relative to another clock’s counter value is referred to as the respective *offset*.

Clock synchronization For real life clocks, the lengths of their ticks may vary over time, and thus the measured progress of time of different clocks is different. Clock synchronization is a general term for efforts to counter this effect. According to [Lai12], the following types of clock synchronization may be distinguished:

- **Frequency synchronization** means clocks running at the same rate. This also referred to as *syntonization*, especially in [IEE08].
- **Phase synchronization** refers to syntonized clocks, where the individual clocks are additionally in phase. A real-life example for this would be multiple people who clap in unison.
- **Time synchronization** Clocks which agree in rate, phase and have the same counter value are referred to as *time synchronized*.

If not otherwise noted, the term *synchronization* will be used in this thesis to refer to time synchronization and the term *syntonization* will be used for frequency synchronization.

Isochronous Events that happen periodically, with exactly the same timely distance to each, other are referred to as being isochronous. As an example, the clock ticks of a perfect clock are considered to be isochronous.

2.2 IEEE 1588 - Precision Time Protocol

This section gives a short introduction into the current revision of the Precision Time Protocol (PTP), which is standardized in IEEE 1588 [IEE08]. The introduction here is simplified, and only covers the topics that are important for later chapters. For a detailed discussion, the reader is referred to the excellent book *Measurement, Control, and Communication Using IEEE 1588* [Eid06] by John Eidson, the initial inventor of PTP. The only drawback of the book is that is was published in 2006 and thus only covers the 2002 revision of the PTP standard, but not the current revision from 2008.

The next revision of PTP is currently in preparation and is expected to be complete in late 2017. It will feature several improvements including specifications for incorporating the White Rabbit[2] technology for sub-nanoseconds synchronization [John Eidson, personal communication, November 2015].

---

[1] These definitions as they are stated here are quite simplified compared to [Kop11], but they convey the same ideas.

[2] `http://www.ohwr.org/projects/white-rabbit`

2.2.1 Overview

PTP is a protocol for network based clock synchronization in the sub-microsecond range, with moderate hardware effort. [Eid06] states its intentions as follows: "IEEE 1588 is designed to fill a niche not well served by either of the two dominant protocols, NTP and GPS. IEEE 1588 is designed for local systems requiring accuracies beyond those attainable using NTP. It is also designed for applications that cannot bear the cost of a GPS receiver at each node, or for which GPS signals are inaccessible."

The Precision Time Protocol (PTP) is specified in the Institute of Electrical and Electronics Engineers (IEEE) standard documents IEEE 1588-2002 [IEE02] and IEEE 1588-2008 [IEE08]. These two versions of the protocol are also referred to as PTPv1 and PTPv2. This thesis does not deal with PTPv1, so whenever the general term PTP is used in this thesis, it refers to PTPv2.

2.2.2 Principle of Operation

PTP provides mechanisms for network nodes to estimate the offset of their local clock to a reference clock in the network via message exchange. This offset estimate can then be used to synchronize the local clock.

A PTP network consists of a number of nodes which are connected in an arbitrary topology. Which clock will synchronize to which other clock is determined by a distributed algorithm, called the *Best Master Clock (BMC)* algorithm: The attributes of the local clock hardware (e.g. accuracy, clock class) are considered to be known to the node (at least via upper bounds), and the network nodes compare their attributes to each other. This information is then used to form a synchronization hierarchy with the best clock on top.

In this hierarchy, neighboring nodes form a *master/slave* relationship: a master sends timing information to its slaves, who will then try to synchronize their clocks to that of the master. A node with connections to multiple neighbors might be a slave to one and a master to another.

Remark: The PTP standard only specifies a method to estimate the offset of a slave clock, and that this offset should be compensated. It does explicitly not specify *how* this should be done: "The specific means for synchronization are out of the scope of this standard but shall result in the minimization of the <offsetFromMaster> value computed by the slave [. . .]" [IEE08]

2.2.3 Offset Estimation Principle

Basically, the offset estimation that is done by PTP consists of two parts:

- Periodic timestamp distribution (via so-called `Sync` messages)
- Delay estimation (how long a `Sync` message needs to travel from the master to the slave)

As an introduction to PTP, we assume a simple example network consisting of only two nodes. Additionally, we assume that the left node is the master of the right node. Figure 2.1 shows such a network.

Remark: PTP network graphs like fig. 2.1 follow a simple notation in this thesis: PTP nodes are shown as large white rectangles, and their individual ports are shown as smaller rectangles. The color of a port symbolizes its current state: green for `MASTER`, blue for `SLAVE` and yellow for `PASSIVE`. Additionally, the links between ports correspond to the relationships of adjacent nodes: a master/slave relationship is shown with an arrow, a relationship with a passive node with a dashed line.

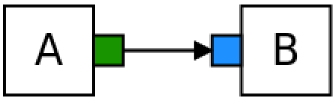


Figure 2.1: Sketch of a simple PTP network consisting of two nodes.

Figure 2.2 shows a possible PTP communication between these two nodes[3]. The master periodically sends `Sync` messages to the slave. In this example we assume that the master is capable of timestamping messages on-the-fly, and thus these messages already contain their own egress timestamps. The slave registers the ingress timestamp when a `Sync` message is received. To calculate its own offset in reference to the master a slave additionally needs to estimate the time that `Sync` messages spend traveling through the network. For this purpose, it periodically sends to the master `Delay_Requ` messages, for which it saves the egress timestamp locally. When the master receives the `Delay_Requ` message, it registers the ingress timestamp and sends it back to the slave in a `Delay_Resp` message. The slave then knows how long its own `Delay_Requ` needed to travel to the master[4]. Both of these communication parts are carried out repeatedly, and the knowledge about these timestamps allows the slave to estimate the offset of its local clock to that of the master.

2.2.4 PTP Concepts and Definitions

When dealing with a PTP related context, the following concepts and terms are assumed:

Clock [IEE08] defined the term *clock* to refer to nodes participating in a PTP network. Unfortunately, we already defined to term clock to refer to a time measuring device. However, this ambiguity should not cause trouble as long as the reader is aware of it, and the correct meaning should be clear from the context. Additionally, the local timing hardware of a node will usually be referred to as the node's *local clock*.

---

[3] The PTP communication shown in the figure could look quite different, depending on the actual PTP configuration. The purpose of this diagram is to show a very simple example.

[4] A basic assumption of PTP (and other two way time transfer protocols like e.g. Network Time Protocol (NTP)) is that the delays on these two paths are symmetric. Asymmetry in the communication paths results in unmeasurable (and thus uncorrected) synchronization offsets, and needs to be properly handled. This is a known challenge, and discussed e.g. in [RMR14].

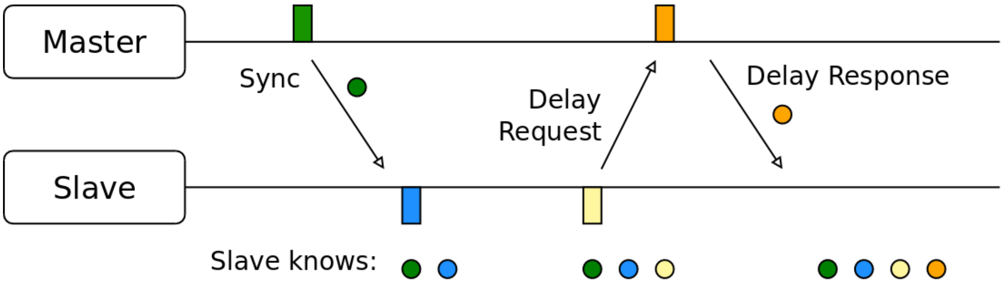


Figure 2.2: Visualization of the offset measurement principle. Rectangles symbolize ingress and egrees time instants, circles symbolize information about such instants.

Port The interfaces via which a PTP clock is connected to the PTP network are called *ports*. Each PTP node has at least one port. A PTP port is a logical access point, thus a clock may provide several ports on a single physical interface.

Port state Each port has a *port state*, which is determined by a standardized state machine. The port's state determines which PTP messages a node will send and receive on this port, and how it will react on them.

Synchronization hierarchy Two connected nodes in a PTP network will either enter a *master*/*slave* relationship or their connection will be disabled. This is handled by the BMC algorithm. The resulting synchronization hierarchy will consist of one (which is the typical case) or more rooted trees.

Timing loop This term describes a network state where the hierarchical synchronization relationships of network nodes contain a loop. The BMC algorithm used by PTP avoids such a network condition.

Grandmaster The top node of a synchronization tree is referred to as the *grandmaster*.

Data sets The local PTP-relevant information of a node is organized in *data sets*. This covers all kinds of information, from static attributes like the clock identity to highly dynamic values like the current time offset to the reference clock. Data sets provide a standardized way to access this information.

Message types The clock offset estimation in PTP is done via message exchange. For this purpose the standard specifies a number of message types. The different types are described in section 2.2.5.

Message classes Message types may belong to one of two classes: they may either be *event messages* or *general messages*. For event messages, it is important *when* they are sent and received, thus they need to be timestamped. General messages do not have this requirement.

Lower layer independence PTP is independent of the lower layer, and can be mapped to various technologies. To be qualified as a lower layer, a technology must meet certain assumptions (like a defined message timestamp point). The PTP standard contains specifications for a number of commonly used protocols. This thesis focuses on PTP over Ethernet, which is specified in *Annex F* of [IEE08].

Message timestamps The foundation of PTP time synchronization is the exchange of timestamped messages. For this purpose, certain messages (event messages) create a timestamp both when they are transmitted (*egress timestamp*) and received (*ingress timestamp*).

One-step/two-step clocks If a clock is capable of inserting the egress timestamp of a message into the message itself while it is being sent, it is referred to as a *one-step clock*. This would be the preferred mode of operation, but it obviously requires specific hardware support (e.g. on-the-fly checksum calculation). If a clock is not one-step capable, it may send the timestamp in another message, called a `Follow_Up` message. Such clocks are referred to as *two-step clocks*.

Residence time When a frame passes through network hardware (e.g. a switch), the time between the frame's ingress and egress is referred to as *residence time*. In case of network devices with multiple output ports, a frame may experience different residence times for each output port. As an example, suppose a 3 port network switch receives a broadcast frame on port 1, and its port 2 already has a multiple frames queued up for sending while its port 3 is idle. The residence time until the frame may leave on the busy port 2 may be quite long, while it could be very short for the idle port 3.

Delay mechanisms For synchronization with PTP, it is necessary to estimate the path delay that a message needs to travel from a master port to a slave port. For this purpose, the PTP standard provides two different methods, which are called *delay measurement mechanisms* (or only *delay mechanisms*). These methods are referred to as End-to-End (E2E) and Peer-to-Peer (P2P) delay mechanisms and are discussed in more detail in section 2.2.9.

This section covers just the PTP terms needed for this thesis, a more complete list is given in chapter 3 (*Definitions, acronyms, and abbreviations*) of [IEE08].

2.2.5 Message Types

PTP nodes carry out different tasks: they build a synchronization hierarchy, measure path delays and distribute synchronization information. For these individual tasks, they use different message types:

Announce messages Messages of type `Announce` are used by PTP nodes to inform their neighbors about their attributes and their current knowledge of the synchronization hierarchy. The information that is contained in these messages is then used to carry out the BMC algorithm (see section 2.2.7 for details). `Announce` messages are general messages, as this information is not time critical.

Synchronization messages A PTP port in `MASTER` state broadcasts its local time periodically to the network using `Sync` messages. The egress and ingress timestamps for `Sync` messages are important, thus they are event messages. Ideally, the egress timestamp of a `Sync` message should be contained in the message itself (one-step clock), which requires special hardware support. If this support is not available, another message is sent containing the egress time information. In contrast to `Sync` messages, these `Follow_Up` messages are not time critical, and are thus general messages.

Delay mechanism messages Depending on the used delay mechanism different types of messages are used (see section 2.2.9 for details). The E2E delay mechanism that was already defined in PTPv1 uses messages of the types `Delay_Requ` and `Delay_Resp` to measure the delay between a slave and its master. PTPv2 introduced the P2P delay mechanism, which measures the delay between neighboring nodes and using the three message types `PDelay_Requ`, `PDelay_Resp` and `PDelay_Resp_Follow_Up`.

Other message types The PTP standard specifies further message types (*Management* and *Signaling*), but these are not important for the discussions in this thesis, and thus won't be described here.

2.2.6 Port States

This section gives a short introduction to the port state machine that is used in PTP. For a full description the reader is referred to chapter 9.2.5 (*State machines*) of [IEE08]. A sketch of the port state machine is shown in fig. 2.3. The standard distinguishes between a *full* and a *slave-only* state machine. The description here covers the full port state machine. The slave-only state machine is basically a subset of what is discussed here (missing the green parts of fig. 2.3).

Remark: During normal operation (after the startup period and in the absence of faults) all ports will be in one of the following three states: `MASTER`, `SLAVE` or `PASSIVE`.

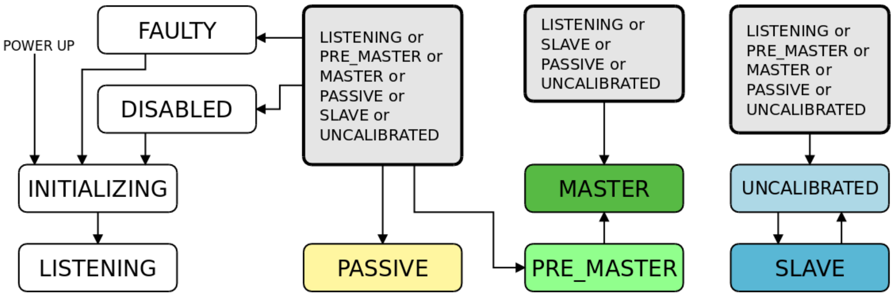


Figure 2.3: Full state machine for the ports of a PTP node. The image is based on Figure 23 in [IEE08], but it is simplified: trivial state transitions as well as transition labels have been left out in this visualization.

`INITIALIZING` This is the first port state after a PTP device is turned on. A port may stay in here as long as it needs to finish its own initialization. When it is done, the only possible follow up state is `LISTENING`.

`LISTENING` A port in this state won't actively participate in synchronization but only listen to the network and wait for the reception of `Announce` messages. According to the received information the node decides how to proceed further. The purpose of this state is to ensure the orderly addition of PTP nodes to an already running network.

FAULTY If a fault occurs, a port moves to this state. Clearing the fault leads again to the state `INITIALIZING`.

DISABLED Explicitly disabling a port leads to this state. After it is re-enabled, it proceeds to `INITIALIZING`.

UNCALIBRATED Whenever a node wants to become a slave of another node, the corresponding port first enters the `UNCALIBRATED` state, and the node starts to synchronize to its chosen master. If the node considers itself synchronized to the master within configured bounds, the port will move on to the `SLAVE` state.

SLAVE In this state a slave is considered to be synchronized to its current master, and it will try to keep this synchronization. If a synchronization error occurs or the slave switches to another master connected to the same port, the port will move back to the `UNCALIBRATED` state again.

PRE_MASTER If a port decides that it is the best suitable master among others, it will enter the `PRE_MASTER` state before changing to `MASTER`. As stated in [IEE08] "an additional mechanism to support more orderly reconfiguration of systems when clocks are added or deleted, clock characteristics change, or connection topology changes is embodied in the `PRE_MASTER` state."

MASTER In this state, a port considers itself the master of the connected network and it periodically broadcasts its locally available timing information. This state is either entered via `PRE_MASTER`, or it is entered directly if the port thinks it is the only available master (this happens if a port does not receive `Announce` messages for a certain amount of time).

PASSIVE A PTP node might decide to not take part in the PTP synchronization on one of its ports by moving it to the `PASSIVE` state. This might be the case for two reasons:

- If a node detects several paths to its grandmaster, it will only use the best path, and deactivate the other ports by moving them to the `PASSIVE` state.
- A node that has access to a external clock synchronization that is assumed to be superior to PTP (e.g. a connected Global Positioning System (GPS) receiver) won't become a slave of another node. If such a node detects better PTP clocks connected to some of its ports, these ports would move to the `PASSIVE` state.

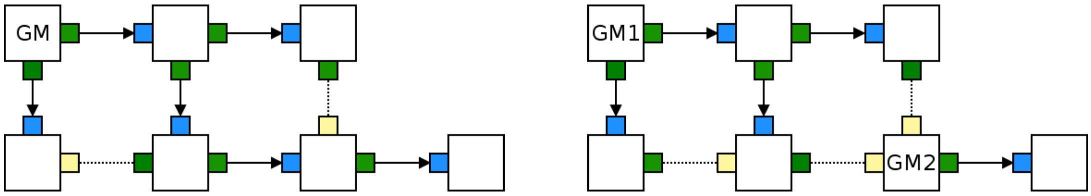
2.2.7 Best Master Clock Algorithm

PTP nodes execute what is called the Best Master Clock (BMC) algorithm to establish and maintain a synchronization hierarchy. A port in `MASTER` state periodically sends `Announce` messages to the network. These messages contain information about the currently known grandmaster, and about the path to the grandmaster. On startup, a node may only know about itself, and assume it is the grandmaster of the system. When a node learns of a better grandmaster, or a better connection to the same grandmaster, it will update its data sets and then broadcast the new information.

The BMC ensures the following properties:

- The decisions that two nodes make will always be *eventually consistent* (there might be short intervals of inconsistency as the state decision events for two neighboring nodes are not synchronous). Thus, it won't happen that two neighboring ports both continue to stay in the `MASTER` state and try to synchronize to each other.
- A node with multiple ports will become a slave on at most one of its ports.
- The synchronization hierarchy will never contain a timing loop.

Two examples are shown in fig. 2.4. Both examples show the same physical network topology, but the nodes are assumed to have different attributes. In fig. 2.4a the node in the upper left has the best clock attributes, and becomes the grandmaster of the remaining nodes. Timing loops in the synchronization hierarchy are avoided by ports that switch to the `PASSIVE` state. The second example assumes that two nodes have excellent attributes (e.g. maybe both of them have a GPS connection), so neither of them will become a slave. Rather, both of them become masters of a small subnet. Connections between these two synchronization networks are removed with passive ports.



(a) Example network with 1 grandmaster (b) Example network with 2 grandmasters

Figure 2.4: Examples for different synchronization hierarchies resulting from the same network, but with different clock attributes.

For a detailed discussion of several interesting cases the reader is referred to section 4.2.3. (*The State Decision Algorithm and Data Set Updates*) of [Eid06].

2.2.8 Clock Types

Depending on the number of ports and the PTP relevant behavior, different types of clocks are distinguished:

Ordinary Clocks (OCs) A PTP device with only one port is referred to as *Ordinary Clock (OC)*. These are the most basic PTP devices, and in a typical PTP network, the endpoints (i.e. the leaves of the synchronization hierarchy) would be of this type.

Application devices, like sensors or actors, would typically be connected to a clock of this type.

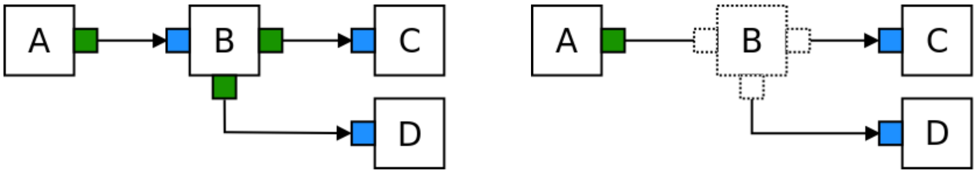
Boundary Clocks (BCs) Multiple OCs could be connected via standard networking hardware (e.g. COTS switches), but this would degrade the precision reachable by PTP: the latencies for queuing frames inside standard network hardware would not be compensated in such a setup, which would lead to both jitter and asymmetry in the path measurement. To counter these effects, the first revision of the PTP standard [IEE02] introduced the concept of *Boundary Clocks (BCs)*: These are PTP clocks with multiple ports, where each port has its own port state. This way, a clock might be a slave on one port, and a master on the remaining nodes. As the name implies, a BC is a boundary for PTP frames: no PTP frame that enters a BC may pass through it. This is in contrast to the concept of a TC which will be explained next. As the ports of a BC act as the end points for any PTP master-slave relationship, the internal frame handling of the switch fabric is not critical for BCs. An example network with a BC is shown in fig. 2.5a.

Transparent Clocks (TCs) The initial concept of BCs showed drawbacks for certain network topologies, especially daisy-chains, as stated in [Eid06]: "Weber and Jaspernite note that in many industrial automation applications, the network will appear as a long linear topology. The current IEEE 1588 protocol requires each node in the link to be a boundary clock, which means that there will be a cascading of control loops and a degradation in the time scale. The proposed solution is to introduce a new type of clock, termed a *Transparent Clock (TC)*, that overcomes this difficulty yet does not affect the operation of the protocol for other nodes in the system." These TCs where then included in the 2008 revision of the PTP standard.

As described in the above citation, a TC interconnects multiple PTP clocks and tries to not affect their communication in any way. This means that for event messages which are supposed to traverse through a TC, the residence time must be corrected. An example network with a TC is shown in fig. 2.5b. The node `A` is the grandmaster of the network, and it synchronizes the slaves `C` and `d`. When `A` sends a `Sync` message, the message is timestamped on the ingress port of `B` (the left port in the picture). The message continues to traverse through the TC, and depending on the current network traffic, leaves the clock at the egress ports to `C` and `D` at possibly different points in time. The individual residence time that the message has spent inside the TC will be corrected at each egress port.

2.2.9 Delay Mechanisms

As already stated, it is necessary to estimate the path delay that a `Sync` message encounters when it travels from a master to a slave. For this purpose, the 2002 revision of PTP introduced the path delay measurement via `Delay_Requ` and `Delay_Resp` messages, which later became known as the End-to-End (E2E) delay mechanism. With the 2008



(a) Node `B` is a Boundary Clock. Node `B` is a slave of `A`, while `C` and `D` are slaves of `B`.

(b) Node `B` is a Transparent Clock. The nodes `C` and `D` are slaves of `A`.

Figure 2.5: Visualization of BCs and TCs. The nodes `A`, `C` and `D` are OCs in both cases.

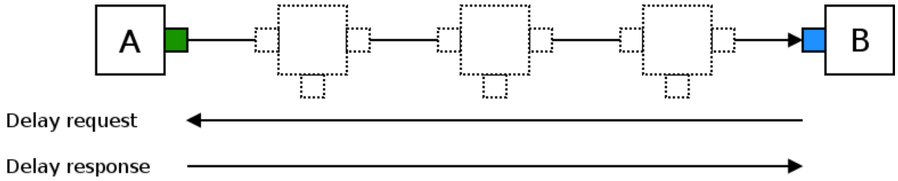
revision of the standard a second approach was added, called the Peer-to-Peer (P2P) delay mechanism.

End-to-End (E2E) When using the E2E delay mechanism, a slave node periodically sends `Delay_Requ` messages to the master, which responds to them with `Delay_Resp` messages. This is what was shown in fig. 2.2. The path delay is estimated using the egress and ingress timestamps of the `Delay_Requ` message. When the two nodes for which the path is measured are interconnected via TCs, then this measurement method measures from one end to the other, hence the name. A sketch of this case is shown in fig. 2.6a. For E2E, the estimated delay is measured for the full path between a slave and its master, and the delay is compensated by the slave. Figure 2.6a shows an example network where a slave measures the delay to its master over a line of TCs. In this example, the full delay that a `Sync` messages suffers when it travels from `A` to `B` is corrected at `B`.

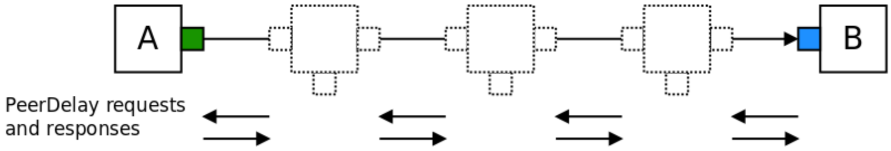
Peer-to-Peer (P2P) With the P2P delay mechanism, the path delay is measured between two neighboring PTP nodes, irrespective of whether they are BCs, TCs, or OCs. This is a notable exception in the behavior of TCs: while they try their best to stay unnoticed for any other traffic, they actively participate in P2P delay measurement. In contrast to E2E delay measurement where the slave node estimates and corrects the full path delay, each node in a P2P delay measurement line estimates the delay for its preceding network link and applies correction for traversing event messages. An example network with TCs and P2P delay measurement is shown in fig. 2.6b. When a `Sync` message from `A` to `B` travels trough the TCs, each TC corrects the path delay for the preceding segment of the path.

Comparison A visual comparison between E2E and P2P is sketched in fig. 2.6. Suppose that in both cases node `A` is the master, while `B` is the slave. In the case of E2E, any slave port measures the full distance to its master. On the other hand, P2P nodes only measure their link to the next neighbor.

This different behavior becomes important especially in large networks. Imagine a network where a single master synchronizes many slaves connected via TCs. Measuring each individual delay from each slave to the master would mean that parts of the network are measured multiple times. This would waste network bandwidth and resources at the master, as it needs to handle each individual delay request.



(a) End-to-End (E2E)



(b) Peer-to-Peer (P2P)

Figure 2.6: Comparison of the two delay mechanism types in the case of a linear topology with TCs.

2.3 Clocks and Clock Noise

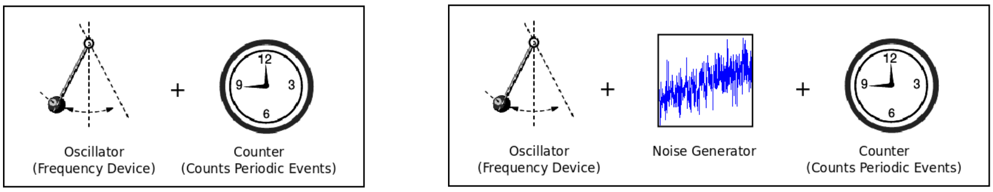
This section gives a short introduction on the attributes of a clock, especially on how they might deviate from the correct value, and how clocks are modeled in this thesis.

2.3.1 Clock Model

As the main goal of this thesis is the simulation of clock synchronization, we need a theoretical model of the clocks that we would like to synchronize. The clock model that is used in this thesis is based on the model described in [AAH00]: on an abstract level, any digital clock may be considered to consist of two parts, an *oscillating device* and a *counter*. The first part (also called the *frequency standard*) is responsible for measuring the length of a given time interval by oscillating at a constant frequency. The second part then has to count the number of oscillations. Figure 2.7a shows a sketch of this clock model. Additionally, we use the following assumptions and definitions for our clock model:

- Increments of the counter value are assumed to be instantaneous. The time instants when these increments happen are referred to as the *ticks* of the clock. In between two ticks the counter value of a clock stays the same.
- The counter value of a clock is an estimate of how many oscillations have occurred since the clock started to count. This value is just an arbitrary integer number, and it is not related to the value of any other clock, or to the global time. We will later add an additional abstraction layer on top that can be used to establish such a relationship through synchronization.

- A node may use its local clock to timestamp events by reading the current counter value. Reading the current counter value is again assumed to be an instantaneous operation. If two events happen between the same two clock ticks, they will be assigned the same timestamp (even if they do not happen at the exact same time instant).
- As the maximum counter value for any real clock has to be finite, it will overflow at some point. However, for any real clock this problem can be solved in a trivial way by providing another abstraction layer to increase the maximum possible counter value. For example, an electronic clock circuit with a 16 bit wide counter value could be easily extended to e.g. 64 bit or more in software. Thus, it is justified for our model to assume that the digital counter of a clock won't overflow at all but just keeps increasing infinitely.
- We will assume that a clock can be used to trigger events on a local node. For this purpose, it is assumed that a node can configure a future timestamp and its clock will notify the node when that value is reached. In a real device, this would correspond to some kind of interrupt mechanism.



(a) Basic model for a digital clock, as given in [AAH00].

(b) Clock model as used in this thesis. Both the oscillator and the counter are assumed to be perfect, and the noise is added by an additional component.

Figure 2.7: Theoretical models for digital clocks. These images are based on an image from [AAH00].

2.3.1.1 Deviation from the Correct Time

For a digital clock to be useful it would be desirable that the clock's ticks occur in an absolute isochronous way. Unfortunately, for any real clock, this is not the case, as stated in [AAH00]: "In principle, if a clock were set perfectly and if its frequency remained perfect, it would keep the correct time indefinitely. In practice, this is impossible for several reasons: the clock cannot be set perfectly; random and systematic variations are intrinsic to any oscillator, and when these random variations are averaged, the result is often not well-behaved; time is a function of position and motion (relativistic effects); and lastly and invariably, environmental changes cause the clock's frequency to vary from ideal."
Thus, we need to specify how such deviations from the correct time are handled in our clock model. For this reason, we will add the following conventions:

- Each of the two parts of our clock model may introduce a deviation from the correct time value. In the context of this thesis, we won't distinguish where such a time deviation is introduced, and just focus on the resulting error. For our theoretical model, we assume that both the oscillator and the counter work absolutely perfect, and introduce a third component in between that is responsible for all noise. This extended model is visualized in fig. 2.7b.
- The reasons for a deviation may be of different kinds, and we don't include all of them in our model:
  - This thesis will especially focus on *random noise processes* that are inherent to quartz-crystal oscillators [AAH00]. For this reason, section 2.3.2.2 will give a detailed discussion on the properties of such noise, and section 3.1 will present the details on how such noise can be simulated.
  - Oscillators may suffer from *systematic deviations* from the correct time. Examples would be aging, frequency drift, or different kinds of external environmental influences (e.g. temperature, pressure, electric fields, ...). While such systematic deviations could be included in our clock model, they are currently not part of it nor of the simulation described in chapter 3. However, a future extension to this work could additionally include these influences. Especially the addition of a temperature model would be worthwhile, as this kind of influence presents a major challenge in real systems.
  - Relativistic effects are not even part of our definition of time, nor of our clock model, and are thus completely ignored in this thesis.

2.3.2 Frequency Stability Analysis

For the later chapters, we need to analyze and compare the frequency stability of oscillators. The discipline that deals with these measures is referred to as *Frequency Stability Analysis (FSA)*. A very good introduction into this field is given in Riley's *Handbook of Frequency Stability Analysis* [Ril08], which describes it as follows: "The objective of a frequency stability analysis is to characterize the phase and frequency fluctuations of a frequency source in the time and frequency domains." This section gives a short overview of the topics that we need later.

2.3.2.1 Definitions and Terminology

IEEE 1139 [IEE09] defines a large set of useful definitions and terminology for the topics of this thesis. A few of them will be used throughout the document, and for the sake of completeness they are repeated here:

Oscillator Output The output of an oscillator can be expressed as

$$V(t) = \big(V_0 + \epsilon(t)\big) * \sin\big(2\pi\nu_0 t + \phi(t)\big) \tag{2.1}$$

where V_0 is the *nominal peak amplitude*
$\epsilon(t)$ is the *deviation from the nominal amplitude*

ν_0 is the *nominal frequency*
$\phi(t)$ is the *phase deviation* from the nominal phase $2\pi\nu_0 t$.

As stated in [Ril08], for the analysis of frequency stability, we are concerned primarily with the $\phi(t)$ term, thus we assume that $\epsilon(t)$ is negligible for the rest of this thesis.

Time Deviation The instantaneous *Time Deviation (TD)* is given by the following definition:

$$x(t) = \frac{\phi(t)}{2\pi\nu_0} \tag{2.2}$$

This value expresses how much the current time estimate of a given oscillator is ahead or behind the nominal error-free case. The TD is measured in seconds. It is also referred to as *time fluctuation* in the literature.

Fractional Frequency Deviation The instantaneous, normalized frequency deviation from the nominal frequency of an oscillator is referred to as the *Fractional Frequency Deviation (FFD)* and is defined as

$$y(t) = \frac{\nu(t) - \nu_0}{\nu_0} = \frac{\dot{\phi}(t)}{2\pi\nu_0} \tag{2.3}$$

The value of $y(t)$ expresses in a normalized way how much a given clock is too slow or too fast at time t. The FFD is related to the TD via the following equations [IEE09, All87]:

$$y(t) = \frac{dx(t)}{dt} \qquad x(t) = \int_0^t y(t')dt' \tag{2.4}$$

The value $y(t)$ is only of theoretical interest, as it is impossible to measure instantaneous frequency [HAB81]. But it can be estimated using the average of $y(t)$ between two measurements of $x(t)$ with a sampling interval of τ:

$$\overline{y}(t) = \frac{x(t+\tau) - x(t)}{\tau} \tag{2.5}$$

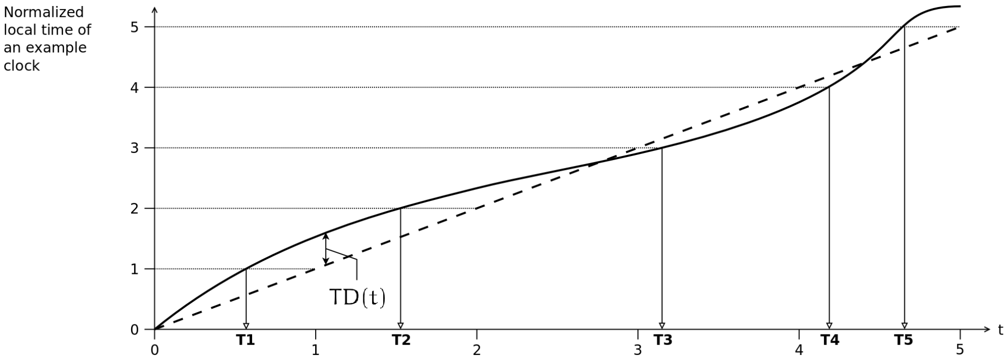
In the literature $y(t)$ is also referred to via several other, slightly different names, e.g. *normalized frequency deviation*, *fractional frequency offset*, *fractional frequency fluctuations*, etc.

Power Spectral Density The one-sided spectral density of the FFD is referred to as $S_y(f)$ [IEE09]. This is an important frequency domain characteristic of frequency stability and is discussed in more detail in section 2.3.2.3.

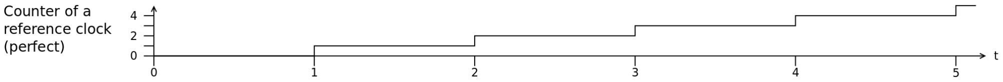
Allan Variance The most important time-domain measurement for clock noise is the so-called Allan Variance (AVAR) [IEE09, Ril08]. A more detailed description is given in section 2.3.2.4.

Figure 2.8 illustrates the definition of TD in combination with our clock model that was introduced in section 2.3.1. For simplicity, the values shown in this diagram are not absolute values but normalized with respect to some nominal tick length. As explained, our clock model consists of three stages: a perfect oscillator, a noise generator, and finally a perfect counter. Suppose we have a look at the output of the different stages of a noisy example clock. If we would measure the progress of time that is given by the perfect oscillator, it would be linear and at all times it would hold that $T_{\mathtt{perfect}}(t) = t$. This is visualized by the straight dashed line in fig. 2.8a. The noise generator stage adds noise to the measured progress of time, and slows it down or speeds it up. The output after this noise generator could look something like the bold line in fig. 2.8a: at first the local time progresses too fast, while later it slows down until it is too slow, just to speed up again. But no matter how much it slows down, it will always continue to rise (i.e. the measured progress of time is strictly increasing). The TD of our example clock would equal the difference between the bold and the dashed line in fig. 2.8a (in this example it is assumed that $\mathrm{TD}(0) = 0$).

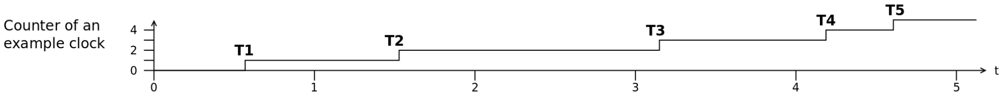
Figure 2.8b and fig. 2.8c show what the counter values look like for a perfect clock (where the noise generation stage is missing) and for our noisy example. For the perfect clock, the counter value looks like fig. 2.8b, where all ticks are exactly isochronous. On the other hand, when the noisy oscillation of our example clock is used to measure time's progress, the value of the counter behaves as shown in fig. 2.8c. The moments when our example clock makes a tick are highlighted in fig. 2.8a as $\mathrm{T1}\,..\,\mathrm{T5}$.



(a) Progress of time as measured by a noisy example clock.



(b) Counter value of a perfect reference digital clock.



(c) Counter value of a noisy digital clock.

Figure 2.8: Visualization of our clock model concepts. All time units are normalized.

2.3.2.2 Powerlaw Noise

The random noise that is intrinsic to common oscillators can be modeled as a stochastic process, and it has some special characteristics. As stated in [Ril08], the random frequency instabilities can be described by the shape of their spectral density: "It has been found that the instability of most frequency sources can be modeled by a combination of Powerlaw Noises (PLNs) having a spectral density of their fractional frequency fluctuations of the form $S_y(f) \propto f^{\alpha}$, where f is the Fourier or sideband frequency in hertz, and α is the power law exponent." This means, that when the Power Spectral Density (PSD) is plotted on a log-log scale, these noise processes will have the shape of a line. While α could have any numerical value in theory, it has been found that for quartz-crystal oscillators five special cases are of importance [AAH00]. In these five cases α has an integer value in the interval $[-2..2]$, and these cases are referred to by the following names:

- $\alpha = 2$: White Phase Modulation (WPM)
- $\alpha = 1$: Flicker Phase Modulation (FPM)
- $\alpha = 0$: White Frequency Modulation (WFM)
- $\alpha = -1$: Flicker Frequency Modulation (FFM)
- $\alpha = -2$: Random Walk (RW)

Figure 2.9 shows an example of what these PLNs look like. Note that different parts of the periodogram are dominated by different noise types. E.g. if they are present, RW and WPM dominate the low and high frequency spectrum below and above certain points. This is why the combined PSD diagram of an oscillator typically has a V or U shape.

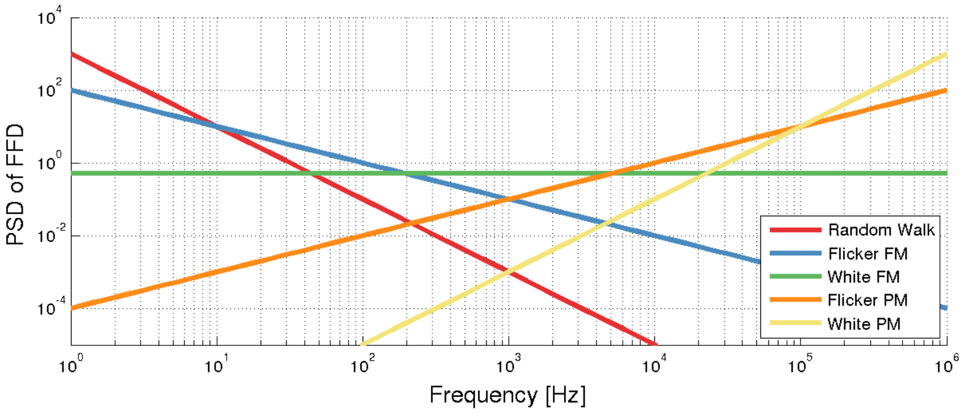


Figure 2.9: Example sketch showing what different kinds of PLNs look like. On a log-log scale, PLNs have the shape of a line.

The TD that an oscillator experiences is a combination of these individual noise processes. A measurement of such a TD might look like the example shown in fig. 2.10. This figure again shows how different types of noise dominate different characteristics of an oscillator's behavior: For example, while the long term behavior is determined by low frequency RW noise, the short term behavior is largely influenced by high frequency noise, like WPM or FPM. PLN can be analyzed in both the time domain and the frequency domain, and the following sections will explain the basic concepts for each.

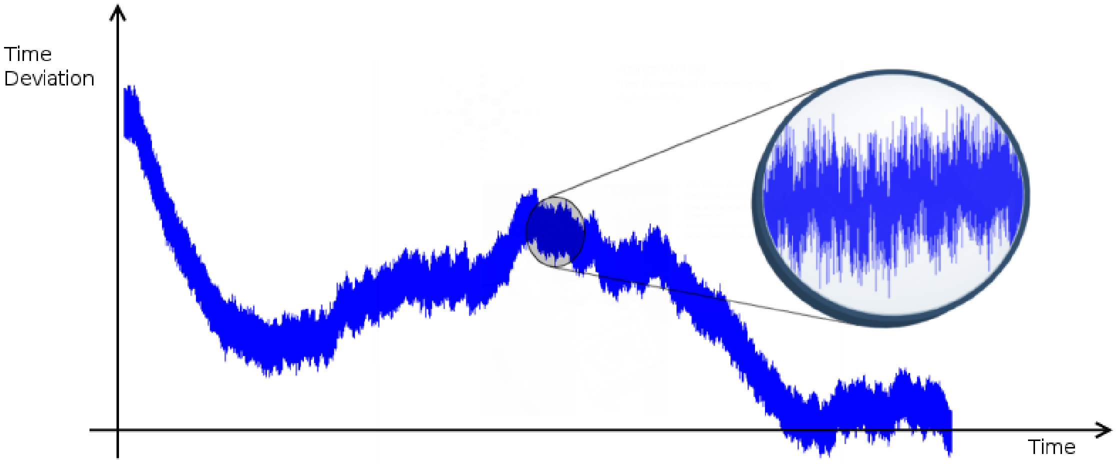


Figure 2.10: Example sketch showing the TD for a combination of PLN types.

2.3.2.3 Frequency Domain Analysis

In the frequency domain, we use the measure $S_y(f)$, which [IEE09] specifies as the "one-sided spectral density of the normalized frequency fluctuations". For PLN, $S_y(f)$ can be assumed to have the form:

$$S_y(f) = \begin{cases} \sum_{\alpha=-2}^{+2} h_\alpha f^\alpha & 0 < f < f_h \\ 0 & f \geqslant f_h \end{cases} \tag{2.6}$$

where		
	$S_y(f)$	is the one-sided PSD of the FFD $y(t)$
	f	is the Fourier or sideband frequency
	h_α	is the intensity coefficient
	α	is the exponent of the PLN process
	f_h	is the high-frequency cutoff of an infinitely sharp low-pass filter

Remark: Some publications (e.g. [Kas95]) assume $S_y(f) \propto 1/f^\alpha$ instead of $S_y(f) \propto f^\alpha$. While these two conventions only differ in the sign of the exponent and are thus easy to convert to each other, care must be taken when combining formulas from different literature sources. Any formulas from other publications that are used in this thesis were converted to fit the conventions used in [IEE09].

There are also other frequency domain measures besides $S_y(f)$, e.g. the PSDs of the TD $x(t)$ ($S_x(f)$) or the phase $\phi(t)$ ($S_\phi(f)$). However, we won't use them in this thesis, so they won't be discussed any further.

2.3.2.4 Time Domain Analysis

Several time domain measures are used for FSA, and a good overview of them is given in [Ril08]. The most important measure is the so-called Allan Variance (AVAR) (symbolized as $\sigma_y^2(\tau)$), which was introduced by David W. Allan in 1966 [All66]. The following paragraphs give an introduction to the AVAR and how to use it to analyze PLN.

Motivation As PLNs are stochastic processes, one might assume that in the time domain the standard deviation (or the standard variance) would be an adequate measure to describe them. However, it turned out that this is not the case for a number of reasons. The most important one is that it does not converge for some kinds of PLN: "One can show that the standard deviation is a function of the number of data points in the set; it is also a function of the dead time and of the measurement system bandwidth. For example, using flicker noise frequency modulation as a model, as the number of data points increases, the standard deviation monotonically increases without limit." [HAB81]

To overcome the problems with standard variance, Allan introduced in [All66] a better suited two-sample variance, which became to be known as the Allan Variance (AVAR). The AVAR has since then become the standard time domain measure for analyzing PLN noise [IEE09].

Allan Deviation (ADEV) The Allan Deviation (ADEV) is the square root of the AVAR. Both measures are commonly found in the literature. As they are strictly related and imply each other, it does not really matter which one of them is used to describe PLN. Throughout this thesis, both of them are used, depending on which better fits the context (e.g. for comparisons with figures from other literature).

Interpretation The AVAR is a measure for the frequency stability of an oscillator over intervals of a specified length τ. Usually it is not calculated for a single interval length, but for a number of different interval lengths. These values are then visualized as a line graph. An example is shown in fig. 2.11. In this example it can be seen that the given oscillator is more unstable in the milliseconds range than in the seconds range. On the other hand, after the seconds range the instability increases again for longer interval lengths. As we will see later, the shape of this curve is related to the PSD shape of the noise processes that are present.

Definition and Calculation [All87] defines the AVAR $\sigma_y^2(\tau)$ as follows:

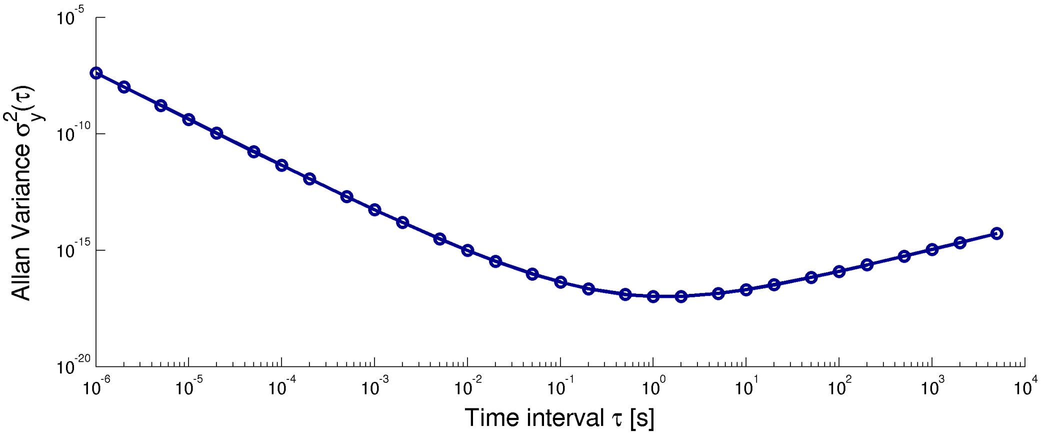


Figure 2.11: Example AVAR plot for interval lengths τ in the range from microseconds to 10^3 seconds. The dots represent the estimated AVAR for those values of τ.

$$\sigma_y^2(\tau) = \frac{1}{2}\left\langle \left(\triangle\, \bar{y}^{\tau} \right)^2 \right\rangle, \tag{2.7}$$

"where $\triangle\ \bar{y}^{\tau}$ is the difference between adjacent fractional frequency measurements, each sampled over an interval τ, and the brackets $\langle\ \rangle$ indicate an infinite time average or expectation value." As it is not possible to sum over infinitely many measurements in practice, the value of σ_y^2 can only be estimated. For a dataset of M FFD values, one can estimate the AVAR using the formula given in [IEE09]:

$$\sigma_y^2(\tau) \cong \frac{1}{2(M-1)} \sum_{k=1}^{M-1} \left[\bar{y}_{k+1} - \bar{y}_k\right]^2 \tag{2.8}$$

Its also possible to express the AVAR estimate in terms of $M+1$ TD values:

$$\sigma_y^2(\tau) \cong \frac{1}{2(M-1)\tau^2} \sum_{k=1}^{M-1} [x_{k+2} - 2x_{k+1} + x_k]^2, \tag{2.9}$$

Equation (2.9) leads to same result as eq. (2.8), but it might be more efficient: usually the result of a measurement is data in the form of TD values, and it would be more computational effort to apply eq. (2.5) on the TD data and then estimate the AVAR via eq. (2.8) than to directly apply eq. (2.9).

Usually, the AVAR values for different interval lengths are computed from the same set of input data. In this case the measurements are sampled at a frequency corresponding to the smallest interval length of interest. AVAR calculations for longer intervals are then done with the same input data, where intermediate values are left out. E.g. if the input data was originally sampled with 1 kHz ($\tau_0 = 1$ ms), the AVAR for $\tau = 2$ ms could be estimated by using every second measurement ($\tau = 2 * \tau_0$). As longer intervals are estimated with a smaller set of input values, this will of course decrease the confidence of the estimate.

Confidence intervals The AVAR is defined as an infinitely large time average, but in practice we estimate its value form a finite amount of data. It is a natural question to ask for the confidence of such an approximation. [All87] states that "a data set of the order of 100 points is more than adequate for convergence of $\sigma_y(\tau)$, though of course the confidence of the estimate will typically improve as the data length increases." To estimate the confidence intervals for a given AVAR result, [IEE09] describes two different approaches:

- A rather simple approach would be to assume a symmetric Gaussian distribution and to estimate a 68% confidence interval $\sigma_y(\tau) \pm I$ for a data set of length M with the relation

  $$I = K_\alpha \cdot \sigma_y(\tau)/\sqrt{M} \tag{2.10}$$

  K_α depends on the type of the PLN. For the five typical α values K_α is in the range of $0.75 - 0.99$, thus slightly looser bounds can be estimated by assuming $K_\alpha = 1$:

  $$I = \sigma_y(\tau)/\sqrt{M} \tag{2.11}$$

- A more sophisticated method would be to assume a χ^2-distribution and to calculate the equivalent number of degrees of freedom depending on the type of noise. While this approach is more complex compared to the first one, it results in tighter bounds for the AVAR estimation. However, this approach is not used in this thesis, and thus the reader is referred to [IEE09, Ril08] for a more detailed discussion.

Drawbacks A major drawback of the AVAR is that WPM and FPM both result in a similar AVAR plot. While the other types of noise can be easily identified by looking at the AVAR, this is not possible for these to types of PLN. This discussed in more detail in section 2.3.2.5.

Improvements and Alternatives Since the introduction of the original AVAR in 1966 a large number of improvements have been made in the field of FSA. This includes both improvements to the AVAR itself (e.g. using overlapping samples) and the invention of new variances (e.g. Total Variance, Thêo Variance, . . .). Again, for a detailed overview the reader is referred to [Ril08]. An important improvement to the classic AVAR that should also be mentioned here is the Modified Allan Variance (MAVAR). In contrast to the AVAR, the MAVAR is able to distinguish between to between WPM and FPM noise. As the classic AVAR is still widely used in the literature, it is used as the basic time domain measure in this thesis, even though other measures (like the MAVAR) would have interesting advantages.

2.3.2.5 Powerlaw Noise Relationship between Time and Frequency Domain

The introduced time and frequency domain measures are related to each other. Remember that we introduced PLN as noise with a PSD shape of $S_y(f) \propto f^{\alpha}$. The dominance of PLN with an exponent of α in the time domain measure $S_y(f)$ leads to an AVAR shape of the corresponding time interval range of τ^{μ}, where μ directly depends on α. Unfortunately, the mapping from α to μ is not bijective[5], as both WPM ($\alpha = -2$) and FPM ($\alpha = -1$) noise lead to an AVAR shape of $\mu = -1$. The basic relationship between $S_y(f)$ and the AVAR is sketched in fig. 2.12.

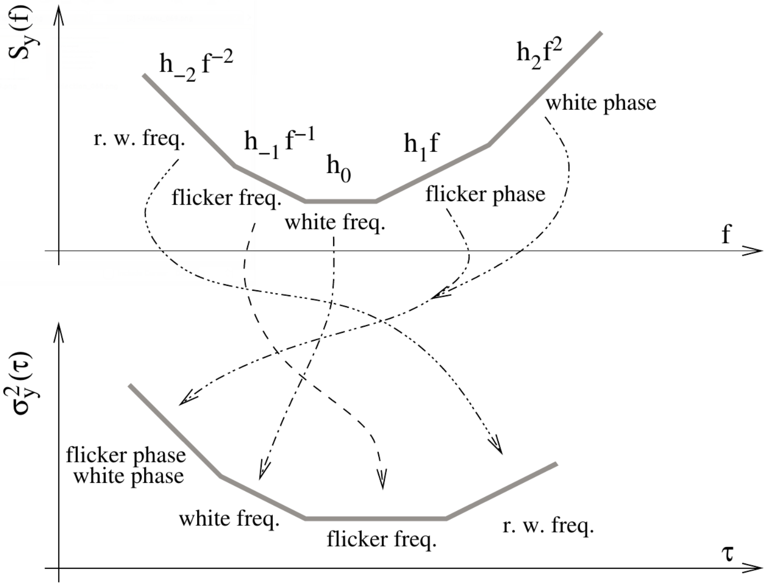


Figure 2.12: Relationship of AVAR and S_y. This plot is based on an image from [Rub05].

Table 2.1 gives a detailed overview of the S_y/AVAR relationship. Notice that the factors D and E for FPM and WPM noise are not constants, but depend on f_h and τ (f_h is the high frequency cutoff frequency as given in eq. (2.6)). This is important for later sections when we want to generate PLN.

---

[5] In the case of the MAVAR the relationship would be bijective.

Table 2.1: Relationship between σ_y^2 and S_y. Source: [IEE09], Table B.2

Noise process	$S_y(f) =$	$\sigma_y^2(\tau) =$
Random Walk	$h_{-2} \cdot f^{-2}$	$A \cdot h_{-2} \cdot \tau^1$
Flicker Frequency Modulation	$h_{-1} \cdot f^{-1}$	$B \cdot h_{-1} \cdot \tau^0$
White Frequency Modulation	$h_0 \cdot f^0$	$C \cdot h_0 \cdot \tau^{-1}$
Flicker Phase Modulation	$h_1 \cdot f^1$	$D \cdot h_1 \cdot \tau^{-2}$
White Phase Modulation	$h_2 \cdot f^2$	$E \cdot h_2 \cdot \tau^{-2}$

where A, B, C, D and E have the following values:

$$\begin{aligned} A &= \frac{2\pi^2}{3} \qquad B = 2 \cdot \ln 2 \qquad C = 1/2 \\ D &= \frac{1.038 + 3 \cdot \ln(2\pi \cdot f_h \cdot \tau)}{4\pi^2} \quad E = \frac{3 \cdot f_h}{4\pi^2} \end{aligned} \tag{2.12}$$

2.3.2.6 Tools for Frequency Stability Analysis

To analyze the frequency stability of measured or simulated data, it is convenient to have software tools. For the work on this thesis, various Matlab scripts have proven to be useful:

- To convert between x and y data, eq. (2.5) and its inverse can be implemented in a straight forward way using the Matlab internal functions `diff` and `cumsum`.
- In the frequency domain, $S_y(f)$, the PSD of y, is of interest. A common method for PSD estimation is *Welch's method*, which is supported in Matlab via the `pwelch` function. Using this method, the raw input data gets split into shorter intervals, to which a window function is applied, and from which then the PSD is estimated. The individual PSD estimates for the shorter segments are then averaged, to get an estimate for the initial data. Figure 2.13 shows the effect of this averaging procedure. It can be seen that for a larger number of averages the estimated PSD is less noisy, but on the other hand the information about lower frequencies is lost. The PSD visualizations in this thesis were made by using a *Hanning* window as the window function. This choice was made as it is recommended in several publications (e.g. [Sch12]) as a good starting point.
- For FSA in the time domain, we are interested in the AVAR. Matlab does not provide packages for AVAR calculation, but support for this measure is available using third-party scripts:
  - The function `allan`[6] can be used to calculate the ADEV for a set of τ values from a given input vector. The input data can be either FFD or TD values. The script accepts either equally spaced samples with a specified sample rate,

---

[6] `http://de.mathworks.com/matlabcentral/fileexchange/13246-allan`

or arbitrary spaced samples with additional timestamp data. Support for the estimation of error bars is implemented using the approach of eq. (2.11).

- The script `Stability Analyzer 53230A`[7] provides a Graphical User Interface (GUI) to interact with a universal counter from Agilent. Its purpose is to read data from a counter and provide various FSA measures, including the ADEV. The script also supports an operational mode where no counter is necessary and the data can be read from a Comma-separated values (CSV) file. The data in the CSV file is assumed to be absolute frequency values between equally spaced sampling points. This tool also provides support for error bars, but in contrast to the other script it performs noise identification and then uses the slightly more detailed formula that was given in eq. (2.10).

The `allan` function provides a convenient command line interface for ADEV calculation, thus it was the main tool for ADEV plot generation used for this thesis. The main purpose for which `StabilityAnalyzer` program was used was to provide a counter-check for the `allan` routine. These two tools were developed by different authors, and they expect their input in different formats, however, their results agree for any input data tested by the thesis author. Thus, it seems justified to expect both of them to work as expected.

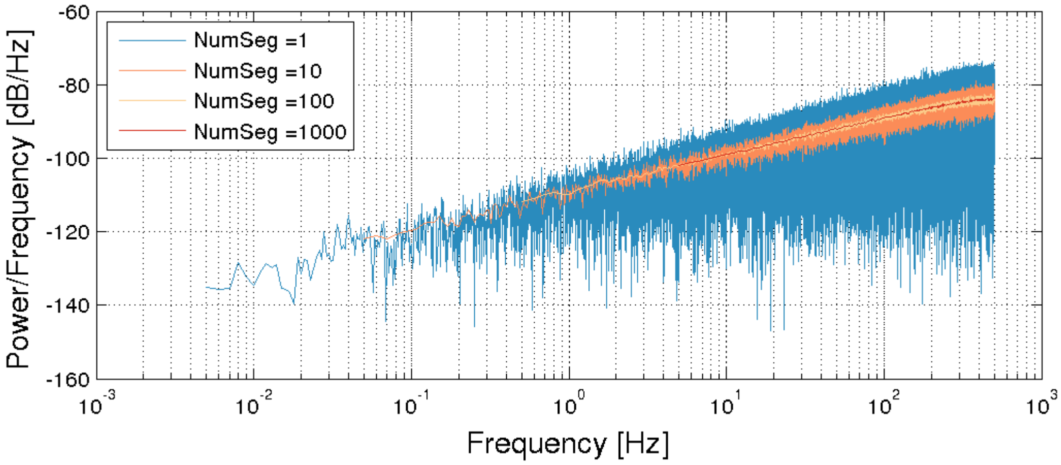


Figure 2.13: Estimates of $S_y(f)$ using Welch's method for different numbers of segments. `NumSeg` gives the number of segments into which the input data was split. The noise sample used for this plot was 1 kHz FPM noise.

---

[7] `http://de.mathworks.com/matlabcentral/fileexchange/31319-stability-analyzer-53230a`

2.4 Simulation Environment OMNeT++

As the goal of this thesis is the simulation of a network protocol, a suitable simulation environment had to be found. The choice was made for OMNeT++ as it is easy to use, well documented, free for academic use and already used in other publications dealing with PTP simulation (e.g. [Ste12], [GNLS11]). As in all sections of this chapter, the information here is only a short overview. A detailed introduction is provided by the excellent user manual [Ope14].

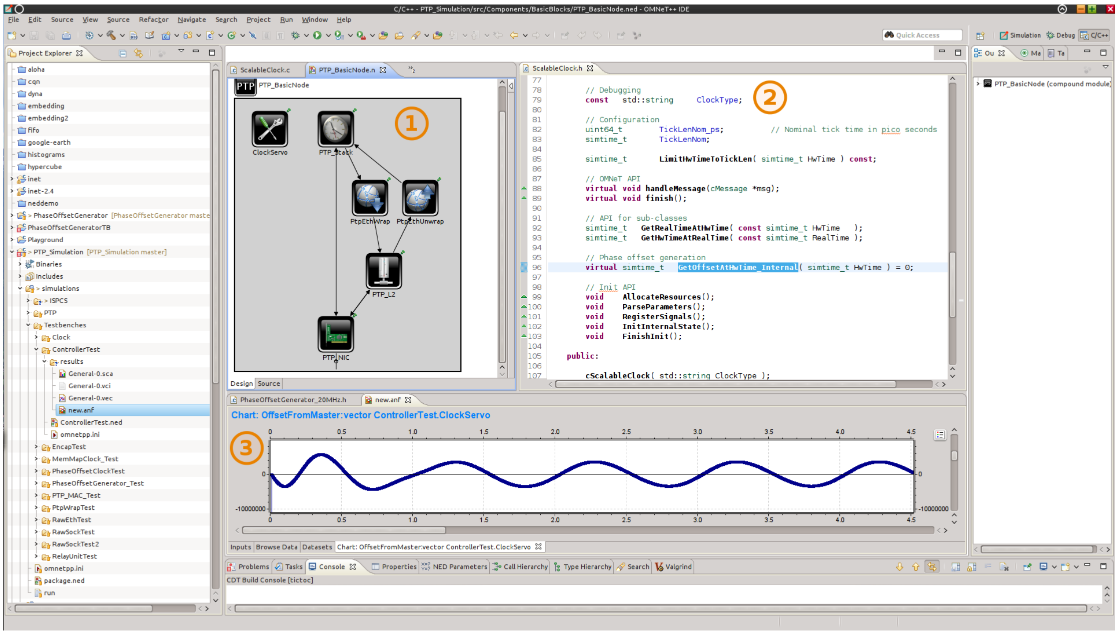
Functional description OMNeT++ is a simulation library and framework for network simulation. It has a generic architecture, and acts as a platform on which a user may build custom simulations. When building a custom simulation, the typical workflow includes:

- Specification of components: OMNeT++ provides a very generic way of how components can be specified, connected and re-used.
- Simulation specification: Once all needed components are implemented, they may be connected to form simulation networks.
- Simulation configuration: It might be of interest to run the same simulation with several different configurations to study the influence of individual parameters. OMNeT++ provides a convenient configuration interface for simulations, including the variation of sets of input parameters.
- Simulation execution: For debug simulation runs, OMNeT++ provides a convenient GUI, while the final simulation runs are executed on the command line to maximize performance.
- Result analysis: The collected simulation data can be analyzed in the GUI, or exported in various formats for other tools.

Basic simulation model An important aspect of OMNeT++ is the fact that it is based on the Discrete Event Simulation (DES) concept. This means that the simulation model is based on a list of ordered events. Any calculation at such an event is assumed to happen instantaneously, and may add more events to the list of future events. No state change can happen between two consecutive events, and these time intervals are simply skipped. This provides the huge advantage that the simulation speed scales inversely to the number of events per simulated time. It is important that the simulation model supports such a simulation concept, as otherwise the speed advantage of DES would be lost. The design of the PLN simulation library that is introduced in section 3.1 was mainly driven by these considerations.

Simulation architecture The building blocks for a simulation are modules which can be connected in a hierarchical order to form more complex modules. The interface of a module is specified in an OMNeT++-specific markup language, which is called the Network Description (NED) language. If a module does not contain any submodule, it is called a *Simple Module (SM)*, and its behavior has to be encoded in C++. On the other hand, if a module consists of one or more submodules, its behavior is fully specified by the behavior of its submodules. Modules of this kind are called *Compound Modules (CMs)*. Any module may have gates which can be connected to other modules via channels, and the individual modules can then communicate by exchanging messages.

Integrated Development Environment (IDE) OMNeT++ also provides an Eclipse-based IDE, which is compatible with the common desktop platforms (Windows, Linux, Mac OS X). A screenshot of the OMNeT++ IDE is shown in fig. 2.14. The IDE provides support for all simulation relevant steps from model specification to result visualization.


C/C++ - PTP_Simulation/src/Components/BasicBlocks/PTP_BasicNode.ned - OMNeT++ IDE
Project Explorer
PTP_BasicNode
ClockServo
PTP_Stack
PtpEthWrap
PtpEthUnwrap
PTP_L2
PTP_NIC
①
ScalableClock.h
②
Chart: OffsetFromMaster:vector ControllerTest.ClockServo
③
0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
-10000000
PTP_BasicNode (compound module)

```
    // Debugging
    const   std::string     ClockType;

    // Configuration
    uint64_t        TickLenNom_ps;          // Nominal tick time in pico seconds
    simtime_t       TickLenNom;

    simtime_t       LimitHwTimeToTickLen( simtime_t HwTime ) const;

    // OMNeT API
    virtual void handleMessage(cMessage *msg);
    virtual void finish();

    // API for sub-classes
    simtime_t   GetRealTimeAtHwTime( const simtime_t HwTime   );
    simtime_t   GetHwTimeAtRealTime( const simtime_t RealTime );

    // Phase offset generation
    virtual simtime_t   GetOffsetAtHwTime_Internal( simtime_t HwTime ) = 0;

    // Init API
    void    AllocateResources();
    void    ParseParameters();
    void    RegisterSignals();
    void    InitInternalState();
    void    FinishInit();

public:

    cScalableClock( std::string ClockType );
```

Figure 2.14: Screenshot of OMNeT++ showing a compound model of a node (①), a source code window (②), and some simulation results (③).

Libraries Additionally, many libraries with predefined OMNeT++ components are available. These components can be customized or extended to fit the purpose for a specific simulation. A popular example is the INET library[8], which contains models for common network components (e.g. switches, network stacks, hardware components, ...). It has been extensively used for the development of the PTP simulations in this thesis.

---

[8] http://inet.omnetpp.org/

CHAPTER 3

Implementation

This chapter will describe the software projects that were implemented as part of this thesis. Section 3.1 describes *LibPLN*, a library for PLN simulation. In section 3.2 we introduce *LibPTP*, our simulation framework for PTP.

3.1 Efficient PLN generation library

To simulate oscillators, we need to simulate PLN. Remember that our clock model (section 2.3.1) consists of three parts:

- A perfect oscillator
- A noise generator
- A perfect counter

Assuming that they are perfect makes the implementation of the first and third part in OMNeT++ straight forward (this will be discussed later in section 3.2.3.1). What is more difficult to implement is the second part (the noise generator): it is responsible for the addition of PLN to the overall output. The library described in this section will implement this second part of the clock model.

3.1.1 Implementation goals

We aim for the following attributes of our implementation:

- **Realistic:** The stochastic characteristics of the produced PLN should match that of a given real oscillator. Of special interest for us are the PSD and the AVAR.
- **Efficient:** As stated in section 2.4, OMNeT++ simulations are based on the DES concept. Thus, the needed simulation effort is directly proportional to the number of events per simulated time, but it does not depend on the amount of simulated time. To take advantage of this concept, our PLN implementation needs to provide

a similar scaling, and should depend as less as possible on the amount of simulated time.

- **Flexible:** While PLN generation is intended to be used as a component in an OMNeT++ model, it turned out that it is also very valuable to have the same functionality implemented as an independent command line tool. Thus the implementation should be done as a library without any OMNeT++ dependencies, so that the same code base can be used for different purposes.
- **Portable:** OMNeT++ supports the common desktop platforms, thus any simulation implemented on it would be quite portable without any additional effort. This is a huge advantage for our PTP simulation and it should not be limited by the introduction of any platform dependencies in the PLN library.

3.1.2 Overview

The implementation presented here is heavily based on a publication by Kasdin and Walter [KW92]. In their publication they discuss different simulation approaches for PLN and their drawbacks. And they also propose another method, which they claim to meet the needed criteria to generate realistic PLN. However, their method is a batch method, and it produces PLN for a number of equally spaced sampling points. As such it is not suitable for a DES environment, because it violates our efficiency criteria to not directly depend on simulation time.

This challenge has already been discussed by other authors, e.g. by Gaderer et al. in [GNLS11]. While they claim that they have improved the Kasdin/Walter PLN generation method to fit the requirements of DES in [GNLS11], they unfortunately only provide a coarse high level description of their modifications and not the required details for reproducing their results.

The goal of this section is therefore to improve the simulation approach of Kasdin and Walter to fit the needs of DES similarly as it is done in e.g. [GNLS11], and to give a detailed description of the underlying theory and the actual implementation.

3.1.3 Generating Powerlaw Noise as proposed by Kasdin and Walter

The original PLN generation method proposed by Kasdin and Walter was introduced in [KW92] and is further discussed in [Wal94] and [Kas95]. Its goal is to simulate an array of equally spaced discrete FFD values, which have a PSD slope proportional to f^{α}. The principle operation of this method is to filter bandlimited white noise through a specially crafted Finite Impulse Response (FIR) filter, so that the PSD gets the desired shape. In detail, the algorithm consists of the following steps to simulate a PLN sample of length N:

- An array of length N is initialized with bandlimited white noise. The original description does not make any assumptions about the distribution of the input noise, however their example implementation uses Gaussian white noise. The amplitude of the resulting noise signal depends on Q^d, the variance of the input white noise. To generate PLN with $S_y(f) = h_\alpha \cdot f^\alpha$ and a simulated sampling interval of τ_0, the white noise variance can be selected as[1]

$$Q^d = \frac{h_\alpha}{2(2\pi)^\alpha \tau_0^{\alpha+1}} \tag{3.1}$$

- An FIR filter of length N is initialized using the following recurrence:

$$\begin{aligned} h_0 &= 1 \\ h_k &= h_{k-1} \frac{\left(k - 1 - \frac{\alpha}{2}\right)}{k} \end{aligned} \tag{3.2}$$

- To apply the FIR filter on the white input noise, the filter coefficients need to be convoluted with the noise data. To save computational effort, the convolution is replaced by a multiplication in the frequency domain using the Fast Fourier transform (FFT) method.

Remark: The authors also provide an example implementation of this method in their original publications [KW92, Kas95], which uses single-precision `float` variables as the basic data type and is based on mathematical algorithms from [PTVF92]. While this implementation works fine for small samples, it shows numerical insufficiencies when it is used to generate larger samples, especially for RW noise. However, reimplementing the same approach with double-precision variables and more modern libraries for the numerical algorithms provides numerically stable results.

Numerical example To demonstrate the basic relationships between the PLN generation formulas given in [KW92] and the FSA formulas given in [IEE09] they will be used in a short numerical example. The purpose of this exercise is to provide the reader with an overview of how the individual formulas are related to each other.

- Consider an oscillator that is affected by FPM noise. This is the case $\alpha = 1$, i.e. $S_y(f)$ increases linearly with the frequency. According to table 2.1, the expected AVAR for this type of noise will be proportional to τ^{-2}.
- We would like to generate noise that was sampled with a frequency of $f_s = 1\,\text{kHz}$. This implies that the interval between two consecutive samples is $\tau_0 = 1\,\text{ms}$. The Nyquist theorem tells us that the highest frequency in the resulting data can be at most $f_h = f_s/2 = 500\,\text{Hz}$.
- Additionally, suppose we would like the PSD plot of the FFD ($S_y(f)$) to have a value of $-23\,\text{dB}$ at $f = 100\,\text{Hz}$.

To generate noise with these attributes, we have to perform the following steps:

---

[1] The sign of the $+1$ in the exponent of τ_0 has been inverted here, as it is assumed to be an error in the original publication [KW92]. This modified version of the formula is then consistent with formulas from [Wal94] and [Kas95].

- We know from eq. (2.6) that $S_y(f) = h_\alpha \cdot f^\alpha$, which for our specific case means that $S_y(f) = h_1 \cdot f^1 = h_1 \cdot f$. The specified value of -23 dB for $S_y(f)$ at $f = 100$ Hz implies that $h_1 \approx 5.0119 \cdot 10^{-5}$.
- Using the value of h_1 together with eq. (3.1) lets us calculate the input variance for the Gaussian white noise: $Q^d \approx 3.9883$.

We can now generate Gaussian white noise with the calculated variance, and apply an FIR filter with coefficients as given by eq. (3.2). Analyzing the resulting PLN in the frequency domain shows a plot as given fig. 3.1. We see that the estimated PSD indeed has the desired shape, and that it passes through the selected point of -23 dB at $f = 100$ Hz.

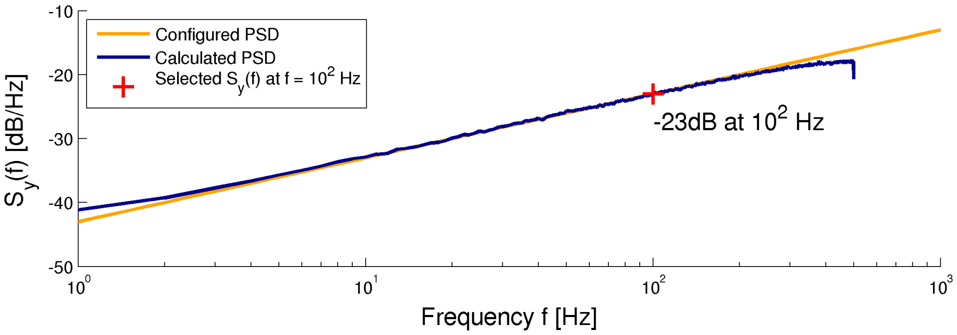


Figure 3.1: PSD plot of our numerical example.

Of course, we might also want to analyze our generated noise sample in the time domain. Table 2.1 gives us a formula for the AVAR of FPM noise: $\sigma_y^2(\tau) = D \cdot h_1 \cdot \tau^{-2}$. We already know the value of h_1, so what we need to do is to calculate the term D. The value of D depends on the highest frequency present ($f_h = 500$ Hz), and the value of τ for which we would like to calculate the AVAR. Suppose we would like to predict the AVAR value at $\tau = 26$ ms. Using the formulas given by table 2.1 we can estimate that $\sigma_y^2(\tau = 26\,\text{ms}) \approx 26.75 * 10^{-3}$. The actual AVAR results together with our estimation are shown in fig. 3.2. As we can see, the estimation fits quite well with the actual result.

The method used in this short example to create PLN with specified AVAR and PSD characteristics will be the basis for the approach in the following sections.

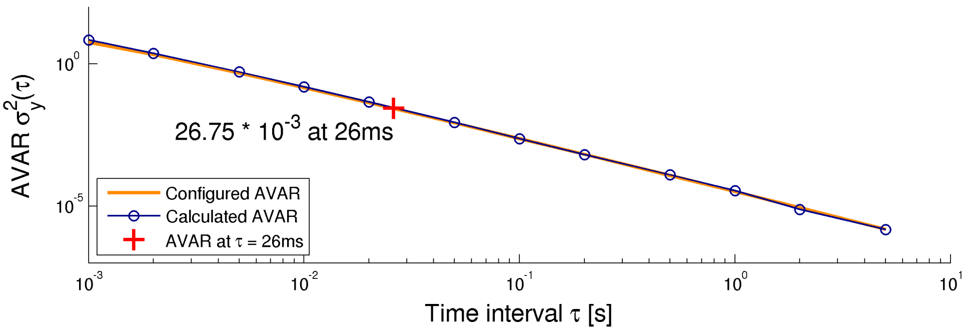


Figure 3.2: AVAR plot of our numerical example.

3.1.4 Modifications of the Kasdin/Walter approach

The original PLN generation approach described in [KW92, Wal94, Kas95] provides a method to simulate the sampling of PLN with a desired shape $\propto f^{\alpha}$, but generates N samples at once, separated by a fixed sampling interval of τ_0. This implies two challenges:

- **Maximum sample size:** Simulating noise data with a length of N samples requires the application of a filter of length N to an input noise vector of length N. On the author's PC this becomes infeasible for $N \geqslant 10^7$. If we want to simulate noise in the MHz region for more than just a few seconds, this becomes very inefficient.
- **Effort depends on simulation time:** As we have specified in our *efficiency* criteria, we would like a noise generation method that gets more efficient when it is sampled less often. Thus, the fixed sampling interval τ_0 of the original approach needs to be worked around.

The following sections will describe how the original Kasdin/Walter approach can be modified to achieve these goals. The design of the implementation presented here took inspiration from the description in [GNLS11], where Gaderer et al state that: "Therefore, following this concept, the oscillator model has to be modified in a way that it is able to simulate long periods of time efficiently and also to keep the fine granularity when needed." They describe that their modifications are based on using multiple sampling frequencies, and we will follow a similar approach. The discussion that follows will be split mainly in two parts:

- The first part will describe in a bottom-up fashion different low-level aspects of PLN generation (section 3.1.5).
- The second part will describe in a top-down approach how these techniques were used to implement a library for PLN generation (section 3.1.6).

3.1.5 Aspects of PLN generation

The following sections provide a discussion of various aspects of PLN generation.

Influence of a limited filter length One important aspect when applying the method described in section 3.1.3 is the length of the applied filter. The length of the filter directly implies a lower bound for the frequency range in which the PSD will have the desired shape. This effect is described in [Kas95] as follows: “[. . .], the spectrum has a $1/f^{\alpha}$ form down to a limiting frequency related to $1/N\Delta t$, where N is the length of the truncated coefficient series. Below this frequency the spectral density is flat - the noise is stationary for all α.” Note that [Kas95] uses slightly different notation than this thesis. In our notation, this means that the lower bound for the desired frequency shape is f_s/N (Δt in [Kas95] is what we call $\tau_0 = 1/f_s$). This means that the frequency interval for which we get the desired shape is given as

$$I(f_s, N) = \left[\frac{f_s}{N}, \frac{f_s}{2}\right] \tag{3.3}$$

Implementation options The recursion for the FIR filter coefficients is given in eq. (3.2). A visualization of these recurrences is shown in fig. 3.3. For the values where α is even (WPM, WFM and RW noise), these filter coefficients take on special values:

- **WPM:** Inserting $\alpha = 2$ into eq. (3.2) leads to $h_0 = 1$, $h_1 = -1$ and $h_2 = 0$. As any h_k is always some multiple of h_{k-1} for $k \geqslant 1$, this means that $h_k = 0$ for all $k \geqslant 2$. The resulting coefficients form the sequence $[1, -1, 0, 0, 0, \dots]$. These are the coefficients of a *discrete derivative filter* [Smi97].
- **WFM:** Inserting $\alpha = 0$ leads to $h_1 = 0$, which implies $h_k = 0$ for all $k \geqslant 1$. Thus, the filter for WFM noise consists of a single 1 and is zero otherwise. The resulting coefficients form the sequence $[1, 0, 0, 0, \dots]$. This is exactly what an *identity filter* looks like. Of course, this is no big surprise, as the input noise is already WFM noise and does not need to be changed.
- **RW:** For $\alpha = -2$, it holds that $h_k = h_{k-1}$ for all $k \geqslant 1$, and thus $h_k = 1$ for all $k \geqslant 0$. A filter with these coefficients is better known as a *running sum filter* [Smi97].

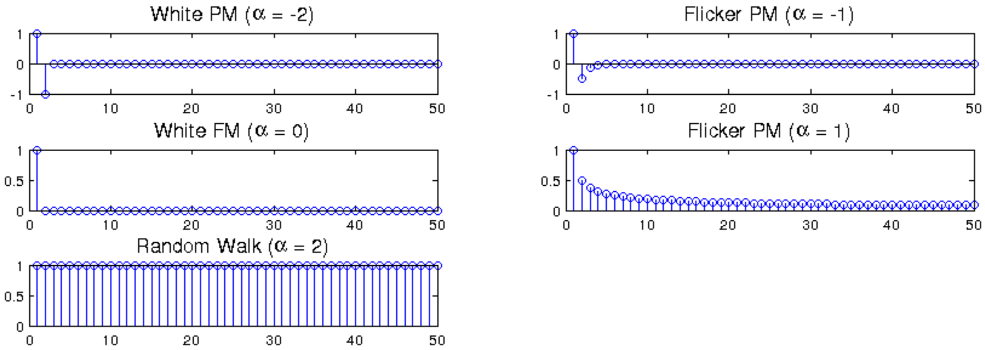


Figure 3.3: Discrete filter coefficients of the Kasdin/Walter method.

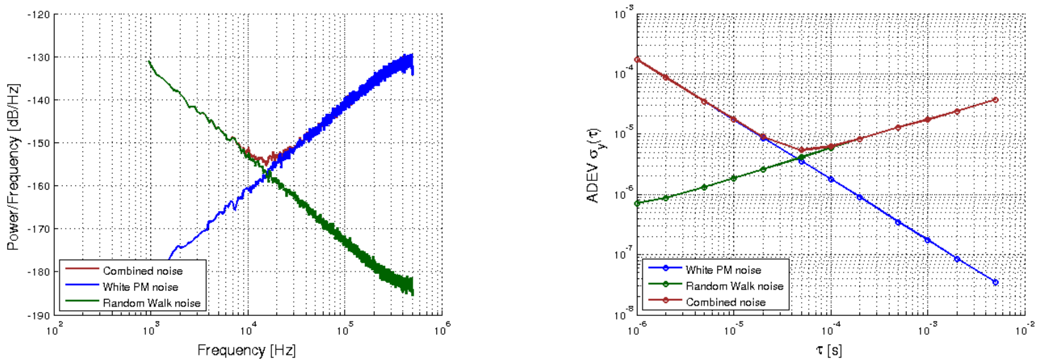
These observations are in perfect agreement to [Ril08], which states about the relationship of PLNs: "Taking the first differences of a data set has the effect of making it less divergent. In terms of its spectral density, the α value is increased by 2. For example, flicker FM data ($\alpha = -1$) is changed into flicker PM data ($\alpha = +1$)."

These three filter types can be implemented efficiently as Infinite Impulse Response (IIR) filters. Thus the same effect as applying the specified FIR filter for these noise types can be achieved much cheaper by directly implementing the filter operations in a recursive way (or no filter at all in the case of WFM). Using a recursive filter not only provides a performance boost, it also gets rid of the lower bound limitation of the resulting frequency interval for these noise types.

Combination of different noise types The PSD of a PLN has the shape of a line in log-log scale. Thus it is easy to see, that in the presence of multiple types of PLNs, there will be regions that are dominated by different kinds of noise. This was already sketched in fig. 2.9. Suppose now there would be an oscillator that suffers from WPM and RW noise, as it is shown in fig. 3.4. It can be seen that the high frequency part of the PSD is completely dominated by WPM noise, and as a result also the corresponding part of the ADEV plot (small values of τ) has a shape typical for WPM. The same holds for the low frequency part of the PSD and the RW noise. Only in the center section where both of them are in the same PSD range both have an influence on the result in the ADEV plot.

An important consequence is that the shape of the PSD of a simulated noise sample could as well be *wrong* in certain frequency regions if the noise gets dominated by another one in the combined spectrum. Suppose for example that the shape of the WPM noise would not continue its f^2 slope for $f < 10^3$, but would become WFM noise with a constant f^0 shape. This would be the case if a shorter filter would have been used to generate the noise, as explained earlier. The difference of the resulting combined PSD would be negligible, and thus also the resulting ADEV would be basically the same. But we could have saved computational effort. We will make use of this observation later, when we combine noise samples that are only correct in certain parts of the spectrum.

Cascading noise generators As already stated, an important part of our PLN generation approach is to simulate PLNs at different frequencies, and to combine them. Figure 3.5 shows an example of the approach that we will apply. Suppose we want to generate noise with a PSD as shown in the top picture (①). To generate it, we would need a filter that is long enough that the PSD has the desired shape for the whole interval, which would be computationally expensive. Thus we split the spectrum into two sub-intervals. We generate the low frequency interval with a shorter filter, that samples at a low frequency. This is shown in (2a). For the finally combined noise, the low frequency part needs to be upscaled to the actually desired frequency. This will be done with interpolation, and is discussed later. As sketched in (2b), the interpolation will cause some unwanted noise in the PSD part of the high frequency interval. On the other hand, the high frequency part of the PSD is also simulated with a shorter



(a) $S_y(f)$ for WPM noise, RW noise, and a combination of both.

(b) ADEV for WPM noise, RW noise, and a combination of both.

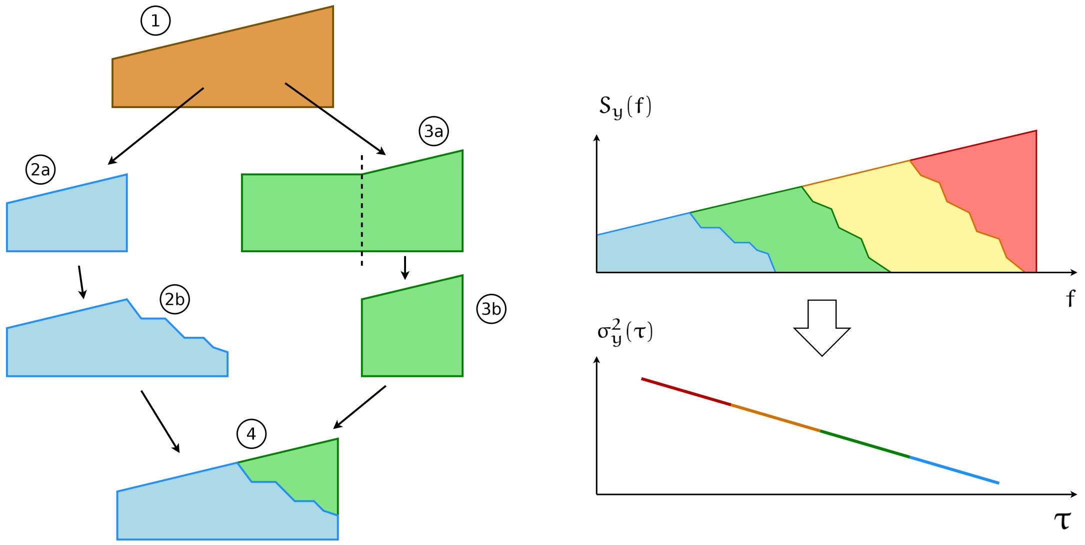
Figure 3.4: Combinations of different noise types.

filter. As the filter is actually too short for the whole interval, the lower part of the high frequency spectrum will have the wrong shape (it will be WFM noise, as discussed in section 3.1.5). This is shown in (3a), together with a cut line. The cut line symbolized what we will do next: we get rid of the low frequency distortions with a high-pass filter, to get a PSD as shown in (3b). When we finally combine the two PSD intervals (④), the unwanted noise that was introduced by the interpolation will be dominated by the correctly shaped noise of the higher frequency interval.

The same approach can be applied repeatedly, as sketched in fig. 3.5b. Using this method, we arrive at a simulated noise sample with characteristics like we wanted, but we have saved computational effort.

Upsampling noise When we simulate PLN at a lower frequency, but would like to interpret it as if it would have been sampled at a higher frequency, it needs to be upsampled. Suppose for example that we simulate noise that was sampled with a frequency of $f_{s,1} = 1\,\text{kHz}$ and use it in the combined simulation of noise that was sampled with $f_{s,2} = 1\,\text{MHz}$. We will need to interpolate the noise with the lower sampling frequency to fit the higher frequency. Unfortunately, any kind of interpolation will cause distortions in the frequency spectrum. We will thus have to make compromise between computational complexity and PSD distortions. A detailed discussion of this topic is found in [Hor74]. In the implementation that is introduced later, we will use either linear or cubic spline interpolation for the TD samples depending on the type of PLN and its simulated sampling frequency.

A comparison of these two interpolation approaches is shown in fig. 3.6: noise that was sampled with $f_{s,1} = 1\,\text{kHz}$ is interpolated to be sampled with $f_{s,2} = 10\,\text{kHz}$. It can be seen that both approaches influence the PSD, but the influence of cubic spline interpolation is smaller. Additionally, the time domain plot of the simulated TD samples

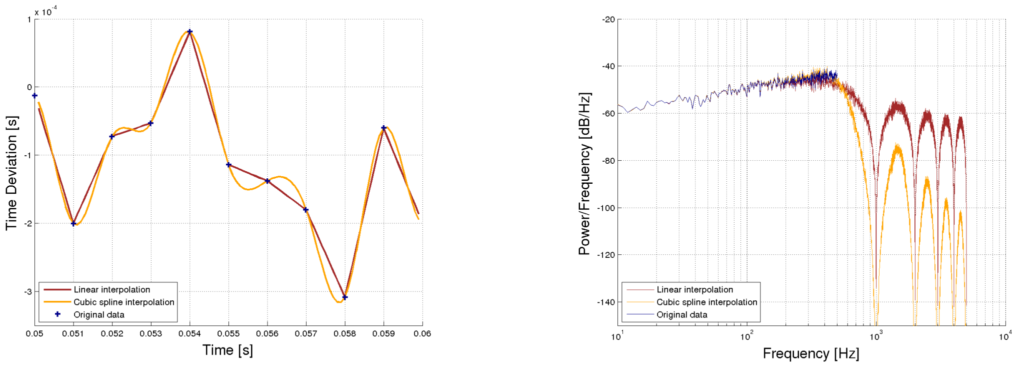


(a) Basic approach how PSD pieces are combined.

(b) Applying the approach in fig. 3.5a multiple times to generate a continuous PSD. This in turn also leads to the desired AVAR shape.

Figure 3.5: Sketch of how PLNs generated at different frequencies can be combined.

is rather smooth, as expected for cubic splines.



(a) TD of interpolated PLN.

(b) PSD of the FFD of interpolated PLN.

Figure 3.6: The effects of linear and cubic spline interpolation in the time domain (TD plot) and the frequency domain ($S_y(f)$ plot).

High-pass filtering As stated before, we want to simulate separate intervals of the PSD individually. Thus, we have to make sure that they do not influence each other when they are combined. The challenge here is that the shape of the PSD flattens out for the low frequencies when the used filter is too short. This is especially a problem for noise types where the PSD increases with the frequency (e.g. WPM, FPM). To counter this effect, we need to apply a high-pass filter for these types of noise. A simple yet efficient filter for this purpose is e.g. the *Blackman window*[Smi97], which is used in the implementation of our PLN generation library.

Skipping intervals Up to now we have already shown how the techniques of upsampling and high-pass filtering can be used to approximate a PSD that would need a large filter with several small filters. Using these observations allows us to approximate a PLN sample of any size. But the computational effort is still tied to the size of the simulated sampling time. What we discuss next combining PLN of different length and different simulated sampling frequency, where we only compute noise samples that are needed for the actual simulation, and skip the rest.

Figure 3.7a shows an example of this approach: noise samples of different lengths are generated at three different sample frequencies. Noise samples that are generated at slower sample frequencies can cover longer intervals with the same filter length. This is sketched in the figure by the larger rectangles of the lower frequencies.

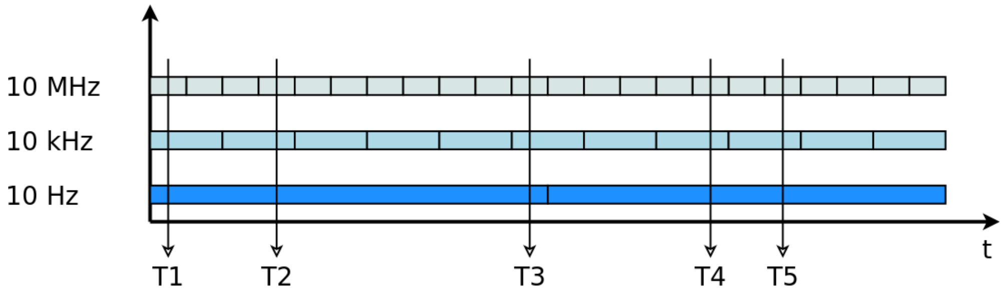
An important assumptions for our later simulation is that we won't need the information of each sample. We will need to know the TD of a clock at irregular intervals, and we would like our clock model to skip unneeded computations to increase the simulation efficiency. The irregular nature of the clock requests is sketched in the plot by the arrows at the times $T_1, T_2, \cdots, T_5$.

If two requests are very close to each other, like T_1 and T_2, we might need the information of the complete frequency spectrum. However, if two requests are far from each other, their respective values won't be closely related, and it might be enough to just simulate the noise contributions of the lower part of the frequency spectrum. For a more specific example: Suppose we read the TD of a quartz oscillator two times with an interval of 1 µs. The difference between these two measurements will most likely be dominated by high frequency noise like WPM. On the other hand, if we measure the same oscillator with an interval of 100 s, high frequency noise probably won't play a role, but low frequency noise like RW will. The basic approach for skipping noise intervals is sketched in fig. 3.7b.

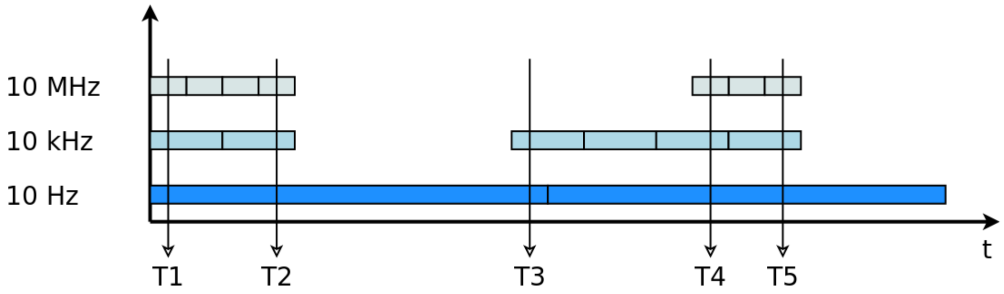
3.1.6 LibPowerLawNoise (LibPLN)

The concepts that were developed in the previous sections can now be combined to form an efficient PLN library, which we will call *LibPLN*. The goal is to have a portable library that can efficiently simulate PLN. The main target for this library is of course the usage as part of an OMNeT++ simulation, but it is also possible to use it in any other program.

In contrast to the previous low level descriptions, this section will introduce the library design in a top-down approach.



(a) Combining noise vectors of different frequencies and lengths.



(b) Unneeded high frequency vectors are skipped to save computations.

Figure 3.7: Sketch of the basic approach to skip noise intervals.

3.1.6.1 Design decisions

To comply with our own requirements (section 3.1.1), the implementation follows the following design decisions:

- It is implemented in C++, as the language is fast, flexible, and widely used. OMNeT++ also uses C++ as the language to model component behavior, and the usage in OMNeT++ is the primary use case for our library. This also favors our language choice.
- The library should be useful on any platform supported by OMNeT++, and it should be useful in applications other than OMNeT++. This implies that
  - No OMNeT++ specific function should be used. In particular any time values will be calculated as `double`, instead of the OMNeT++ specific type `simtime_t`.
  - Any used software library should be portable, and available on the typical desktop platforms:
    - **Gaussian white noise:** To generate Gaussian white noise in a platform independent way, the *Boost*[2] library is used.
    - **Fast Fourier transform:** The implementation of the FFT algorithm is based on the *Fastest Fourier Transform in the West (FFTW)*[3] library.

---

[2] `http://www.boost.org/`

[3] `http://www.fftw.org/`

* **Spline interpolation:** For the cubic spline interpolation an open source implementation called *Cubic Spline interpolation in C++* [4] is used.

3.1.6.2 Architecture

The main task of PLN generation has been divided into individual subtasks that are implemented by different components. To reduce complexity, we follow the *Name & Conquer*[5] approach and introduce new notation to refer to these individual components. These names are arbitrary, but hopefully chosen in a way so that they are easy to understand and remember. The relationships between the individual components are shown in fig. 3.8 as a Unified Modeling Language (UML) class diagram, and the following paragraphs will discuss them in more detail.

TD Generator Our clock model consists of three parts, where the center part is responsible for noise generation. We measure the noise as the TD at time t. A component that implements this relationship $TD(t)$ is referred to as *TD Generator (TDG)*. A TDG can be asked at any time t_1 for the TD at any other time t_2, whether the requested time is in the past or the future. Additionally, its answers to any former request will stay valid forever.

TD Oracle While a TD Generator could be implemented, it would be computationally very expensive. We will thus only use it as a theoretical concept, and introduce the derived concept of a *TD Oracle (TDO)*. A TDO behaves similar to a TDG, but its requirements are relaxed:

- A TDO is allowed to forget. Thus, a request asking for the TD at any time in the past may be rejected. This removes the burden of having either a deterministic function for $TD(t)$ or keeping every past request in memory.
- A TDO may estimate the TD instead of accurately calculating it if the requested time is in the distant future. This enables us to skip unnecessary time intervals, and thus become efficient for DES. What *distant future* means in this context will be explained later.
- Requests to a TDO may become invalid when other requests are made before the first requested time is reached. Example: at time $t = 2$ the TD at $t = 10$ is requested. However, later, at time $t = 3$ the TD at time $t = 5$ is requested. This second request invalidates the first one. The purpose of this requirements relaxation is that it allows our library to be inconsistent with previous answers. This is needed in the case when a previous guess for a future TD would be inconsistent with later necessary exact calculations. However, the consequences are not severe: the original caller just has to request the TD at $t = 10$ again.

The purpose of a TDO is to be as exact as a TDG when it is necessary, and to be slightly inaccurate but faster when it is possible.

---

[4] `http://kluge.in-chemnitz.de/opensource/spline/`

[5] The term *Name & Conquer* is adopted from [GKP94].

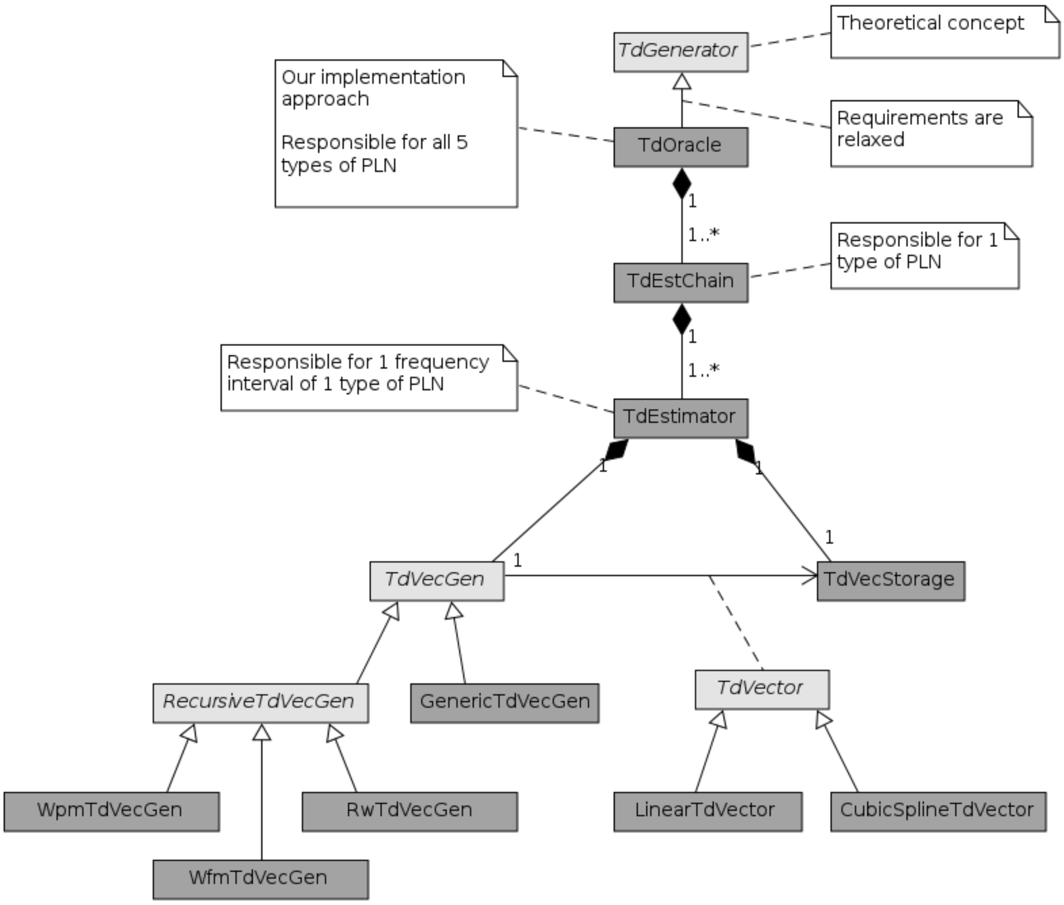


Figure 3.8: UML class diagram for *LibPLN*. Abstract classes are shown in light gray, actual implementations in dark gray.

TD Estimator (TDE) Chain The returned TD value from a TDO contains the noise contributions from all five PLN types. Internally, each PLN type is handled by a separate component. These components are called *TDE Chains (TDECs)*.

As described in section 3.1.5, it is more efficient to split the generation of PLN into several frequency intervals. The components that are responsible for the generation of a single frequency interval are referred to as *TD Estimators (TDEs)*. A TDEC contains one or more TDEs, and forwards any TD requests to them and combines their results.

TD Estimator As already stated, we gain the efficiency of our implementation approach by skipping the calculation of unneeded time intervals. The TDEs are the components which are responsible for generating the individual lines as shown in fig. 3.7. The decision when a calculation is needed or may be skipped is handled in the TDEs. Each TDE is responsible for the PLN generation in a specified frequency interval of the PSD. Depending on the time for which a TD is requested, different cases are distinguished:

- If the TD is requested for the distant future, the TDE decides that its own contribution to the combined TD can be neglected. In this case, it will skip the accurate calculation, and estimate the TD at the requested time. In the current implementation, it as estimated as a single random number from the white-noise generator.
- If the TD is requested for the present or the near future, the TDE will consider its own contribution as *valid*, and compute it accurately.
- If the TD is requested for the past, it is rejected. This complies with our relaxed requirements, and simplifies the implementation.

Up to now we have mentioned several times the terms *near* and *distant future* without explicitly defining what we mean. These terms depend on the current state of a TDE, and thus we need some more notation related to this state to define them. A TDE works in a batch mode: it produces and stores arrays of TD samples. These sample arrays are called TD Vectors (TDVs), and will be discussed later in detail. For now it is only important that a sequence of TDVs forms a continuous TD function for the time interval that is covered by this sequence. The sequence of currently known TDVs changes during the lifetime of a TDE: new samples will be calculated, while others that are already in the past may be forgotten. The TDVs that are currently in the storage form the time interval for which a TDE currently has detailed knowledge about the TD function. We will call the first point in time that is contained in the current TDV sequence the *begin time*, and refer to it is t_{beg}. Similarly, the last point in time in this sequence is called the end time, or t_{end}. The current time at which a request is made to a TDE is referred to as t_{now}. Additionally, each TDE is configured with a time interval called T_{val}. This interval provides us the threshold up to which a TDE would consider its contribution to the combined TD as valid:

- The *near future* is now defined as the time interval $t_{now}..(t_{end} + T_{val})$.
- Anything beyond $t_{end} + T_{val}$ is called the *distant future*.

These terms and the different cases for requests are sketched in fig. 3.9. The choice for the configured T_{val} is a compromise: shorter intervals will allow a TDE to skip more calculations, longer intervals will cause the resulting PLN samples to be more correct.

To fulfill its purpose, a TDE relies on two internal components:

- The generation of continuous TDVs is handled by a *TDV Generator (TDVG)*.
- The component that stores the currently known TDVs and from where TD requests are actually answered is called the *TDV Storage (TDVS)*. Thus, the current contents of the TDVS are what determine the values for t_{beg} and t_{end}.

When a request is made to a TDE, it will either decide to skip the calculation, or it will use the TDVG to fill the contents of the TDVS until the requested time is contained in the storage, i.e. until it holds that $t_{req} \leqslant t_{end}$. The decision handling and internal interactions between those components are shown in fig. 3.10 as an UML sequence diagram.

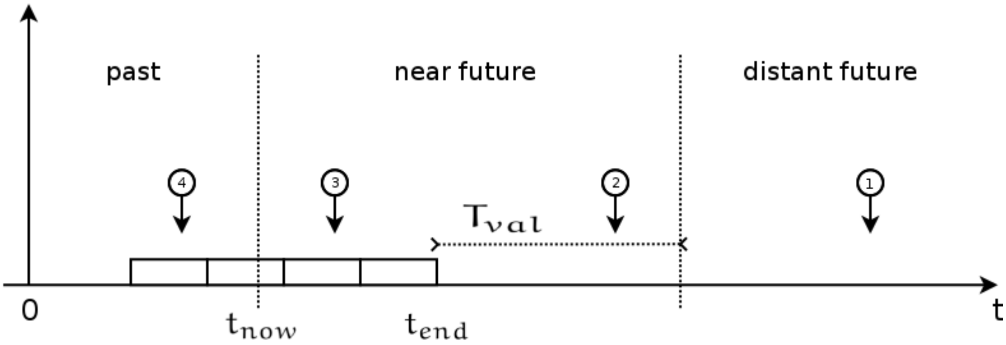


Figure 3.9: Basic case distinction in the request evaluation stage of a TDE. ① is a request for the distant future. ② is for a time in the near future, which has not been calculated yet. On the other hand, ③ is for an already calculated time in the near future. For the decision logic, ② and ③ are handled completely the same. ④ shows a request for the past, which would be rejected.

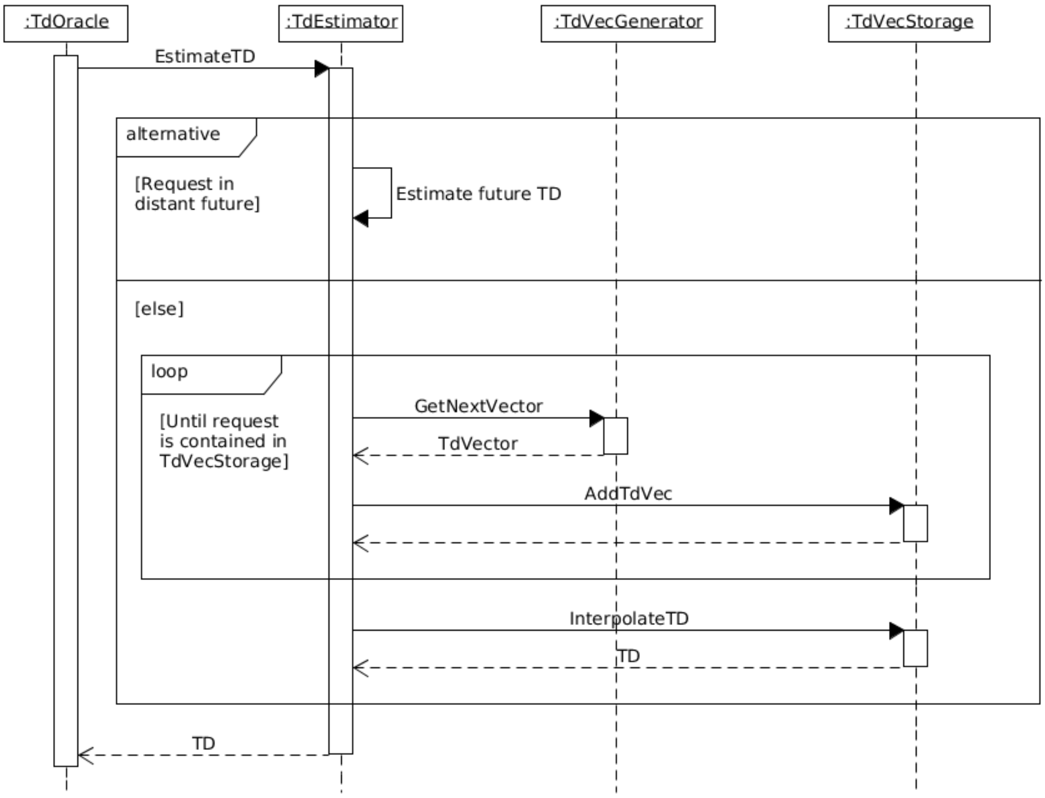


Figure 3.10: UML sequence diagram showing the internal interactions of a TDE.

TD Vector A TDV is the basic entity that is used to handle PLN: it contains a sequence of sampled TD samples. Each TDV covers a certain time interval, and several consecutive TDVs cover a continuous interval, i.e. the end time of one TDV equals the beginning time of the next TDV. While TDVs are based on simulated TD samples with a configured sampling frequency, they support the interpolation for any time inbetween those samples. The current implementation supports two ways: linear and cubic spline interpolation.

TDV Generator The TDVG are at the heart of our library: this is were the actual PLN generation happens. Whenever a new TDV is requested, the TDVG will generate one. How this is done depends on the type of noise and the chosen implementation option:

- **Generic implementation:** In its most generic form, a TDVG implements what is described in the PLN generation approach by Kasdin and Walter[KW92]: White noise is generated and filtered by a specially designed FIR filter. The generated noise is additionally passed through a high-pass filter to avoid influence on lower frequency intervals. The last stage is a conversion from FFD to TD values (using eq. (2.5)).
- **Recursive implementation:** For generating PLN with an even value of α (WPM, WFM and RW), the FIR filter is replaced by a recursive implementation.

Figure 3.11 shows a sketch of the generic TDV generation pipeline.

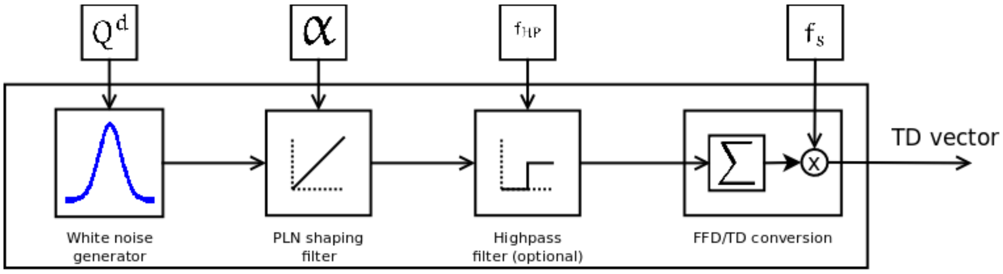


Figure 3.11: Generic TD vector generator pipeline: White noise generated with a standard variance of Q^d, and shaped to be proportional to f^α. After that, it is filtered to drop contributions to the frequency spectrum below f_{HP}. The relative FFD values are then converted to absolute TD values.

TDV Storage The task of a TDVS is quite simple compared to that of the other components: it stores TDVs and when a TD request is made, it finds the corresponding TDV and forwards the request. The other tasks of the TDVS are just related to housekeeping: it tries to find a compromise between memory usage and efficiency by deciding when to free past TDVs and it performs basic plausibility checks on the TDVs it is handed, e.g. if they indeed form a continuous time interval.

3.1.6.3 Evaluation

Remember the goals that we have set for our implementation in section 3.1.1: it should be *realistic*, *efficient*, *flexible* and *portable*.

Portability, Flexibility Portability and flexibility are fulfilled by design: we have avoided any OMNeT++- or platform-specific dependencies and only used libraries that are available for the usual desktop platforms.

Efficiency To be efficient, we required that the computational effort should not depend on the simulated time, but only on the density of requested TD values. This was reached by the skipping behavior that is implemented in the TDEs. Figure 3.12 shows an example of the resulting TD when our TDO is sampled at varying frequencies. At larger sampling intervals, high frequency TDEs will decide to skip their calculations, and save computational effort. This can be seen in fig. 3.13, which shows the simulation speed at different sampling rates. It can be seen that for smaller sampling frequencies, the computational effort decreases. The steps in this graph correspond to the threshold where individual TDEs decide that their own contribution won't be valid for the combined result.

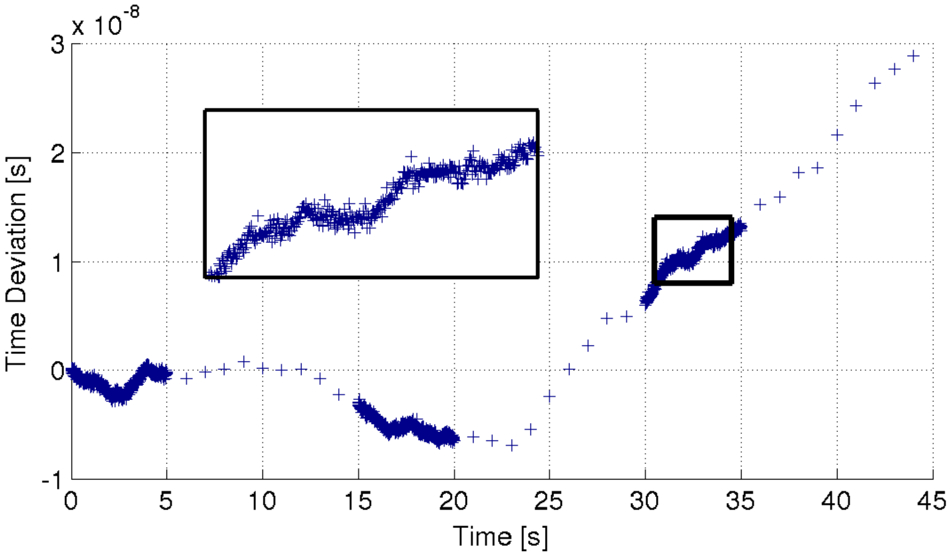


Figure 3.12: TD sampled with changing sampling rates.

Realism For the goal of realistic PLN generations, we wanted our library to generate PLN as it would be generated by a real oscillator, both in the time and frequency domain. As an example, a TDO has been implemented with a desired AVAR shape as given in [GLN$^+$07]. The example supposes that the oscillator ticks with 20 MHz, and the generated PLN is assumed to go up to this value. Thus the highest simulated

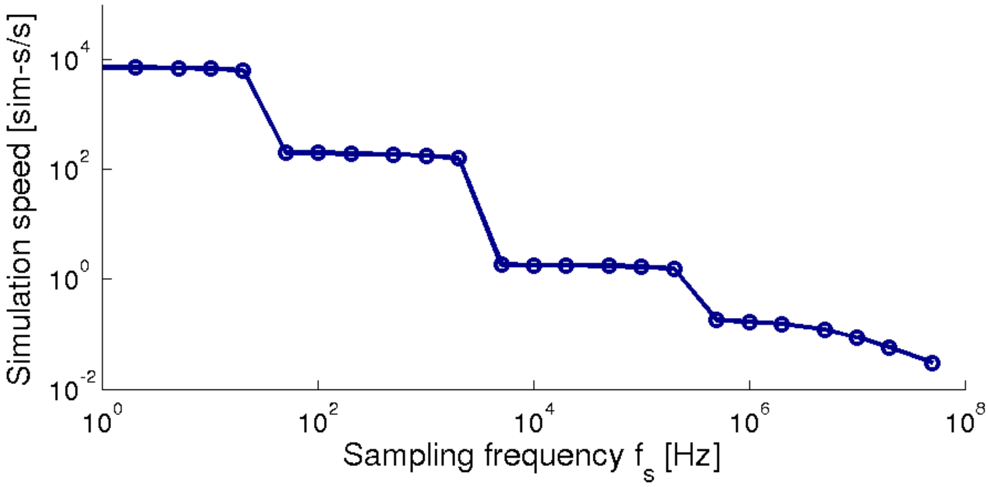


Figure 3.13: Simulation speed for different sampling rates, given in simulated seconds per real seconds.

sampling frequency is set to 40 MHz. An additional assumption was made about WPM and FPM noise: from the AVAR plot it is not possible to deduce if both of these PLNs are present, and what their ratio could be. It is assumed that they are both present in this example, and that they contribute the same amount of noise to the AVAR. The implemented TDO has been sampled at frequencies between 50 MHz and 1 Hz. For every sampling frequency, 10^8 samples were taken[6]. Figure 3.14 and fig. 3.15 show the results of this experiment. The PSD which is given in fig. 3.15 shows slight artifacts of our approach: at 1 MHz a step can be seen that is due to the necessary sharp high-pass filter, and it is obvious that the TDEC for WFM noise does not start contributing until a sampling frequency of 100 Hz. However, the actual ADEV plots which are shown in fig. 3.14 are basically indistinguishable from the specified plot (shown as a black background plot) at any frequency below 20 MHz, and for any given interval length. Thus, the implementation approach presented in this section is also considered to be realistic.

---

[6] On the author's Personal computer (PC), this is the upper bound of what can be analyzed in a feasible time.

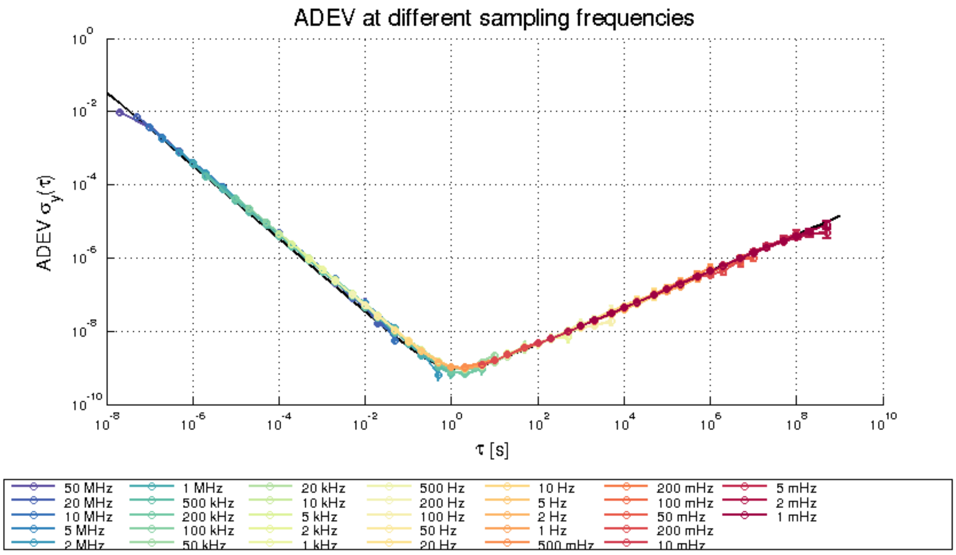


Figure 3.14: Resulting ADEV for different sampling rates. The black line in the background was the configured ADEV. The oscillator was assumed to tick with 20 MHz.

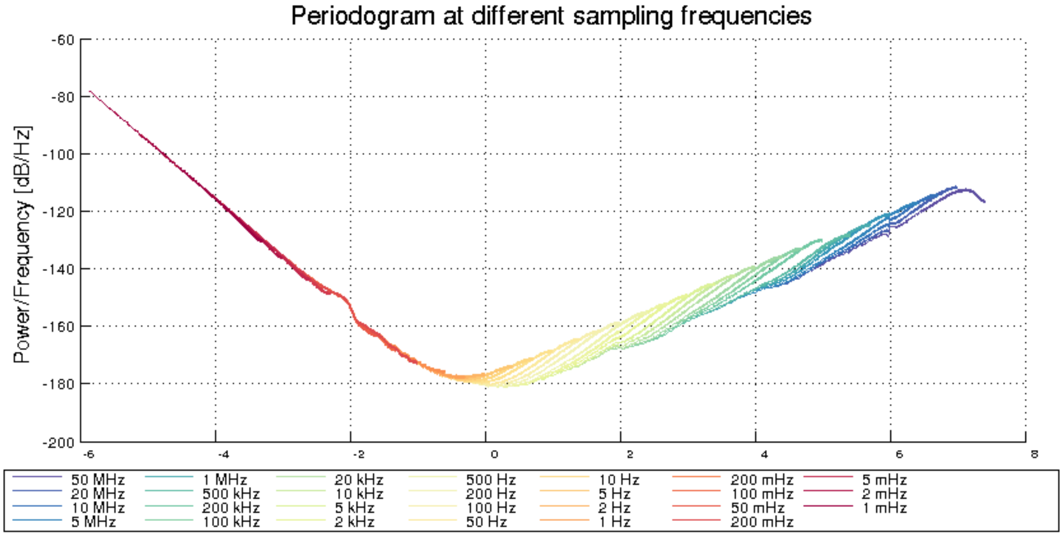


Figure 3.15: The PSD for the FFD corresponding to the measurements in fig. 3.14. It can be seen that the PSD for high frequency WPM noise scales with the sampling rate. The irregularities in the PSD plot stem from the imperfection of the used filters.

3.2 OMNeT++ simulation models

As already stated, this thesis deals with the simulation of PTP using the OMNeT++ network simulation framework. This chapter describes in more detail how this is done.

3.2.1 Project relationships

The task of simulating PTP networks is solved by a combination of different projects. An overview over these projects and their relationships is given in fig. 3.16. Projects shown in yellow are upstream projects, all other icons represent projects that were established for this thesis. As described in section 3.1 the *LibPLN* library depends on the FFTW, Boost and Spline libraries. Using *LibPLN*, several examples for ADEVs of oscillators are implemented in another library called *LibPLN_Examples*. This example library in turn is used by *LibPTP*, the main simulation library for PTP components. The focus of this section will be the description of *LibPTP*. Additionally, *LibPTP* depends on the Boost and INET libraries, and it makes use of a GNU General Public License (GPL)-licensed *Buttonized*[7] icon set. Several generic OMNeT++ components that were developed for this thesis, but which are not PTP-specific are included in a library called *OMNeT_Utils*. These utilities are distributed in a separate library in the hope that they might be useful for other projects with no relationship to PTP. Finally, the components in *LibPTP* are used to construct virtual PTP networks, in a project which is referred to as *PTP_Simulations*. These simulations will be discussed in chapter 4.

Remark: All software projects which were developed for this thesis are available under open-source licenses from the author's Github account[8].

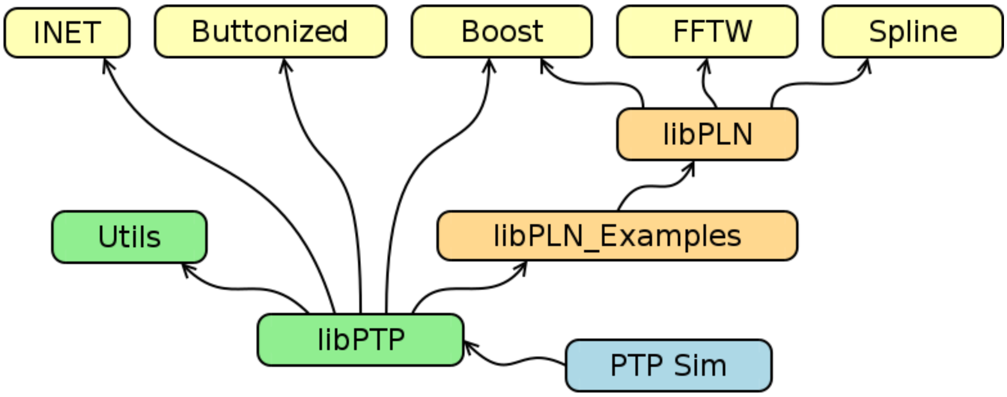


Figure 3.16: Relationships of the projects involved for the PTP simulation. Arrows symbolize a *depends upon* relationship.

---

[7] http://kde-look.org/content/show.php/?content=161553

[8] https://github.com/w-wallner/

3.2.2 PTP node architecture

At the core of the PTP simulation is the model for a PTP-capable network node. The model used for this thesis is shown in fig. 3.17. The left side of the figure shows the internal structure of a *PTP_BasicNode*. This model can be customized via a large set of parameters, and can be used as either a Ordinary Clock, Boundary Clock or Transparent Clock. Internally, it consists of several simulated software components, and a simulated PTP-capable Network Interface Card (NIC). The internal structure of such a NIC is show on the right side of fig. 3.17. We will first give the reader a coarse overview of these components, and provide a detailed discussion of each component later in section 3.2.3. The individual components shown in fig. 3.17 are as follows:

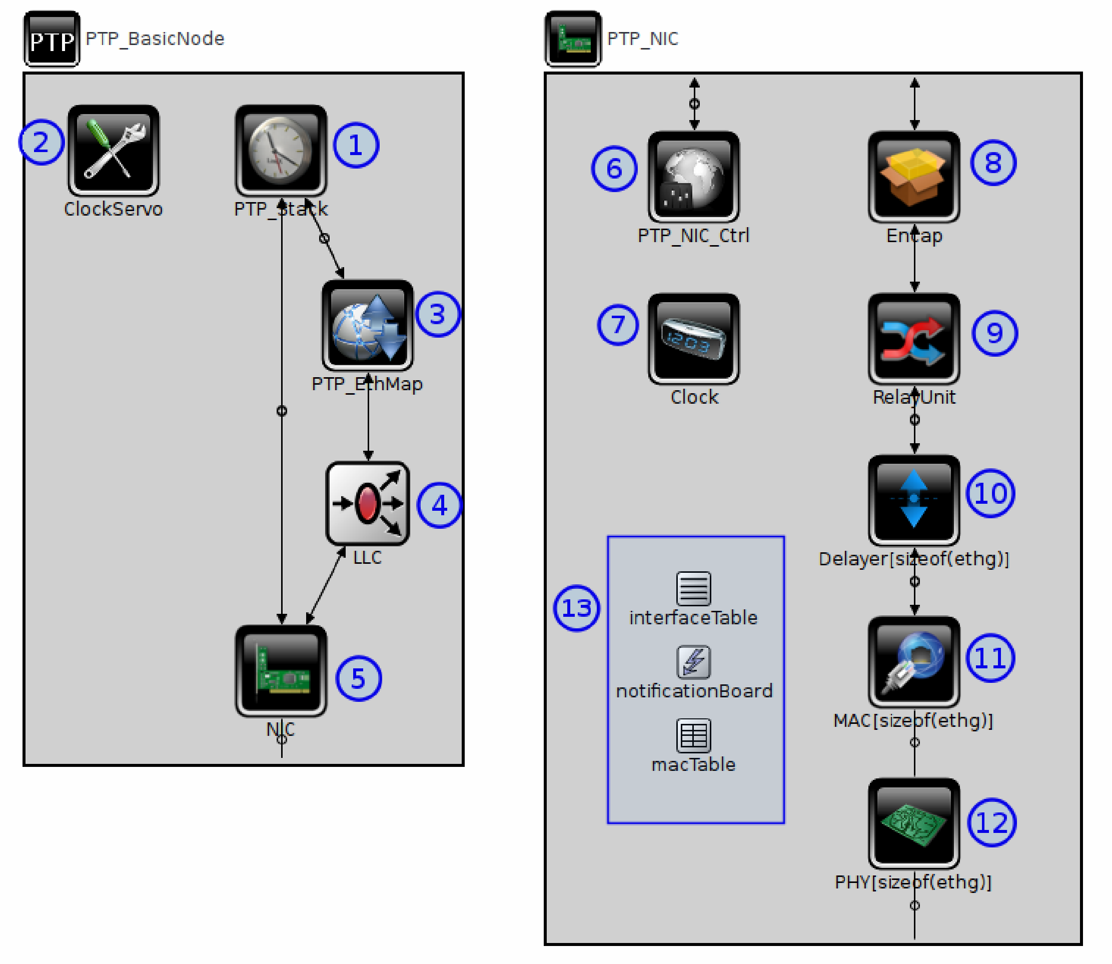


Figure 3.17: Architecture for a basic PTP-capable network node.

① This is one of the most important components in our system: the simulated PTP stack. Most parts of [IEE08] are implemented in this component. The PTP implementation is independent from the actual physical connections of the simulated PTP ports.

② A clock servo for correcting the measured time offsets.

③ In our case we are interested in the performance of PTP over Ethernet, which is specified in Annex F of [IEE08]. This part of the standard is implemented in the PTP/Ethernet-mapping component.

④ The network interface should not be exclusively used by the PTP stack, but should be accessible by arbitrary applications. On the other hand, we need to make sure that received PTP messages are forwarded to our software stack. In a real system, a solution for this multiplex/demultiplex challenge would be implemented as part of the Logical Link Control (LLC) layer of the OS or the system's firmware. Our LLC component simulates such a behavior.

⑤ The final component is a simulated PTP-capable NIC. The internal structure of this NIC is shown on the right side of fig. 3.17.

⑥ The hardware on a PTP-capable NIC needs to be configured with certain PTP relevant parameters, e.g. the currently measured path delay in a P2P network. Such hardware support is simulated in a component called *PTP_NIC_Ctrl*.

⑦ The actual local hardware clock of our network nodes is simulated as part of the NIC. This component is responsible for introducing realistic noise, to provide an interface to correct such noise, and to schedule events.

⑧ The *Encap* component encapsulates generic PTP messages into Ethernet frames, and also handles the decapsulation for received frames.

⑨ On a device with multiple ports, we need to decide to which ports a received message has to be forwarded. This is done in the *RelayUnit* component.

⑩ To simulate delays inside the individual PTP nodes there are special delay components on the paths between the relay unit and the output ports.

⑪ Each Ethernet port is connected with its own MAC. Besides the typical network related functions provided by a MAC our simulated devices are supposed to be PTP-aware and have additional responsibilities, e.g. timestamping PTP frames.

⑫ Real Ethernet physical layers (PHYs) tend to have different forwarding delays in receive and transmit direction. The purpose of our simulated PHY components is to introduce such delay asymmetry.

⑬ Several components inherit from parts of the INET library which have certain assumptions about their neighborhood, e.g. that there has to be a component with the name `interfaceTable`. The components marked with ⑬ are there to meet these requirements.

3.2.3 Individual parts

This section provides a more detailed description of the individual parts in our model of a PTP-capable network node.

3.2.3.1 Clock module

Our clock module needs to fulfill several tasks, and to reduce the overall complexity these tasks are implemented in a hierarchy of different classes.

`HwClock` The most basic class is called `HwClock`. Its purpose is to implement our three-part clock model, consisting of a perfect oscillator, a noise generator and a perfect counter. Thus, this component will transform the perfect continuous real-time into the noise afflicted discrete hardware time of our local clock.

A combination of only the first and the last part of our clock model can be implemented in a simple way: OMNeT++ provides the Application Programming Interface (API) call `simTime()`, which returns the current value of the simulated perfect real-time. Using integer division, the remainder of the current `simTime()` and the tick length of our simulated clock corresponds to the phase of the oscillator part. On the other hand, the quotient corresponds to the number of completed oscillations, which is the value that our simulated counter should currently have. The third component, the noise generator, can be implemented by adding a TD value to the current real-time before the division. For this purpose, our `HwClock` model is equipped with a component called a *TdGen*[9], as it generates TD values. The TdGen is an abstract concept, and *LibPTP* provides several possible implementations:

- `PerfectTdGen`: A TdGen that returns always 0. The local time estimate of a `HwClock` using this implementation will always be perfect.
- `ConstantDriftTdGen`: The TdGen provided by this implementation is a linear function of real-time.
- `SineTdGen`: This implementation provides a TD that resembles a sine function, depending on real-time.
- `LibPLN_TdGen`: While the first three implementations are mainly interesting for developing and debugging purposes, this implementation actually provides realistic TD values by using the *LibPLN* library which was described in section 3.1. On the other side, this implementation is of course slower than the naive other variants.

`AdjustableClock` Building atop `HwClock` is the `AdjustableClock` class, which provides an API for other modules to change the current time value as well as to scale the progress of measured time on the local clock. These adjustments make it possible to establish a relationship between the otherwise arbitrary numerical values of several clocks, and are thus the basis for any synchronization.

`ScheduleClock` The last remaining task for our clock model is that of event scheduling, which is implemented in the `ScheduleClock` class. In simulations that are not as

---

[9] This should not be confused with the internal component of our PLN generation library called the TD Generator (TDG). Both of these components generate TD values, thus the similar names, but they operate on different levels of our simulation model.

time sensitive as ours, modules can schedule events using OMNeT++'s `scheduleAt()` API. However, this would mean that scheduled events occur always at the correct real-time, which is not what we need. We want that the scheduled local events suffer from the noise of the local clock. For this purpose, the `ScheduleClock` class provides the `ScheduleRelativeEvent()` and `ScheduleAbsoluteEvent()` API functions for other modules to schedule their local events.

3.2.3.2 PHY

The PHY in our simulation is a very simple module: its only purpose is to provide a configurable delay on both the receive and transmit paths. This can be used to simulate the asymmetry characteristics that are often found in real network chips.

3.2.3.3 MAC

Our MAC module is derived from the `EtherMAC` module which is provided by the INET library. The basic INET module already provides the generic network functionality of a MAC, and our module only has to add PTP specific behavior on top of that. Our PTP-aware MAC implements the following tasks:

- PTP frames are timestamped on ingress and egress.
- If needed, the residence time of a PTP frame is corrected on egress.
- In P2P networks, the ingress port's *peerMeanPathDelay* is added to PTP frames on egress.
- The configured asymmetry correction for the port is applied to the relevant frames on ingress and egress.
- The MAC can also be configured to simulate a fault. The user may specify a time when the fault should happen, and how long it should last. Until the MAC recovers, it will drop any received frames. The node's PTP stack will be notified of both the fault and the recovery event to act accordingly.

To fulfill its tasks, the MAC relies on the help of a clock module for the timestamping and a module of type `PTP_NIC_Ctrl` for PTP-related support.

3.2.3.4 Relay Unit

The task of a relay unit is to decide to which output ports the frame of an input port should be forwarded. Our relay unit implementation is built on top of the `MACRelayUnit` module of the INET library. Generic Ethernet frames are handled completely by the INET implementation. The handling of Ethernet frames that contain PTP messages is done by our implementation. What happens to a certain frame depends both on the message type as well as the PTP clock type of the current network node. E.g. a `Sync` frame received by a TC would have to be forwarded to all other ports, while the same frame would be swallowed by a BC. On the other hand, a received P2P frame may never be forwarded to another port and has to be swallowed in any case.

3.2.3.5 Delayer

On the receive and transmit paths between the relay unit and the MAC modules there are `Delayer` modules. The components can be used to simulate path delays of messages that pass through one of our network nodes (e.g. they influence the residence time in a TC).

This delay model is a very simplified approach. The actual behavior of packet delays in switches has not been researched as part of this thesis. Thus, this is the part of our model that is probably the *most unrealistic*. Any user of our simulation model needs to be aware of this restriction.

However, the influence of this simplification should be rather small: "What is important is the precision of ingress and egress timestamps and how well the local clock rate agrees with that of the grandmaster. If the residence time could be measured perfectly the details of transit across the switch fabric are irrelevant." [John Eidson, personal communication, November 2015]

3.2.3.6 PTP_NIC_Ctrl

The *PTP_NIC_Ctrl* module supports the individual MACs with several PTP related services:

- **Configuration interface:** the PTP stack needs to configure its logical PTP ports with parameters (e.g. measured path delay, or asymmetry correction values). However, the stack has no knowledge about the actual physical implementation of its logical ports. The *PTP_NIC_Ctrl* acts as a generic interface that interacts with the stack and forwards the configuration to the physical MAC modules.
- **Request interface:** In certain situations, the MACs need to request the creation of new frames from the local PTP stack. An example would be a two-step TC that receives a one-step `Sync` frame. It can't correct the residence time of the `Sync` message on-the-fly, thus it needs to add this time to the correction field of a corresponding `Follow_Up` frame. But as the `Sync` frame is a one-step message, the TC can't expect a `Follow_Up` frame from the original sender. In this case, it has to forward the `Sync` frame as a two-step message, and create its own `Follow_Up` frame. In our simulation model, the egress MAC can in this case request a `Follow_Up` message at the `PTP_NIC_Ctrl` module, which forwards the request to the local PTP stack.
- **Notify interface:** Similar to requests, the MAC use the `PTP_NIC_Ctrl` module to notify the PTP stack of certain events (e.g. faults).
- **Message matching:** The handling of certain messages by the MAC depends on values of different messages. E.g. in a two-step TC the residence time correction for a `Sync` frame needs to be written into the correction field of a matching `Follow_Up` frame. The MACs need a place to store and retrieve such time values, and this is again supported by the *PTP_NIC_Ctrl* module.

3.2.3.7 Encap

The higher layers in our network nodes (everything above the NIC) deals only with PTP messages, while all modules inside the NIC deal with Ethernet frames. The interface provided by the NIC is the `Encap` module, which handles the translation between the different message types. Our `Encap` module inherits from the `EtherEncap` module that is provided by the INET library. It additionally provides PTP specific behavior, which takes care of two things:

- It handles the routing information needed for PTP frames so that they are forwarded to the correct port.
- To be visually more appealing, the PTP frames in our simulation have special icons. The `Encap` module also assigns the Ethernet frames the corresponding icons if they contain PTP frames.

Packages that are received from the higher layer are expected to have meta information assigned, so that the `Encap` module knows what information has to be filled into the Ethernet frame (e.g. the destination MAC address).

3.2.3.8 Logical Link Control

The purpose of the Logical Link Control (LLC) module is to provide access to the NIC to different services. It has several connections for higher layers, but only a single connection for a lower layer. Any package that is received on a higher layer input gate will be forwarded to the single lower layer output gate. A package that is received on the lower layer input gate will be forwarded to exactly one higher layer output gate. The decision for the output gate is based on the `EtherType` of the received message in this case.

3.2.3.9 PTP/Ethernet mapping

The *PTP/EthernetMapping* modules handles two different tasks:

- The PTP standard specifies in its Annex F how PTP messages have to be encapsulated into Ethernet frames. It defines e.g. that the `EtherType` has to be `0x88F7`, or that the destination MAC address for `PDelay_Requ` messages has to be `01:80:C2:00:00:0E`. This information is added for PTP messages in transmit direction.
- The PTP stack is agnostic to the actual physical connection of its PTP ports, it just sends and receives PTP messages on different gates. In our case, all of these messages go through the same network stack, enter the same NIC, but leave the node through different Ethernet ports. To support the routing decisions in the relay unit, the *PTP/EthernetMapping* module adds meta information about the designated output port to the PTP messages.

3.2.3.10 PTP Software stack

Most parts of [IEE08] are implemented inside the `PTP_Stack` module. The standard describes different tasks that have to be implemented, and some of them are specific to a certain PTP port while others apply to the whole network node. The structure of the `PTP_Stack` module is designed to match these specifications. The main stack class has an array of `cPort` class instances available, that cover the port specific handling of PTP. The handling of individual PTP messages is implemented in service classes. E.g. the sending and receiving of `Sync` messages, as well as the handling of associated timeouts is implemented in a class called `SyncService`. As this would be an example of a port specific service, each instance of `cPort` has an instance of `SyncService`.

Main PTP stack class Only a minor part of PTP is implemented in the main stack class. Most tasks are delegated to the ports, and from there further to the individual service classes. The remaining tasks for the main PTP stack class are as follows:

- **Datasets:** The PTP standard specifies several data sets that are not port specific: *defaultDS*, *currentDS*, *parentDS* and *timePropertiesDS*. These data sets are handled by the main stack class.
- **Gateway:** Access to common resources is governed by the main stack class. E.g. there is only one clock servo, and which port may currently steer it has to be decided by the main PTP stack class depending on all port states.
- **State decision support:** The PTP standard specifies that in certain intervals every port has to decide about its future state. For these state decisions, the individual ports need knowledge from all other ports. The main PTP stack class collects the needed information, and provides it to all ports.
- **Message handling:** Messages are received by the main stack class, and have to be forwarded to the port corresponding to the receiver gate. On the other hand, ports don’t have direct access to the gates of the stack module. Thus, if a port wants to send a frame, it requests the transmission of a frame by the main stack class.

Ports As with the main PTP class, also the ports implement only a minor part of the protocol. Most tasks are delegated to the service classes. The two relevant areas that ports deal with are as follows:

- **Port data set:** According to the specification, each logical PTP port has to have an associated port data set (*portDS*) with port specific information and configuration.
- **State handling:** The PTP standard specifies a state machine for each individual port. Most of what is implemented in the port class deals with the management of this state machine. Depending on the current state of a port, different services are turned on or off. E.g. a port in `MASTER` state will have an active `AnnounceService` to send `Announce` messages, while a port in `SLAVE` state will have the same service disabled.

Services There are mainly two different classes of services: port-specific or generic. All services except the state decision service are port-specific.

State Decision Service The state decision service is started when a node starts up, and remains active independent of individual node states. Its task is to trigger reoccurring state decisions for all ports. When the state decision service is activated, it collects information from all ports about their best known foreign master node (called `Erbest` in PTP terms). From this information it calculates information about the best known foreign master, referred to as `Ebest`. Each port then decides based on the common `Ebest`, its own `Erbest` and its current state what its next state should be, and changes the state accordingly. This service is the implementation of the BMC algorithm.

Announce Service When active, this service broadcasts information about the best known master on its associated port. The frequency of these `Announce` messages is determined by the configured announce interval. If the port is in a state where it needs a master (`SLAVE` and `UNCALIBRATED`), a timeout for missing `Announce` messages is monitored. In case the master would fail, this would then trigger an error condition and a reevaluation of the current state.

Sync Service A port in `MASTER` state will periodically broadcast its current time to its neighboring PTP nodes. This is handled by the port's `SyncService`. On nodes in the `SLAVE` or `UNCALIBRATED` state the sync service is responsible for the handling of received `Sync` and `Follow_Up` frames. This means it has to calculate the current estimate for the *offsetFromMaster* value, and to steer the node's clock servo accordingly. Depending on the configuration, the sync service might apply a filter method on the *offsetFromMaster* value before handing it to the clock servo. The clock servo has to decide how the node should react on the current offset estimate (e.g. scale the clock, or carry out a time jump), and the sync service is then responsible for carrying out the clock servo's decision.

Delay and PDelay Service The path delay associated with a given port is handled by its *DelayService* or *PDelayService*, depending on which type of delay mechanisms is supported by this node. In case of an E2E node, the measured *meanPathDelay* value is stored in the nodes common *currentDS* data set. If the node simulates P2P delay measurement, then the measured *peerMeanPathDelay* value is stored in the port's *portDS* data set. Similar to the sync service, also the delay measurement services can apply a filter on the estimated measurement values.

3.2.3.11 Filters

The PTP stack of a node estimates several parameters of its surrounding network: *offsetFromMaster*, *peerMeanPathDelay* (P2P) and *meanPathDelay* (E2E). Instead of using these estimates directly, they could be filtered, e.g. to improve the robustness of the clock servo decisions. *LibPTP* provides the basic infrastructure to supply customized filters for these measurements, as well as two very basic filter implementations:

- **Identity filter:** All measurements are used directly.
- **Running average:** The average of the last N values is used. This filter can optionally discard the minimum and maximum of the last N measurements and calculate the average using the remaining $N - 2$ values. The simple discarding of these values improves the filter robustness quite well in the case of transient spikes in the measurement.

Figure 3.18 shows an example of an *offsetFromMaster* measurement where a simple running average filter with $N = 5$ and discarding of minimum and maximum is applied.

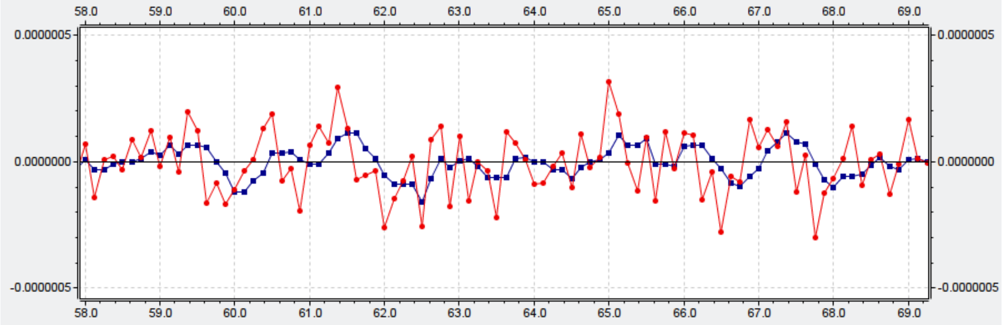


Figure 3.18: Estimation for the *offsetFromMaster* measurement: raw (red) and filtered (blue).

3.2.3.12 Clock servo

The PTP standard states that the estimated offset should be compensated, but it does not state how this should be done. While there would be many possible choices for the design of a clock servo, it seems rather common to use a PI-based servo type. This is based on the following observations:

- [Eid06] states that “[. . .] most reported implementations use a simple proportional-integral controller.”
- The two open source implementations *LinuxPTP* and *PTPd* use a PI-based clock servo.
- The thesis author has personally asked manufacturers on fairs/conferences about their clock servo implementations, and the most common answer was “PI-based”.

LibPTP provides a generic framework to implement custom clock servos, and provides a simple implementation based on the clock servo design of the *LinuxPTP* project. The clock servo performs a series of initialization steps, and finally behaves as a PI controller. These steps are as follows:

- `SYNTONIZE`: In the first step, the clock servo approximates the relative frequency of the local clock compared to the master clock over a configured number of sampling intervals. This approximation is later used as the starting value for the integral part of the PI clock servo.
- `JUMP`: Simply scaling the clock might take a long time to correct a large offset. Thus, the second step performs a time jump: it directly sets the local clock to the estimated time of the master clock.
- `SCALE`: Finally, the remaining clock offset is handled by a PI controller. In case the *offsetFromMaster* reaches a certain preconfigured threshold, the clock servo restarts the state machine to begin again with syntonizing.

3.2.3.13 Time Difference Observer

To analyze the results of a PTP simulation, one often wants to calculate the difference between the local times of two nodes. Additionally, it is nice to have this information available during the simulation run, especially when using the GUI. For this purpose, *LibPTP* contains a special simulation component called the `TimeDiffObserver`. This module gets configured with the relative paths to two clock modules, and a time interval. During the simulation it queries the two clocks in the specified interval, shows the difference to the user in the GUI and traces it for later analysis. An example of such a module in action is shown in fig. 4.4.

3.2.4 Configuration options

The individual components of our PTP-aware network can be configured in various ways. The most interesting examples are listed here:

- **Clock types:** a PTP node may assume different roles:
    - **OC/BC:** If the parameter `PTP_Clock_Type` is set to `ORDINARY`, the PTP stack will behave as an OC or a BC, depending on the configured number of ports. The boolean parameter `slaveOnly` further specifies if such a node may become master on any of its ports or not.
    - **Standard conform TC:** In case `PTP_Clock_Type` is set to `TRANSPARENT` and `slaveOnly` is *true*, the PTP node will behave similar to a TC as described in the PTP standard: it will only actively participate in the P2P delay mechanisms, and otherwise try to be as transparent as possible.
    - **Active TC:** *LibPTP* provides support to simulate a type of clock that is not specified in the PTP standard: an *active* TC. This is a TC that may actively take part in the BMC algorithm, and in case it is the best available clock also become `MASTER` on its ports. For this clock type, `PTP_Clock_Type` needs to be set to `TRANSPARENT` and `slaveOnly` to *false*. *Remark:* This is one of the advantages of a simulation-based approach: it is easy to customize the models and to try out non-standardized behavior.

- **PTP options:** To simulate a wide variety of PTP devices, *LibPTP* provides parameters to set many PTP options, e.g. clock attributes, delay mechanism, BMC algorithm, two-step flag, and many more.
- **PTP profiles:** The PTP standard got extended by several domain specific profiles. Basic support for PTP profiles is also implemented in *LibPTP*. In case a user configures a node to use a certain profile, *LibPTP* ensures that all parameters match the given ranges of the specified profile. In case the profile is specified as `CUSTOM`, any combination of parameter values is allowed. Some PTP profiles define rather complex additions to the PTP standard. Implementing these feature would require more effort and is not covered by the basic profile support *LibPTP* provides.

3.2.5 Simulation GUI

Once all components are implemented and the required simulations specified, they are usually carried out on the command line in a batch fashion and analyzed once they are done. However, during development, debugging and for instructional purposes it is very valuable to have an intuitive GUI to see what is going on inside the individual components. OMNeT++ provides several mechanisms to provide a convenient GUI to the user, and *LibPTP* tries to make use of it where possible.

Node icons To spare the user from configuring the individual parameters of all nodes in a simulation, *LibPTP* provides the most common network nodes as preconfigured modules with special icons. The meaning of these icons is explained in fig. 3.19.

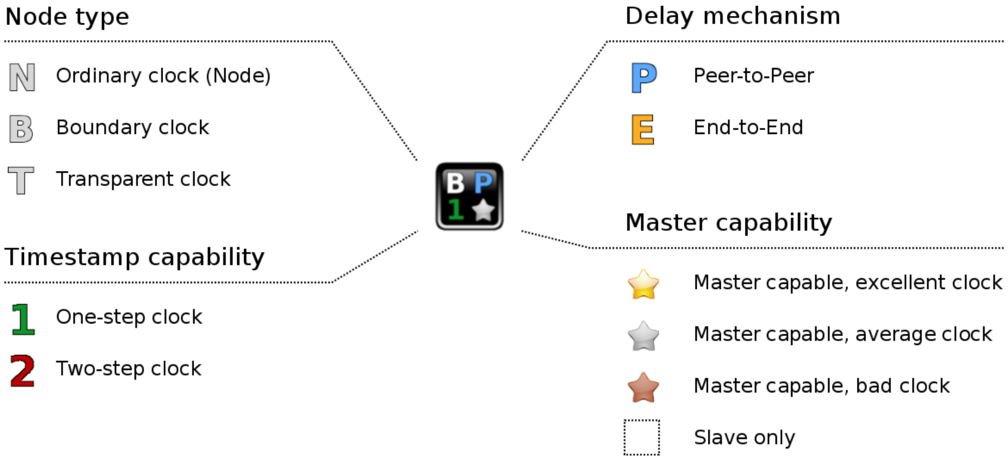


Figure 3.19: Explanation of the symbols used for the example nodes. The example shown in the center would be a one-step BC, that supports P2P delay measurement and is master capable (with average clock attributes).

Figure 3.20 shows an example network using these node symbols. On the left side is a master-capable OC with a good clock, which will become the grand master of the network. It is connected to a BC, which is in turn connected to two daisy chains ending with OCs. The top chain consists of two slave-only TCs, while the bottom chain consists of a slave-only TCs and another BC. All nodes are one-step clocks, except the second TC in the top chain.

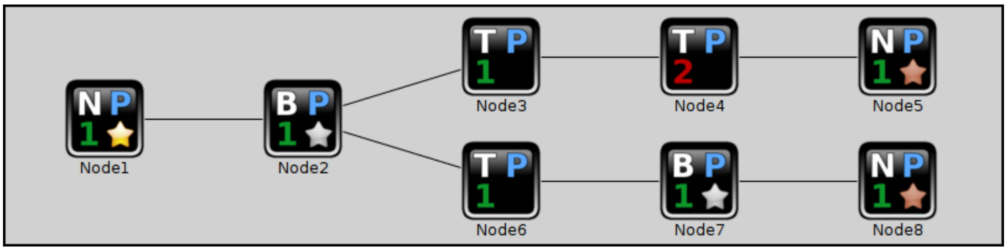


Figure 3.20: Example PTP network to show the individual node symbols.

Message icons As with the node types, it also helps to have different icons for the individual PTP message types. The icons used by *LibPTP* are shown in fig. 3.21.

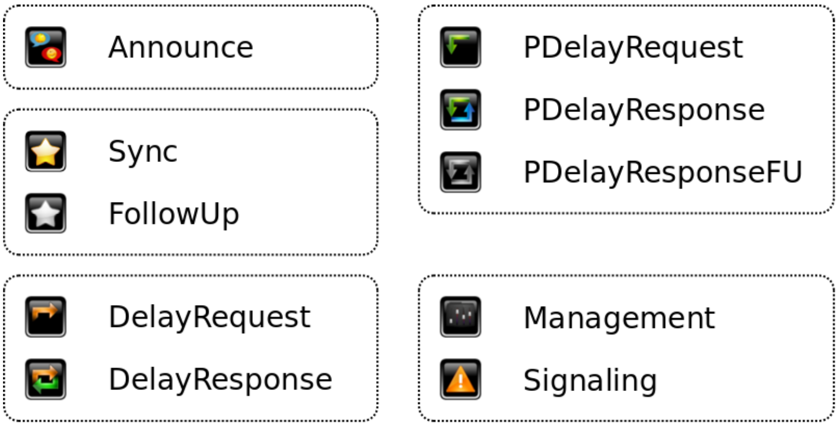


Figure 3.21: PTP message icons used by *LibPTP*.

Other GUI tools In addition to the already mentioned GUI tools, *LibPTP* also supports:

- **`WATCH()` macros:** OMNeT++ provides macros to show the values of module-internal variables in the GUI. These macros are available for member variables, pointers and also for standard C++ containers like `std::vector`. Using these macros it is possible to e.g. watch the current state of the PI controller during runtime.
- **Tool tips:** Another useful GUI feature provided by OMNeT++ are tool tips that are shown when the user hovers with the mouse cursor over a module. *LibPTP* uses these tool tips to show the port states of a PTP node during runtime. An example of this feature is shown in fig. 4.2b.

CHAPTER 4

Evaluation

This chapter describes several experiments that were carried out with *LibPLN* and *LibPTP*. The parameter space for these simulations is huge, and it is clearly not possible to cover all possible configurations. The experiments that are shown in this chapter were chosen as they are representative to show what kind of insights can be gained from the usage of *LibPLN* and *LibPTP*.

Section 4.1 will give an overview of the basic assumptions that were made for these simulations, while section 4.2 deals with the actual experiments. Finally, section 4.3 will discuss the results.

4.1 Assumptions

For each experiment, a huge amount of parameters that could influence the outcome of the experiment has to be set. If not otherwise noted in the experiment descriptions, the following assumptions will be used:

Clock frequency The oscillators in our simulated devices tick with a frequency of 20 MHz. Higher frequencies would be interesting, but increase the computational complexity for the noise generated by *LibPLN*. The chosen value is a compromise based on the computational power of the author's laptop.

ADEV for oscillators One of the most important parameters for the simulation of clock synchronization is the actual noise of the oscillators. The example library of *LibPTP* currently supports two oscillator models:

- In [GLN+07] a detailed formula for the ADEV of an oscillator is given. Thus, it is an interesting candidate for an implementation in *LibPLN*. Unfortunately, the publication does not describe in detail what kind of oscillator was measured. On the other hand, [Eid06] contains an ADEV diagram with an oscillator referred to as *CTS CB3LV*, which is described as "an inexpensive oscillator of the type found on PC motherboards". When comparing the curves of these two oscillators, it can be seen that they are in a similar range. Based on these observations it is assumed that also the ADEV described in [GLN+07] can be used to represent an oscillator as it would be expected in typical COTS devices. This example is referred to as *AvgOsc* in *LibPLN*, and will be the default PLN noise model for our simulations.

- Another implemented example in *LibPLN* is that of an oscillator from a wrist watch. It is based on the published ADEV curve given in [Lom08]. Compared to the other oscillator, it suffers from a higher noise level. Additionally, it does not suffer from WPM or FPM noise, and it has its ADEV minimum at intervals several decades longer than the *AvgOsc* example. *LibPLN* refers to this oscillator as *WatchQuartz*.

A comparison of the ADEVs of these two oscillator examples is given in fig. 4.1. The figure additionally shows the ADEV of a precision oscillator[1] as a reference.

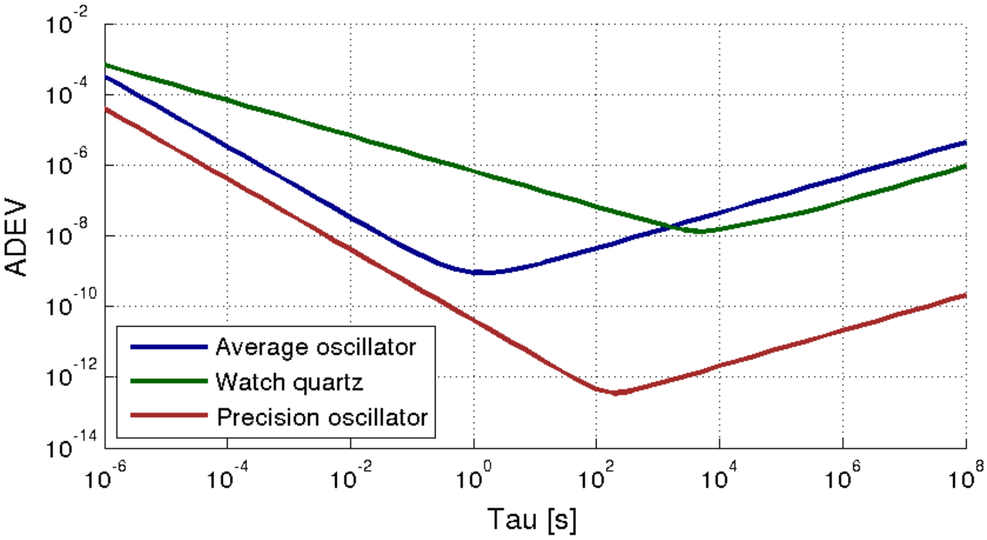


Figure 4.1: Side-by-side comparison of the ADEVs of the *AvgOsc* and *WatchQuartz* example oscillators. Additionally, the ADEV of a precision oscillator is shown to provide the reader a reference.

Clock servo control As stated in section 3.2.3.12, the implemented clock servo is based on a PI controller, with additional logic to carry out time jumps. The actual configuration for the proportional and integral parts will be based on a parameter study, which will be discussed later in the respective experiment description.

---

[1] Pascall Oven Controlled Crystal Oscillator (OCXO) by Rohde & Schwarz, described in [Ram12]

Maximum clock scale factor If a clock device supports the scaling of its time measurement, this option will have device specific limits. According to the data sheet of Intel's i210 network chip [Int12], its clock can be scaled by up to 32767999 ppb. This limit will be used in our simulations.

Network switch delays The switching delay of the individual nodes is simulated via special delayer components inside the NIC. As stated in section 3.2.3.5, a study of the actual switching behavior of real devices was not part of this thesis, and the implemented delay model is very simple. The experiments in this chapter are based on the assumption that frames experience a uniformly distributed delay in the range $1\,\mu s..10\,\mu s$ between the MAC and the relay unit in both directions.

Simtime-scale The timely granularity for simulations in OMNeT++ can configured via the *simtime-scale* parameter. This configuration is a compromise between simulation precision and the maximum possible simulation time. The default setting in OMNeT++ is that the `simtime_t` data type has a resolution of picoseconds, which implies an upper bound for the simulation time of ≈ 100 days. This configuration seems reasonable for our experiments, and was thus used in all following simulations.

PHY delay The delay of PHYs depends on various parameters. The default behavior in *LibPTP*, which is also used in these experiments, is modeled after the delays given in the data sheet of an Ethernet PHY from Marvell [Mar06]. When used with a Reduced Gigabit Media-Independent Interface (RGMII) interface for Gigabit Ethernet, the receive and transmit delays are specified to be in the ranges $176..208\,ns$ and $76..84\,ns$, with steps of $8\,ns$. The actual distribution of these delays is not mentioned in the data sheet. For the sake of simplicity, these values are assumed to be uniformly distributed in our experiments.

Initial conditions When the simulation of a PTP network starts, it is assumed that individual nodes are powered up, and that they don't have any preconfigured knowledge about their environment. How long a node needs for its own initialization can be configured, and for our simulations this value will be set to be uniformly distributed between $10..100\,ms$. The initial local time of the individual nodes could also be set to any value. To keep the graphs simple, we apply the following convention: the eventual grandmaster clock of a network will always start with a local time of $0s$, and all other clocks will start up with a local time in the range of $0..5\,ms$.

Environmental conditions As already stated, our simulation does not take environmental influences like temperature or pressure into account. Thus, our simulations may be representative for laboratory conditions where such influences are stable, but not for environments with harsh conditions like e.g. industrial automation. Users of our simulation environment must be aware of these current limitations.

PTP intervals Various PTP services are handled periodically, and the interval length is configured as the exponent of 2 s, e.g. setting the value of *logAnnounceInterval* to 2 would mean that `Announce` messages are sent every 4 s. If not otherwise mentioned, *logAnnounceInterval*, *logSyncInterval*, *logMinDelayReqInterval* and *logMinPdelayReqInterval* will all be set to 0 (i.e. $2^0 = 1$ s).

Network infrastructure If not otherwise mentioned, we assume that our nodes are connected with Gigabit Ethernet hardware, and that CAT5 Ethernet cables with 2 m length are used. The OMNeT++ model for the Ethernet cables are based on the ones provided by the INET library. The most important parameter is that the delay of such a cable is modeled with a value of 5 ns/m.

Warm up periods Each simulation will begin with a transient period where the BMC algorithm establishes a synchronization hierarchy. For experiments where this is not the main focus, but rather the performance of the clock synchronization in the stable configuration, the initial startup period may be skipped. OMNeT++ provides a special configuration parameter called `warmup-period` specifically for this purpose. Thus, if graphs of the experiment results do not begin at time 0 s, this initial period has deliberately been left out.

4.2 Experiments

The following experiments discuss different aspects of PTP, its implementation in *LibPTP*, and the influence of various parameters on a PTP network. Section 4.2.1 deals with the BMC algorithm, section 4.2.2 describes experiments within a simple PTP network of two nodes and finally section 4.2.3 discusses experiments in networks with a daisy-chain topology.

4.2.1 BMC Algorithm Evaluation

Before we carry out actual synchronization experiments, we would like to test our implementation of the BMC algorithm. The following section will describe a test network given in [Eid06], and discuss how *LibPTP* can be used to simulate the execution of the BMC algorithm and how to gain insight in the individual steps of the algorithm.

4.2.1.1 Experiment 1: Test case 5 of [Eid06]

The book [Eid06] contains several theoretical test cases for the BMC algorithm. The example given in test case 5 is the most interesting, as it is the most complex one. A sketch of this network is shown in fig. 4.2a. The given network consists of OCs and BCs, where the top left node is the best clock in the network, and some nodes are wired in a ring. An execution of the BMC algorithm is expected to solve two challenges in this network:

- OC 1 should be elected as the grandmaster of the network, as it is the best available clock.
- The ring should be broken up. BC 8 is expected to move one of its ports which are adjacent to BC 5 and BC 7 to the `PASSIVE` state. Which one of these ports is selected to be `PASSIVE` does not really matter, and if all other attributes of the two paths are equal the decision might depend on the clock identities. In the shown example the port which is adjacent to BC 7 is chosen to be `PASSIVE`.

Figure 4.2b shows an actual execution of this network with *LibPTP*, and the screenshot contains also a tool tip which shows the port states of BC 8. As expected, one of its ports is in the `SLAVE` state, one is in `PASSIVE` state, and the two port which are adjacent to OC 6 and OC 9 are in `MASTER` state. For the execution of this simulation, the clock identities of the individual BCs have been chosen in a way so that indeed the port adjacent to BC 7 is the one that moves to the `PASSIVE` state to break up the ring.

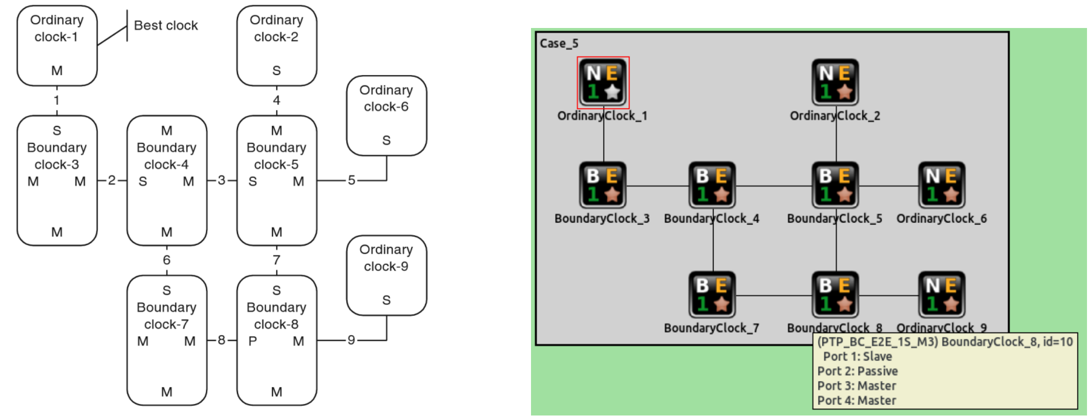
In complicated networks, it might be interesting to trace the individual state decisions of nodes and how they reach these decisions. *LibPTP* provides the user with a variety of trace signals like the currently known *Erbest* data of each port or the individual port state decisions. Additionally, the implementation of the BMC algorithm can be configured to different levels of verbosity for individual nodes. In fig. 4.3 this information is shown for port 2 of BC 8. On startup, it is in the `INITIALIZING` state, and moves to `LISTENING` after that. At this point in time, all other nodes are also in `LISTENING`, and thus they don't broadcast `Announce` messages. Because of this, port 2 thinks it is alone in the network, and wants to become grandmaster (state decision *M2*). Next, the BC 8 learns about other masters, but port 2 still decides to be master (state decision *M3*). At time 0.5 s, port 2 learns about a path to the eventual grandmaster of the network, and becomes a slave. When BC 8 finally receives information about a better path to the grandmaster, port 2 decides to be passive (after time 0.6 s). Thus, in this experiment the observed behavior matches the expectations.

4.2.2 Simple network with 2 nodes

A very basic test case for a PTP network is shown in fig. 4.4. The network consists only of two OCs, where only one of them is master-capable. Additionally, the simulation contains a *TimeDiffObserver* node to measure the time difference between these nodes. The image shows the network at a point in time where the BMC algorithm has already established a master-slave hierarchy and the left node is sending a `Sync` message to the right node.

4.2.2.1 Experiment 2: Parameter study of clock servo parameters

Before we can start our synchronization experiments, we have to select a configuration for our PI clock servo. As the servo design is based on that from the LinuxPTP project, it would be natural to follow their parameter selection approach. The LinuxPTP project uses a heuristic to calculate the servo parameters based on the sync interval.

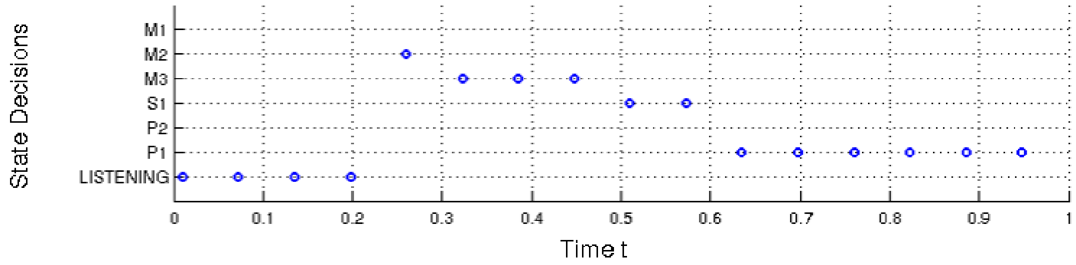


(PTP_BC_E2E_1S_M3) BoundaryClock_8, id=10
Port 1: Slave
Port 2: Passive
Port 3: Master
Port 4: Master

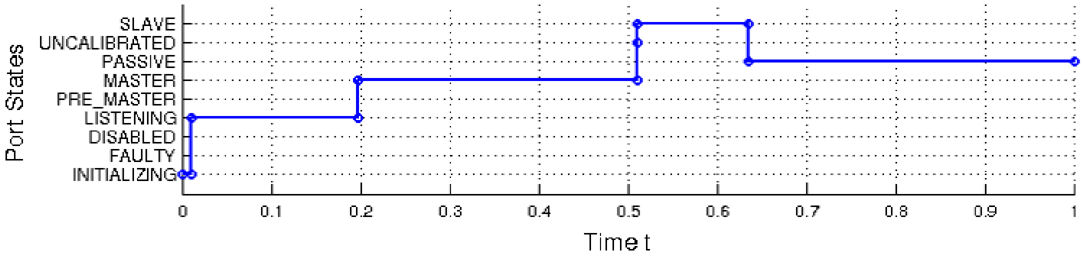
(a) Test case 5 for the BMC algorithm as it is given in [Eid06].

(b) Simulation of the test case 5 network. The tool tip in the screenshot shows the port states of BC 8. Port 2 is the port which is adjacent to BC 7.

Figure 4.2: Theoretical model of a test case for the BMC algorithm and practical execution with our implementation.



(a) State decisions of port 2 of BC 8.



(b) Port states of port 2 of BC 8.

Figure 4.3: Plots of the state decisions and corresponding port states of port 2 of BC 8.

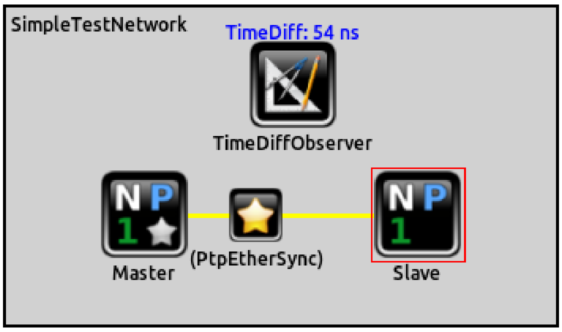


Figure 4.4: Simple PTP network with two OCs, where only one node is master-capable. A `TimeDiffObserver` shows their current time difference. The image also shows a `Sync` message that is in transit from the master to the slave.

Unfortunately, the LinuxPTP heuristic with its default configuration leads to unstable results when used in our simulation with long sync intervals ($logSyncInterval \geqslant 0$).

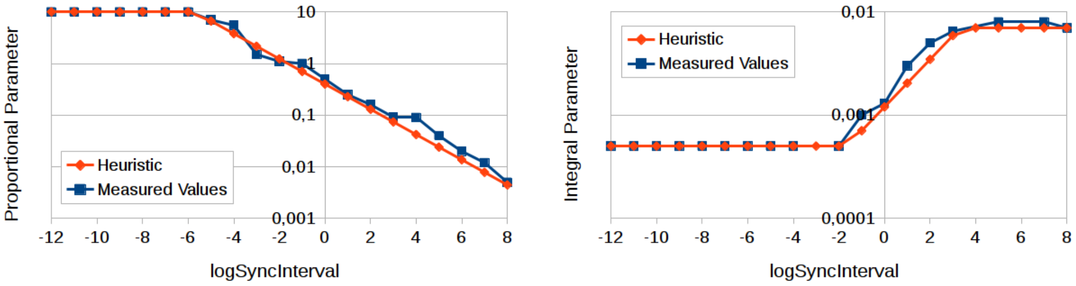
Fortunately, this thesis deals with a simulation framework for PTP which is capable of carrying out parameter studies. We can thus simulate our network with various clock servo configurations, and empirically search for a suitable one in the parameter space. 'Suitable' here means minimizing the mean offset without introducing instabilities. To find our heuristic for the PI clock servo, we will carry out the following steps:

1. Select a sync interval.
2. Set the integral parameter to 0, and carry out a parameter study for the proportional parameter.
3. Select the most suitable proportional parameter from the previous steps.
4. Carry out another parameter study for the integral parameter and again select the most suitable parameter.
5. Repeat steps $1 - 4$ for all sync intervals of interest.
6. Estimate a heuristic for both parameters.

Proportional parameter Figure 4.5a shows the result of the parameter study for the proportional parameter in blue. The proportional parameter increases for decreasing sync intervals, until it reaches a maximum of 10 for a *logSyncInterval* value of -6.

Integral parameter The measurement results for the integral parameter are shown in fig. 4.5b in blue. For long sync intervals, the most suitable values are around 8. When the sync intervals get shorter, the integral parameter decreases until it reaches a minimum of 0.005.

Both parameters scale exponentially with the *logSyncInterval* parameter in certain ranges, and converge to upper and lower bounds otherwise. Thus our heuristic will follow the same approach. The actual configuration values for our heuristic have been manually chosen, so that the heuristic stays slightly below the measured values. Figure 4.5a and fig. 4.5b show the resulting parameter heuristic (in orange) in comparison to the measured values.



(a) Proportional parameter (b) Integral parameter

Figure 4.5: Measurements and heuristic for the proportional and integral parameters.

4.2.2.2 Experiment 3: Parameter study of the synchronization interval

One of the most important configuration options for a PTP network is the configured synchronization interval. It thus seems natural to carry out a parameter study for the *logSyncInterval* parameter. In this experiment, we will synchronize our slave with different sync intervals while it is driven by either a *WatchQuartz* or an *AvgOsc* and compare the results. The configured range for the *logSyncInterval* parameter will be in the range of $-10..8$, which corresponds to $\approx 1\,\mathrm{ms}..256\,\mathrm{s}$.

Figure 4.6 shows the results of this simulation. To improve the confidence in the results, each configuration has been simulated multiple times with different random number seeds. The ADEV plot in fig. 4.6a is based on data from only the first repetition, while the data shown in the mean and jitter plots (fig. 4.6b and fig. 4.6c) is the maximum over all repetitions respectively. Interpreting the results leads to the following conclusions:

- Increasing the synchronization frequency decreases the low frequency noise, until the ADEV converges to an oscillator specific curve. The high frequency parts of the ADEV plots basically stay unmodified. Figure 4.6a only shows the ADEV curves for the fastest and slowest sync interval. The intermediate steps have been left out to keep the figure simple. What is not shown in the figure is that the ADEV curves for *logSyncInterval* $\leqslant -6$ already look like the ones for *logSyncInterval* $= -10$ which are shown in the graph as dashed lines.
- Figure 4.6b shows the maximum of the observed mean offset over several simulation runs. As already expected after analyzing the ADEV plot, *logSyncInterval* values below -6 lead all to similar results. Both oscillators converge to the same mean value, which is in the range of the clock granularity (10^{-7} corresponds to 2 clock ticks for a 20 MHz clock). Above this value it can be seen that decreasing the sync interval also decreases the mean offset from the master. What is noteworthy in this figure is that the *WatchQuartz* actually performs slightly better for the mean offset than the *AvgOsc*.

- The measured jitter[2] values of both oscillators are shown in fig. 4.6c. Increasing the synchronization frequency decreases the jitter for both oscillators, until they reach an oscillator specific minimum. The *AvgOsc* performs much better than the *WatchQuartz* in this measurement, which is as expected, as it suffers from a lower noise level (as shown in the ADEV plot).

As a final conclusion from this experiment, we can say that the clock characteristics improve for shorter synchronization intervals up to a certain value. Further decreasing the synchronization interval will only use more network bandwidth and computation resources, without providing the corresponding synchronization benefits.

Remark: A similar parameter study is given in [GNLS11]. While our results agree qualitatively, the actual numerical results don't match. The reasons for this discrepancy could be manifold, e.g. different clock granularity, filter configuration, clock servo configuration, etc. But as their simulation description is rather coarse, and their implementation is not available to the best knowledge of the author, comparing the results remains difficult.

4.2.2.3 Experiment 4: Path asymmetry

The offset estimation specified by PTP is based on the assumption that the path delay between two nodes is identical in both directions. As this assumption does not always hold in real networks, section 11.6 of [IEE08] specifies *path asymmetry correction*. The following experiment will show the effect of path asymmetry, path asymmetry correction, and how these are handled by *LibPTP*. For this purpose, we simulate the network three times:

1. Without any delay in both PHYs (and thus with a completely symmetric path).
2. With asymmetric delays[3] in each PHY, but without asymmetry correction.
3. With asymmetric delays as in the previous case, but this time with asymmetry correction in the slave.

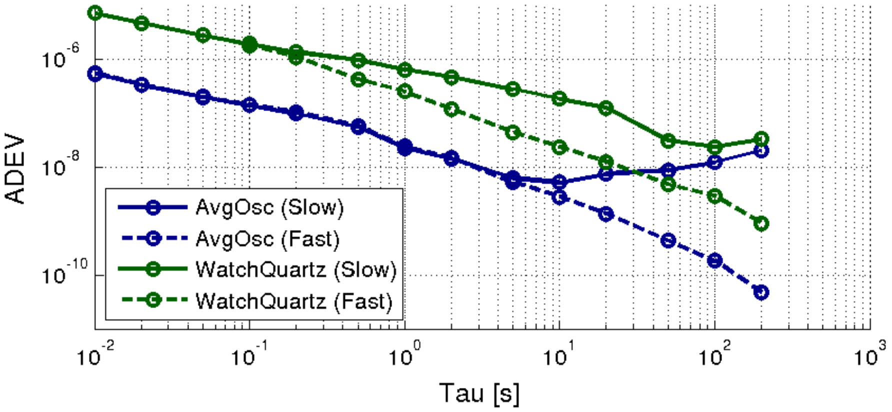
The master will be configured with a receive delay of $400\,\mathrm{ns}$ and a transmit delay of $1400\,\mathrm{ns}$, which means the PHY of the master contributes $1000\,\mathrm{ns}$ to the overall asymmetry. The corresponding delays of the slave's PHY will be configured as $1000\,\mathrm{ns}$ and $600\,\mathrm{ns}$, resulting in an asymmetry of $400\,\mathrm{ns}$. Adding these numbers, we get a total asymmetry of $1400\,\mathrm{ns}$, and expect the offset estimate of the slave to wrong by $700\,\mathrm{ns}$ (half of the path asymmetry). In the final simulation run, the slave's asymmetry correction value will be set to $700\,\mathrm{ns}$.

Figure 4.7a shows the `offsetFromMaster` measurements of the slave from these three simulation runs. As the slave does not realize the path asymmetry, the estimated offset from its perspective is equally low in all three configurations. The fact that the actual offset is different can be seen in fig. 4.7b. The wrong path delay measurement

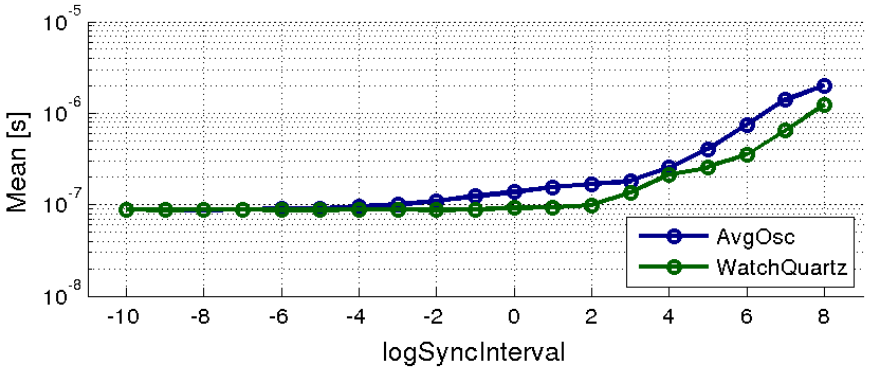
---

[2] *Jitter* in this context means absolute difference between maximum and minimum observed offset.

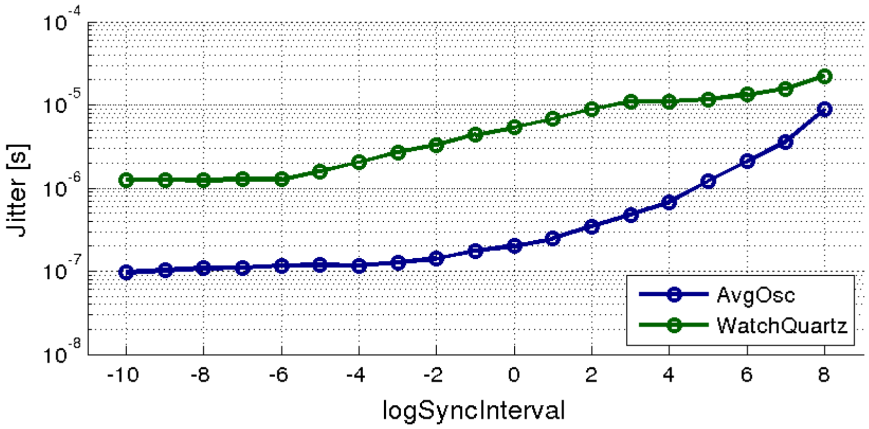
[3] The delay values in this experiment are purely for demonstration purposes, and are not based on any real values from data sheets or similar sources.



(a) Comparison of ADEVs of both oscillators for different sync intervals. Bold lines correspond to a slow sync interval (*logSyncInterval* = 8), dashed lines to a fast sync interval (*logSyncInterval* = -10).



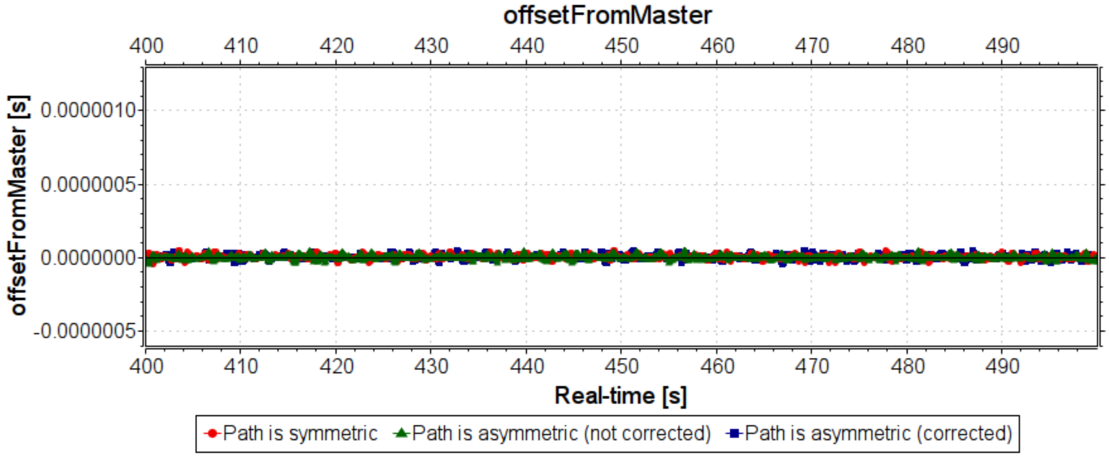
(b) Mean value of the measured offset from the master clock.



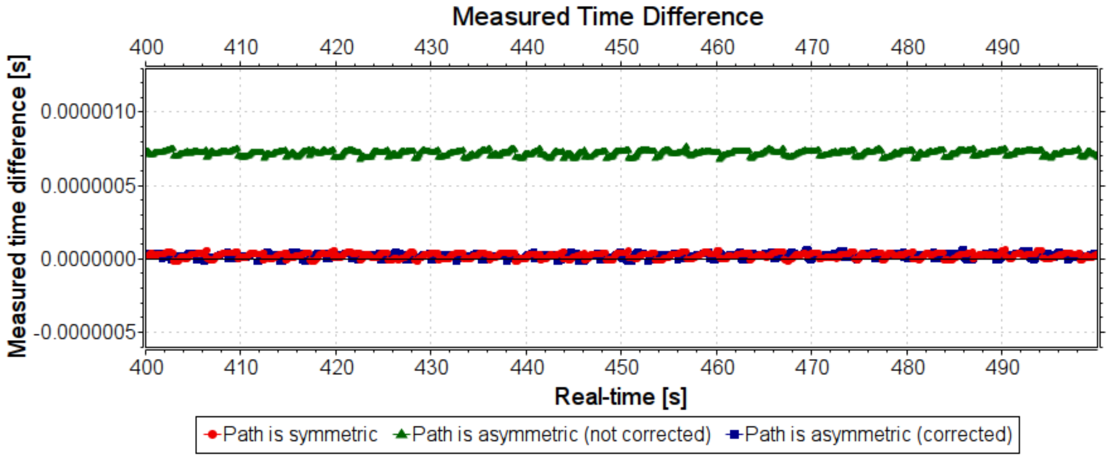
(c) Jitter of the measured offset from the master clock.

Figure 4.6: Different characteristics of both oscillators for various sync intervals.

in the second simulation run leads to a significant offset from the master. Setting the slave's asymmetry correction parameter to a suitable value in the third simulation run leads to similar results as in the first run without any asymmetry.



(a) Measurements of the `offsetFromMaster` value.



(b) Measured clock differences between the master and the slave node.

Figure 4.7: Measurements for different asymmetry configurations.

4.2.3 Daisy chain

This section deals with PTP networks that form a daisy chain. The daisy chain topology is interesting for several reasons:

- In certain domains, networks are often realized using this topology, e.g. in industrial automation.

- Simulations of daisy chains allow us also to draw conclusions about related topologies like stars.

4.2.3.1 Experiment 5: Startup time of BCs

An interesting aspect of PTP networks is the time needed to reach a stable clock hierarchy. If the default BMC algorithm is used in a daisy chain network with BCs, this time might become quite long, as the following experiment shows. If the state decision of a port is *M3*, section 9.2.6.10 of [IEE08] states that the *qualificationTimeoutInterval* shall be the *announceInterval* multiplied by the *currentDS.stepsRemoved* value plus 1. Thus, for each node in a daisy chain the *qualificationTimeoutInterval* gets longer. Let us symbolize the *qualificationTimeoutInterval* of a node n as $T_{qual,n}$ and the *announceInterval* as T_{ann}. It holds then that the total time T_{tot} for these timeouts for N nodes is given as $T_{tot} = \sum_{n=1}^{N} T_{qual,n} = \sum_{n=1}^{N}((n+1)*T_{ann}) = \sum_{n=2}^{N+1}(n*T_{ann}) = (\frac{(N+1)(N+2)}{2} - 1)*T_{ann} = O(N^2)$.

Figure 4.8 shows the port states of a daisy chain consisting of 50 BCs, where the `LogAnnounceInterval` is set to −2. It can be clearly seen that the time the nodes spend in the the `PRE_MASTER` state increases with every hop. For 50 nodes, the above formula leads to $T_{tot} = 331.25\,s$. This value is only a lower bound, but it is a good estimate for how long a given network needs to converge to the final clock hierarchy. This estimate matches the actual result shown in fig. 4.8 quite well.

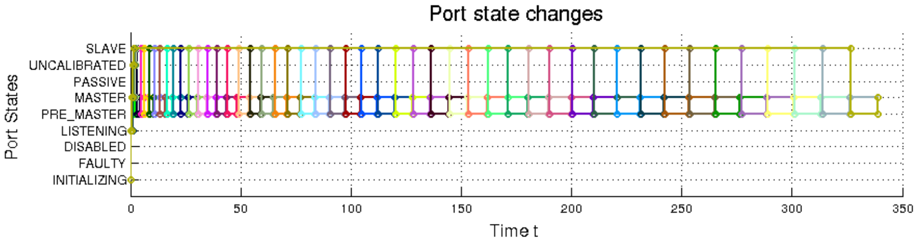


Figure 4.8: Port states of BCs in a daisy chain for the ports which face away from the eventual grandmaster. A legend for this figure was purposely left out to keep the graph simple.

Thus, if fast network convergence is of importance for a given application, the number of BCs that can be placed in a row quickly becomes a limiting factor.

4.2.3.2 Experiment 6: Daisy chain configurations

Motivated by the experiment described in section 4.2.2.2, it would be natural to ask for the influence of the sync interval in a daisy chain. Additionally, it would be interesting to try different kinds of clocks. As already mentioned in section 2.2.8, [IEE08] introduced the concept of TCs to avoid the cascading of control loops in a chain of BCs.

To test the influence of these two parameters, we simulate a daisy chain of 50 clocks driven by an *AvgOsc* connected to a master clock with a perfect reference oscillator. The network will consist either of only BCs or TCs. For this experiment, we will measure the actual offset of every tenth clock from the master.

Figure 4.9 shows the measured jitter, similar to what was shown in fig. 4.6c for a single node. This figure compares the jitter for BCs (blue) and TCs (green) with a distance of $10..50$ hops to the master and for values of *logSyncInterval* in the range of $-6..4$.

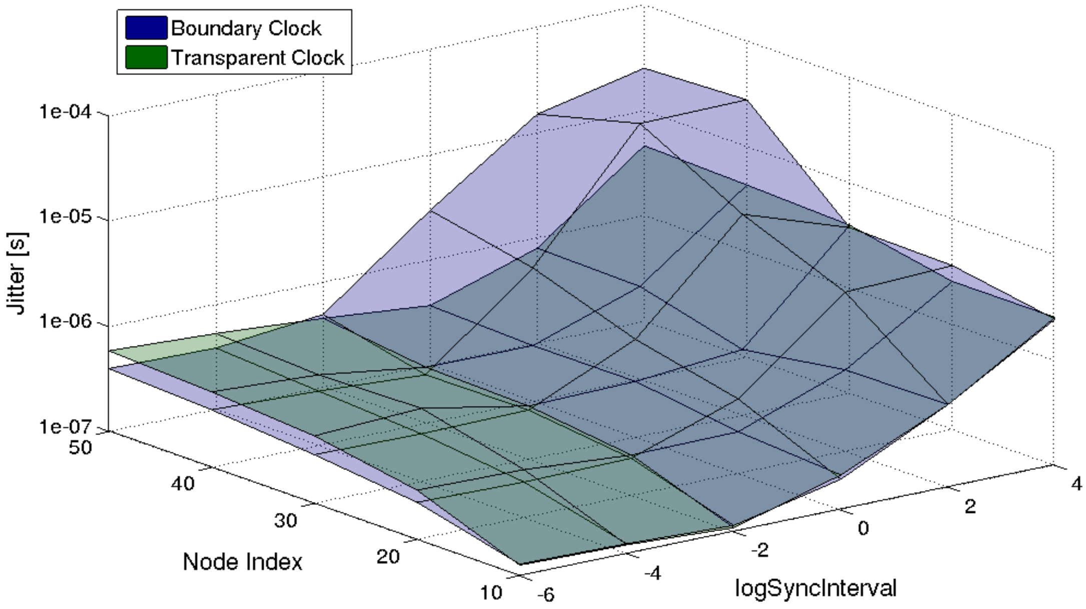


Figure 4.9: Jitter comparison for daisy chains of BCs and TCs for different sync intervals.

For all configurations it holds that a larger distance to the master leads to an increased jitter. This is as one would expect. For long sync intervals (*logSyncInterval* $\geqslant 0$), the comparison of BCs and TCs also behaves as expected: while the jitter increases for both clock types with longer sync intervals, TCs perform much better than BCs. A surprising result is the jitter for values of *logSyncInterval* $\leqslant -2$: At -2, there is hardly any difference between these two types of clocks, and for faster sync intervals BCs actually performed *slightly better* in this experiment!

4.3 Result discussion

The experiments that were discussed in this section increased our confidence in both the plausibility as well as the usefulness of our approach to gain insight in PTP networks.

LibPLN has proven to provide the needed infrastructure to simulate PLN as it is commonly found in oscillators, and the simulation speed is impressive compared to the unmodified batch simulation approach. On the other hand, the simulations with

LibPTP have provided results that were in line with the expectation in some cases, but rather surprising in others:

- For the PI parameter study, simulations with *LibPTP* have proven to be useful as an easy empirical approach to find an adequate heuristic to chose a clock servo configuration.
- As already expected in advance, the jitter of network nodes decreases with an increase in the synchronization frequency. Also expected was that at some oscillator specific point further increasing the number of `Sync` messages would not provide a benefit any more. This was indeed the case for both simulated oscillators.
- One simulation result that was initially surprising, but completely expected in retrospect was the quadratic runtime of the BMC algorithm of BCs in a daisy chain. This could be easily overlooked when reading the specification given in [IEE08], as happened to this author. The circumstance that this behavior showed up in the simulation and turned out to be conforming to the specification speaks again for the plausibility of our implementation.
- A rather unexpected simulation result was that BCs can keep up quite well with TCs in daisy chains for short synchronization intervals. This result was really surprising, and might need further investigation.

CHAPTER 5

Conclusion

Both *LibPLN* and *LibPTP* have shown to be useful, and to provide a way to gain insight in the domains of clock noise and time synchronization via PTP.

5.1 Contributions of this Work

To the knowledge of the thesis author, there were no free software components available to efficiently simulate oscillator noise as well as to simulate PTP networks at the beginning of this thesis. Both of these challenges have been dealt with in the course of this thesis, and the resulting components are publicly available under the terms of the GPL. Thus, anyone interested in time synchronization via PTP can use these components to gain insight into this technology. In particular, the contributions of our work can be summed up as follows:

- *LibPLN* provides a portable and fast implementation of the original approach for the simulation of Powerlaw Noise (PLN) as it was proposed by Kasdin and Walter in [KW92]. In terms of numerical accuracy, it even surpasses the original implementation, especially for RW noise and large sample sizes. *LibPLN* supports different use cases:

  Batch generation of PLN: The original PLN generation approach in [KW92] is a batch method, which generates one large chunk of simulated noise at once. Using various optimizations (as described in section 3.1.5), the noise generation can be made more efficient. By simulating and combining PLNs from oscillators with different frequencies, limitations of the original approach can be worked around. To show how the library can be used to cover this use case, *LibPLN* comes with a small command line utility for batch PLN generation in its `Examples` directory.

  On-the-fly PLN generation for DES environments: By selectively skipping computations when they won't make a noticeable difference to the overall result the original PLN generation approach can be made fit for Discrete Event Simulation (DES) environments. This allows us to implement efficient clocks with realistic noise in OMNeT++. The usage for OMNeT++ simulations was the main motivation for the initial development of *LibPLN*.

- With *LibPTP* we have provided an OMNeT++-based simulation framework for the Precision Time Protocol (PTP), as it is specified in IEEE 1588. IT is based on standard network components from the INET library, and extends them with PTP functionality. *LibPTP* implements most of the IEEE 1588 standard, and thus it can already be used to simulate many different PTP use cases. Extendability was a design goal, and thus *LibPTP* should be able to serve as a useful tool for Design Space Exploration (DSE). Using *LibPTP*, systems designers can simulate the impact of e.g. different PTP implementation options, clock servo designs, network fault behavior, or other PTP relevant attributes.

5.2 Future Work

The contributions of this thesis only solve a part of the overall problem. The following steps would further increase the usefulness of our software:

- **Extending PTP options:** *LibPTP* implements a subset of the possible PTP options. Interesting extensions would be of unicast support, UDP as the transport layer as well as `Management` and `Signaling` messages.
- **Mainlining *LibPTP*:** Currently the *LibPTP* library depends on the INET library, but is developed independently. If the internal structure of the referenced INET models changes, these two libraries could become incompatible. Thus it would be useful to try to mainline at least parts of the functionality back to INET.
- **Extending *LibPLN*:** At the moment *LibPLN* provides only a very limited set of oscillator examples, and it is rather difficult to add additional models. It would be beneficial for its users to simplify the library interface for oscillator specifications, as well as to increase the number of readily available examples.
- **Model additional noise sources:** The current simulation model completely ignores noise sources such as temperature or pressure. To gain more realistic simulation results, it would of course be interesting to add these influences to the simulation model.

APPENDIX A

Acronyms

ADEV	Allan Deviation
API	Application Programming Interface
AVAR	Allan Variance
BC	Boundary Clock
BMC	Best Master Clock
COTS	Commercial off-the-shelf
CM	Compound Module
CSV	Comma-separated values
DES	Discrete Event Simulation
DSE	Design Space Exploration
E2E	End-to-End
FFM	Flicker Frequency Modulation
FFD	Fractional Frequency Deviation
FIR	Finite Impulse Response
FPM	Flicker Phase Modulation
FSA	Frequency Stability Analysis
FFT	Fast Fourier transform
FFTW	Fastest Fourier Transform in the West
GPL	GNU General Public License
GPS	Global Positioning System
GUI	Graphical User Interface
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IIR	Infinite Impulse Response
LLC	Logical Link Control
MAC	Media Access Control
MAVAR	Modified Allan Variance

NED	Network Description
NIC	Network Interface Card
NTP	Network Time Protocol
OC	Ordinary Clock
OCXO	Oven Controlled Crystal Oscillator
OMNeT++	Objective Modular Network Testbed in C++
OS	Operating System
P2P	Peer-to-Peer
PC	Personal computer
PHY	physical layer
PI	proportional-integral
PLN	Powerlaw Noise
PSD	Power Spectral Density
PTP	Precision Time Protocol
RGMII	Reduced Gigabit Media-Independent Interface
RTS	Real-Time System
RW	Random Walk
SM	Simple Module
TC	Transparent Clock
TD	Time Deviation
TDG	TD Generator
TDO	TD Oracle
TDV	TD Vector
TDVG	TDV Generator
TDVS	TDV Storage
TDEC	TDE Chain
TDE	TD Estimator
UML	Unified Modeling Language
UDP	User Datagram Protocol
WFM	White Frequency Modulation
WPM	White Phase Modulation

Bibliography

[AAH00] David W. Allan, Neil Ashby, and Clifford C. Hodge. *Application Note 1289: The Science of Timekeeping*. Agilent Technologies, 2000.

[All66] David W. Allan. Statistics of Atomic Frequency Standards. *Proceedings of the IEEE*, 54(2):221–230, February 1966.

[All87] David W. Allan. Time and Frequency (Time-Domain) Characterization, Estimation, and Prediction of Precision Clock Oscillators. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, UFFC-34(6):647–654, November 1987.

[DFF+07] Alessandro Depari, Paolo Ferrari, Alessandra Flamminni, Danielle Marioli, and Andrea Taroni. Evaluation of Timing Characteristics of Industrial Ethernet Networks Synchronized by means of IEEE 1588. *Instrumentation and Measurement Technology Conference Proceedings*, May 2007.

[Eid06] John C. Eidson. *Measurement, Control, and Communication Using IEEE 1588*. Springer, 2006.

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994.

[GLN+07] Georg Gaderer, Patrick Loschmidt, Anetta Nagy, Roman Beigelbeck, Jörgen Mad, and Nikolaus Kerö. An Oscillator Model for High-Precision Synchronization Protocol Discrete Event Simulation. *Proceedings of the 39th Annual Precise Time and Time Interval Meeting*, pages 363–370, November 2007.

[GNLK08] Georg Gaderer, Anetta Nagy, Patrick Loschmidt, and Nikolaus Kerö. A Novel, High Resolution Oscillator Model for DES Systems. *Proceedings of the Frequency Control Symposium*, pages 178–183, May 2008.

[GNLS11] Georg Gaderer, Anetta Nagy, Patrick Loschmidt, and Thilo Sauter. Achieving a Realistic Notion of Time in Discrete Event Simulation. *International Journal of Distributed Sensor Networks*, 2011, July 2011.

[HAB81] D.A. Howe, D.W. Allan, and J.A. Barnes. Properties of Signal Sources and Measurement Methods. In *Thirty Fifth Annual Frequency Control Symposium. 1981*, pages 669–716, May 1981.

[Hor74] L.L. Horowitz. The effects of spline interpolation on power spectral density. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 22(1):22–27, Feb 1974.

[IEE02] IEEE: The Institute of Electrical and Electronics Engineers, Inc., The Institute of Electrical and Electronics Enginee, 3 Park Avenue, New York, NY 10016-5997, USA. *IEEE Std 1588-2002: IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, November 2002.

[IEE08] IEEE: The Institute of Electrical and Electronics Engineers, Inc., The Institute of Electrical and Electronics Enginee, 3 Park Avenue, New York, NY 10016-5997, USA. *IEEE Std 1588-2008: IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, July 2008.

[IEE09] IEEE: The Institute of Electrical and Electronics Engineers, Inc., The Institute of Electrical and Electronics Enginee, 3 Park Avenue, New York, NY 10016-5997, USA. *IEEE Std 1139-2008: IEEE Standard Definitions of Physical Quantities for Fundamental Frequency and Time Metrology – Random Instabilities*, February 2009.

[Int12] Intel Corporation. *Intel Ethernet Controller I210 Datasheet*, 2012.

[Kas95] N. Jeremy Kasdin. Discrete Simulation of Colored Noise and Stochastic Processes and $1/f^\alpha$ Power Law Noise Generation. *Proceedings of the IEEE*, 83(5), May 1995.

[Kop11] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer Publishing Company, Incorporated, 2nd edition, 2011.

[KW92] N. Jeremy Kasdin and Todd Walter. Discrete Simulation of Power Law noise. *Proceedings of the 1992 IEEE Frequency Control Symposium*, May 1992.

[Lai12] Jeff Laird. *Clock Synchronization Terminology*. InterOperability Labority, University of New Hampshire, June 2012.

[Lom08] Michael Lombardi. The Accuracy and Stability of Quartz Watches. *Horological Journal*, February 2008.

[LY11] Yingshu Liu and Cheng Yang. OMNeT++ Based Modeling and Simulation of the IEEE 1588 PTP Clock. *Proceedings of the International Conference on Electrical and Control Engineering*, pages 4602–4605, September 2011.

[Mar06] Marvell Technology Group Ltd. *88E1111 Datasheet - Integrated 10/100/1000 Ultra Gigabit Ethernet Transceiver*, October 2006.

[Ope14] OpenSim Ltd., Budapest, Hungary. *OMNeT++ Manual*, July 2014.

[PGGS07] Fritz Praus, Wolfgang Granzer, Georg Gaderer, and Thilo Sauter. A Simulation Framework for Fault-Tolerant Clock Synchronization in Industrial

Automation Networks . *IEEE Conference on Emerging Technologies and Factory Automation*, pages 1465–1472, September 2007.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C (2Nd Ed.): The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.

[Ram12] F. Ramian. *Application Note: Time Domain Oscillator Statiblity Measurement Allan variance*. Rohde & Schwarz, February 2012.

[RGNL10] Felix Ring, Georg Gaderer, Anetta Nagy, and Patrick Loschmidt. Distributed Clock Synchronization in Discrete Event Simulators for Wireless Factory Automation. *IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, pages 103–108, September 2010.

[Ril08] William J. Riley. *NIST Special Publication 1065: Handbook of Frequency Stability Analysis*. National Institute of Standards and Technology, U.S. Department of Commerce, July 2008.

[RMR14] C. Riesch, C. Marinescu, and M. Rudigier. Experimental verification of the egress and ingress latency correction in PTP clocks. In *Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS), 2014 IEEE International Symposium on*, pages 59–64, Sept 2014.

[Rub05] Enrico Rubiola. The Leeson effect - Phase noise in quasilinear oscillators. *ArXiv Physics e-prints*, February 2005.

[Sch12] Hanspeter Schmid. *How to use the FFT and Matlab's pwelch function for signal and noise simulations and measurements*. Institute of Microelectronics, University of Applied Sciences Northwestern Switzerland, August 2012.

[Smi97] S.W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Pub., 1997.

[Ste12] Jens Steinhauser. A PTP Implementation in OMNET++. Bachlor's thesis, Faculty of Informatics, Vienna University of Technology, 2012.

[Wal94] Todd Walter. Characterizing Frequency Stability: A Continuous Power-Law Model with Discrete Sampling. *IEEE Transactions on Instrumentation and Measurement*, 43(1):69–79, February 1994.