Masterarbeit

Evaluation and Implementation of Time-Synchronization for Distributed Systems

Martin Kammerhofer

Institut für Technische Informatik
Technische Universität Graz
Vorstand: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß

Begutachter: Univ.-Prof. Dipl.-Ing. Dr. techn. Bernhard Rinner
Betreuer: Dipl.-Ing. Dr. techn. Allan Tengg

Graz, im Oktober 2009

Abstract

Time synchronization of geographically dispersed clocks is a classical problem. An important application is synchronization of distributed computer systems. The problem has been extensively studied for the last 30 years and a large number of algorithms and protocols has been proposed.

This text structures the problem by decomposing it into three partial problems: Precise measurement of the difference between the readings of remote clocks and a local clock, estimation and extrapolation of inner states of the involved clocks, and continuous or periodic adjustment of the local clock to improve future synchronization. Approaches to these partial problems from metrology, mathematics/statistics, and control theory are outlined in a theoretical part and their use in synchronization protocols is described.

The practical part discusses design and implementation of a time synchronization service for the I-SENSE project. I-SENSE is an intelligent multi-sensor multi-level data-fusion framework for distributed embedded systems. Synchronized clocks are mainly required for temporal ordering of video frames from two or more cameras. An evaluation of the implementation on Pentium M and TMS320C64X processors is presented and potential further improvements are discussed.

Kurzfassung

Die Synchronisation von geographisch verteilten Uhren ist ein klassisches Problem. Eine wichtige Anwendung ist die Synchronisation von verteilten Computersystemen. Zu diesem Problem gibt es sehr umfangreiche Literatur aus den letzten 30 Jahren in der zahlreiche Algorithmen und Protokolle beschrieben werden.

In dieser Masterarbeit wird das Problem auf die folgenden drei Teilprobleme zurückgeführt: Präziser Vergleich des Uhrenstandes einer lokalen Uhr mit entfernten Uhren, Abschätzung und Extrapolation der Zustandsvariablen der beteiligten Uhren und Verbesserung der Synchronisation durch Regelung der lokalen Uhr. Lösungsansätze zu diesen Teilproblemen kommen aus den Disziplinen Metrologie, Mathematik/Statistik und Regelungstechnik. Sie werden im Grundlagen-Teil zusammengefasst dargestellt und ihre Anwendung in Synchronisations-Protokollen wird gezeigt.

Im praktischen Teil wurde für das I-SENSE Projekt ein Zeitsynchronisationsdienst entworfen und implementiert. I-SENSE ist ein intelligentes Multi-Sensor Fusion Framework für verteilte eingebettete Systeme. Synchronisierte Uhren werden hauptsächlich für das zeitlich korrekte Kombinieren von Einzelbildern verschiedener Videokameras benötigt. Eine Beschreibung und Evaluation der Implementierung auf Pentium M und TMS320C64X Prozessoren wird präsentiert und potentielle weitere Verbesserungen werden diskutiert.

Statutory Declaration

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

Eidesstattliche Erklärung

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

Ort	Datum	Unterschrift

Danksagung

Diese Masterarbeit wurde im Jahr 2008 am Institut für Technische Informatik an der Technischen Universität Graz durchgeführt.

Zu Beginn möchte ich mich bei meinem Betreuer Allan Tengg für seine tatkräftige und kontinuierliche Unterstützung und die konstruktive Zusammenarbeit bedanken. Weiters danke ich meinem Begutachter Herrn Prof. Dr. Bernhard Rinner für seine präzisen Verbesserungsvorschläge und seine Geduld.

An dieser Stelle auch ein herzliches Dankeschön an meine Eltern, deren Verdienste aufzuzählen den Rahmen dieser Seite bei weitem sprengen würde. Ganz besonderer Dank gebührt meiner Partnerin Susanne für den Rückhalt den sie mir gegeben hat und für alles was sie mir abgenommen hat bzw. worauf sie verzichten musste, wenn ich an der Tastatur saß.

Ich widme diese Masterarbeit unserer Tochter Nika, die im Zeitraum des Verfassens geboren wurde.

Graz, im Oktober 2009 Martin Kammerhofer

FINAL

FOR PUBLIC RELEASE

Contents

List of Figures

List of Tables

1. Introduction

Most distributed tasks require some sort of synchronization. A straightforward and intuitive way of supplying synchronization is *time synchronization*. Synchronized clocks have many uses in distributed systems. They simplify many distributed algorithms and improve their performance [Lis93].

Although time synchronization is a classical problem, there is no *general* solution. The literature on the subject is vast—there are thousands of publications and research is still ongoing.[1]

Among the reasons for the abundance in literature and protocols are the strong dependency of time synchronization implementations on specific properties of the hardware and software environment, and the tremendously varying demands of individual applications. The focus on a specific use case—time synchronization for the I-SENSE middleware—will therefore narrow down the broad path that is implied by the title of this thesis.

This chapter will motivate why the I-SENSE framework needs a time synchronization implementation in section 1.1, set the goals for this thesis in section 1.2, and sketch the structure of this text in section 1.3.

1.1. Motivation

The major goal of the I-SENSE research project[2] is a scalable and embedded architecture for various multi-sensor applications [KRT06, TKR07, KTR08]. The project combines the scientific research areas *multi-sensor data fusion* and pervasive embedded computing. The main idea is to provide a generic architecture, which supports distributed realtime multi-level data fusion on an embedded system.

The architecture of an I-SENSE sensor node is depicted in fig. 1.1 and a photo is shown in fig. 4.1 on page 48.

Distributed fusion applications are described by a *fusion model*. The fusion model can be represented as a weighted directed acyclic task graph. An example is shown in fig. 1.2. Several fusion tasks f_i process data from different sensors. Synchronized timestamps are a prerequisite for this multi-sensor data fusion.[3]

[1] A March 2009 CiteSeer query for "time synchronization" brought up 3046 results. A Google Scholar query for "synchronization protocol" yielded 6190 articles. A significant part of the third millennium research is in the context of (wireless) sensor networks.

[2] `http://www.iti.tugraz.at/en/research/isense/index.html`

[3] More exactly, while multi-sensor data fusion without time synchronization is doable, it requires more complex correlation algorithms, more CPU and memory resources, and is therefore avoided.

As a case study for the I-SENSE approach, a traffic surveillance system has been developed. Video cameras, microphones and light barriers are deployed as sensors. Sensor fusion is used to achieve vehicle detection, tracking and classification. Video stream fusion requires temporal alignment at the frame level. If any two clocks that generate frame timestamps differ by an offset less than

$$\text{offset} < \frac{1}{2 * \text{highest frame rate}} \tag{1.1}$$

then matching of frames can be done in straightforward ways.[4][DFH+08] With practical video frame rates, equation 1.1 translates to a requirement of a few milliseconds for the upper clock error bound.

An ideal clock would proceed at a rate of 1 second per second of standard time. Practical clocks are imperfect and deviate from this ideal rate. Crystal oscillator based computer clocks show manufacturing dependent frequency tolerances, frequency deviations that depend on temperature, and frequency aging effects. Unattended computer clocks therefore drift apart. Fig. 1.3 shows the clock skew of a computer clock. The depicted clock is rather good.[5] Frequency errors of a few hundred ppm are not uncommon in cheap crystals oscillators for computer clocks [MD08]. Moreover, temperature dependent clock rate variations can be expected to be much larger in outdoor deployed embedded systems. A constant clock skew of only 1 ppm accumulates a clock offset error of 86,4 ms over a day.

Time synchronization for the I-SENSE framework—although planned from the beginning of the project—was still unimplemented prior to this thesis. Only an intra-node pre-synchronization feature was available, i. e., the clocks of all processors within a sensor node (cf. fig. 1.1) were synchronized during middleware startup, and pre-configured static rate corrections were applied. This was sufficient for development and testing, but would not have worked over extended periods of time.

1.2. Objective

The goal of this thesis is to extend the functionality of the I-SENSE framework with a mechanism for time synchronization. This extension must perform adequately, to allow fusion of two real time video streams originating from different sensor nodes. The following subgoals are defined:

1. Investigation of existing procedures for time synchronization in distributed systems
2. Selection of a suitable mechanism or protocol
3. Implementation of the chosen solution on the Windows XP embedded (Pentium-M) and DSP/BIOS (TMS320C64X) platforms
4. Evaluation of the implementation

To have some margin for higher frame rates, the aim for maximum clock offset error is 5 ms.[6]

[4] This assumes that creation of timestamps is perfect, i. e., reading the clocks does not introduce additional time offset errors.

[5] The machine is `fitipc150`, a commodity PC located in the ITI VLSI laboratory, which happened to be the main development machine for the practical part of this thesis.

[6] The 5 ms value follows from an assumed maximal frame rate of 100 Hz inserted into equation 1.1. The current traffic surveillance use case performs well with frame rates of only 15. . . 30 Hz.

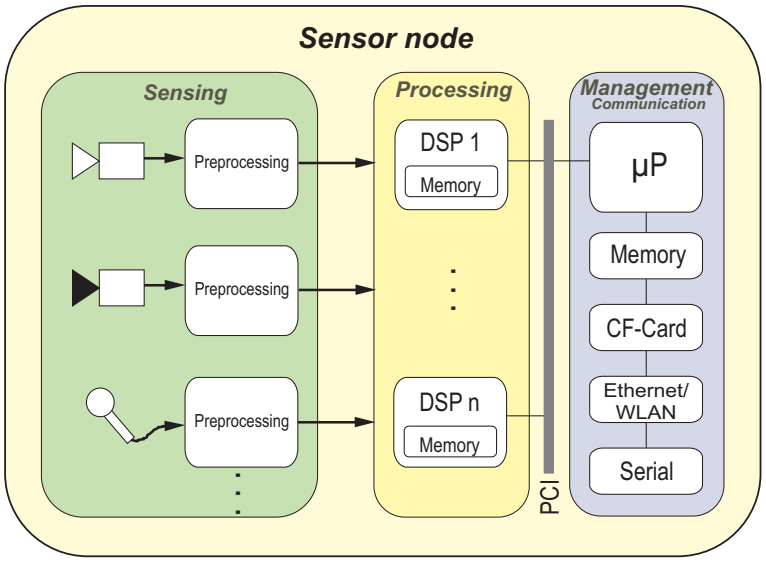


Figure 1.1.: I-SENSE node hardware architecture

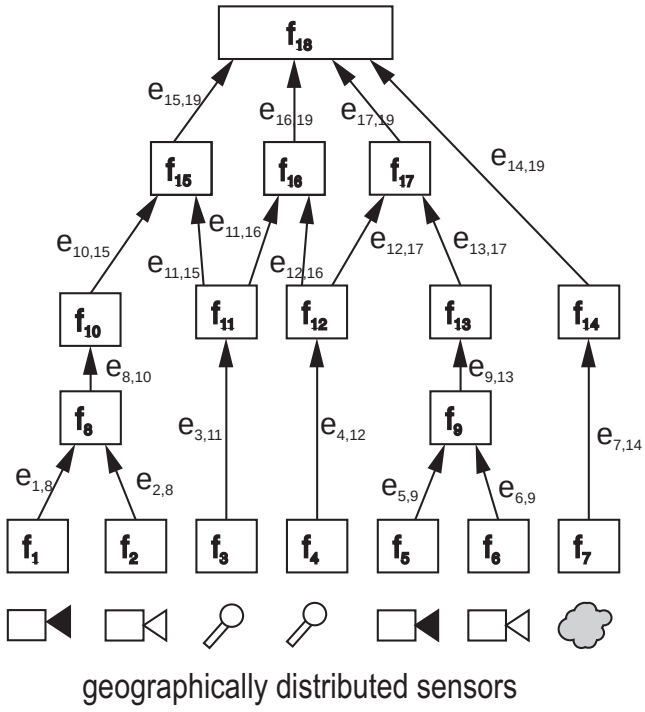


Figure 1.2.: A simple fusion model [TKR07]

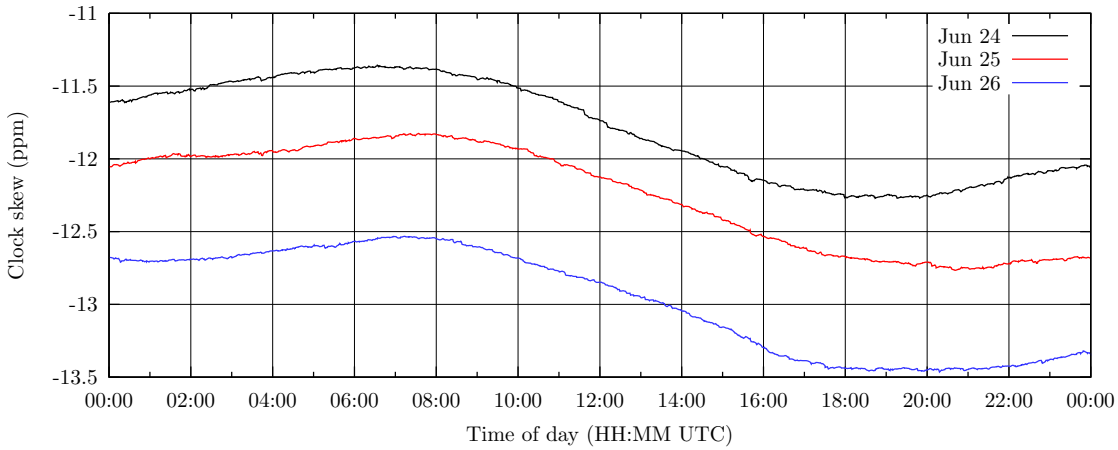


Figure 1.3.: Diurnal frequency variation of a computer clock over three consecutive days

1.3. Thesis Outline

Chapter 2 gives a thorough introduction into the problem of time synchronization. Important concepts like timescales, clocks, measurement of remote clocks, and synchronization methods are explained. Most of the chapter treats synchronization from a high level perspective, but details pertaining to commonly used operating systems and current PC hardware are given too.

Chapter 3 explores the huge subject of synchronization protocols for distributed computer systems. First a discussion of several important classification criteria explains, why the design space for protocols is so large. The following protocol survey presents some key concepts of important contributions to the field. For reasons of space, only the NTP and SNTP protocols are described in detail, but references to important papers and survey literature are given.

Chapter 4 first describes aspects of the I-SENSE platform, which are relevant for time synchronization implementation. The high level structure of the implementation is then explained with reference to four important design decisions. Several object oriented clock abstractions are presented next, on which the implementation is built. An implementation overview concludes the chapter.

Evaluation was somewhat constricted by lack of resources.[7] However several experiments have been conducted, which yielded very satisfying results. The setup and methodology of these experiments is presented together with graphical results in chapter 5.

Chapter 6 concludes the thesis with a summary and suggestions for future work.

[7] Hardware timestamping facilities and embedded I-SENSE nodes.

2. Background and Terminology

2.1. Time, Clocks and Timescales

2.1.1. Properties of time

Time is the physical quantity that can be measured with the highest accuracy; yet still there is no satisfactory answer to the (philosophical and scientific) question "What is time?" [Sch94b]. We cannot study the flow of time under a microscope, impede it or experiment with it. We do not know what exactly happens when time passes. Important for our purposes are, that

- time differences can be measured with clocks, and
- time defines the (temporal) order of events.

2.1.2. Relativistic effects

Time is neither absolute nor independent from space. According to the theory of special relativity it is possible, that different observers, even after correcting for propagation delays, find different orderings for the same set of events.[1] Time dilatation depends on the ratio of the relative speed between clocks v to the speed of light c as $[1-(v/c)^2]^{-1/2}$ (Lorentz factor). A consequence of general relativity is, that time flows slower in higher gravitational fields. Gravitational time dilatation makes a clock gain 9.4 ns per day, when lifted up 1 km from sea level.[2]

The magnitudes of relativistic effects are very small at everyday live speeds ($v \ll c$) and locations near the earth surface. The timing uncertainty of computer clocks is several orders of magnitude larger than these effects. Newtonian space-time is therefore assumed for the rest of this thesis.

2.1.3. The second

Time interval is one of seven base quantities of the International System of Units (SI) [BIP06]. The unit of time interval is the second.

Historically the second has been defined as the fraction $1/86400$ of the mean solar day. Because the spin rate of the earth is irregular on short time scales and decreasing on long time scales,[3] from

[1] This is known as relativity of simultaneity.

[2] At latitude 40° the clock increases its rate by $1.091 \times 10^{-16}\,\mathrm{m}^{-1}$ [Vig07, p. 8-22].

[3] The long-term average rate of increase in the length of the day is about 1.7 ms per century. Immanuel Kant suggested as early as 1754 a steady deceleration of earth rotation due to tidal friction [NMM+01].

1960 to 1967 the SI second was defined as a certain fraction of the tropical year 1900 (ephemeris second). Since 1967 the definition of the second is completely decoupled from astrometry. The current definition of the SI second is derived from an atomic resonance of the ^{133}Cs (caesium) atom:[4]

> The second is the duration of 9 192 631 770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the caesium 133 atom.
>
> This definition refers to a caesium atom at rest at a temperature of zero Kelvin [BIP06].

Today a small number of national metrology laboratories realize the unit of time through primary frequency standards. The best of these primary standards produce the SI second with a relative standard uncertainty of some parts in 10^{16} [BIP07].

2.1.4. Timescales

Clocks only count intervals. To agree on *dates*, timescales are needed. A *timescale* is a system of assigning labels (dates) to *time instants* (events). (The ISO 8601 standard specifies numeric representations of date and time.) The origin (reference date) of a timescale is called the *epoch*. A (global) timescale should have the following properties [Ari05]:

Reliability Clocks have a large MTBF. The timescale tolerates failure of clocks, i. e., there are redundant clocks.

Frequency stability The unit of scale is constant. Two time scale readings determine the length of a time interval.

Frequency accuracy The unit of scale is as close as possible to its definition.

Accessibility The timescale is universally accepted. It provides a way to date events for everyone.

Time is an immaterial quantity. Measurement of time is based on physical phenomena that depend on time. There are two ways to get a timescale from a physical phenomenon.

Dynamic timescale A dynamical physical system is observed. The system has a mathematical model in which time is the independent parameter. The model allows, given an observed state of the system, to determine unambiguously the time of the observation. Particular states of the system (events) serve as labels on the timescale. The unit of time is conveniently defined e. g., as a time constant of the mathematical model or the interval between observable periodic events. The *Universal Time* family of timescales (UT0, UT1, UT1R, UT2 and UT2R) are dynamic.

Integrative timescale The unit of time is a time interval defined by a reproducible physical phenomenon. The timescale continuously accumulates (counts) the units of time.[5] Other than a well defined unit, a integrative timescale needs a convention about a fixed origin. All atomic timescales are integrative.

[4] When better frequency standards than caesium atomic clocks become widely available, the definition of the second will certainly change again.

[5] A realization of an integrative timescale accumulates errors in the realization of the unit too.

Today global timescales are realized through international cooperation. In 1912 the Bureau International de l'Heure (BIH) was founded. Its responsibility for combining different time measurements was taken over by the Bureau International des Poids et Mesures (BIPM) in 1987.

International atomic time (TAI)

National laboratories usually operate a number of atomic clocks. These clocks run independently. Based on the results of local comparisons between these clocks a combined laboratory timescale is created. The combined timescale is (usually) more accurate and stable than any of the contributing clocks. These timescales are designated TA(k) for laboratory k.

The BIPM uses comparisons between some 300 atomic clocks in about 60 national time laboratories to form International Atomic Time (TAI). The details of the algorithms used to compute TAI are quite complex and have been changed several times in the past [GA05].

Methods of comparison of distant clocks are a prime requisite to calculate TAI. Frequency and time transfers are made with GPS and by two-way satellite links. The uncertainty of clock comparison is today between a few tens of nanoseconds and a nanosecond for the best links [GA05].

TAI is a "paper clock" not available in real time. TAI and UTC are disseminated every month by Circular T, a monthly publication of the BIPM.

Version 4 of the Network Time Protocol (NTPv4) can be configured to disseminate the TAI – UTC offset in addition to UTC [LM00].

Universal time (UT)

Universal time (UT1) is based on the rotation angle of the earth on its axis relative to the mean sun. It is popularly, but erroneously, known as Greenwich Mean Time (GMT). The rotation of the Earth and UT1 are now monitored by the International Earth Rotation Service (IERS).[6] Modern techniques like *Very Large Baseline Interferometry* (VLBI) allow the determination of UT1 with an uncertainty of 10 µs [GA05].

UT1 has applications in astronomy, geodesy, space navigation and satellite tracking; but it is nowadays of little importance for the general public.

Coordinated universal time (UTC)

UTC is today the basis for almost all official national timescales and therefore widely available. Since 1 January 1972 00:00:00 UTC the present system is in use. The UTC second ticks synchronously with the TAI second. Leap seconds are infrequently added to the UTC timescale to keep the absolute value of dUT1 = UT1 – UTC below 0.9 s.[7] UTC – TAI is -34 s at the time of this writing. Scheduled

[6] Knowing the universal time of a sextant sighting was historically very important for determining the longitude. A timing error of 1 s leads to an error in longitude of 15 arc-seconds, i. e., 463 m on the equator.

[7] In theory leap seconds can also be removed from the UTC timescale. This has never happened, and—according to present knowledge about earth rotation—it is unlikely that it will ever happen.

Date	UTC Time	UTC – TAI [s]	POSIX time [s]	dUT1 [s]
2008 December 31	23h 59m 59s	-33	1230767999	-0.592841
2008 December 31	23h 59m 60s	-33	1230768000	-0.592841
2009 January 1	0h 0m 0s	-34	1230768000	0.407159

Table 2.1.: Positive leap second

insertions of leap seconds are announced several months before the fact through the biannual IERS "Bulletin C".

Insertion of a 61st second into the last minute of the year 2008 is shown in table 2.1. Removal of a second from UTC, i. e., a minute with only 59 seconds, is very unlikely to ever happen.

Because of the problems associated with leap seconds, a new definition of UTC is being discussed. [NMM+01] Several international scientific organizations are currently evaluating the subject. The U. S. submitted a proposition to abandon leap seconds (and replace them with leap hours) to the ITU-R in 2004. No decision will will be made before 2011.

Representation of UTC in POSIX and Windows

Most operating systems cannot fully cope with leap seconds. Operating systems represent timestamps internally as an integer count of (a constant fraction of) seconds. Structured representations—seconds, minutes, hours and so on—are used only for input and output and are conveniently interpreted with reference to the local time zone, with or without daylight saving time. Two assumptions are built in:[8]

1. Timestamps are to be interpreted as the number of seconds between the specified time and the epoch.
2. Each day (since the epoch) has a duration of exactly 86400 seconds.

These assumptions about the timescale contradict the definition of UTC.

The rationale for assumption 2 is to keep algorithmic conversions between the internal scalar representation and the broken down representation simple. Otherwise, a faithful implementation of the conversions would incur the overhead of consulting a leap second table.

It is impossible to implement a uniform and continuous UTC timescale based on the above assumptions. Operating systems with UTC clocks have to handle leap seconds somehow. Some options for positive leap second handling are:

1. ignore leap seconds
2. jump back 1 s at the beginning of the leap second
3. jump back 1 s at the end of the leap second

[8] Specified e. g., in POSIX.1/IEEE 1003.1-1996 and handled the same way in Microsoft Windows operating systems.

t_A [s]	t_B [s]	$\lvert t_A - t_B \rvert$ [s]
$L+0.5$	$L+0.6$	0.1
$L+0.5$	$L+1.6$	1.1
$L+1.5$	$L+0.6$	0.9
$L+1.5$	$L+1.6$	0.1

Table 2.2.: Temporal orderings of ambiguous timestamps

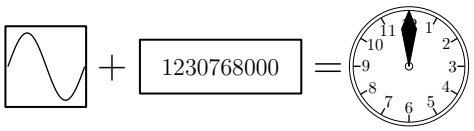


Figure 2.1.: Oscillator + Counter = Clock

4. stop the clock during the leap second
5. reduce the clock frequency in an interval around the leap second to gradually loose one second.[9]

POSIX as well as Windows use method 1 if no time service is configured, i. e., they loose synchronization to UTC after the leap second.[10] Method 3 is used when operating as an (S)NTP client and/or server. Stepping back the clock violates the fundamental assumption of monotonic increasing time.[11] Clock steps introduce discontinuities to the timescale. A leap second inserted at time L causes ambiguous timestamp during the interval $[L, L+2]$. As an example consider two sensor events A and B with scalar timestamps $L+0.5\,\mathrm{s}$ and $L+0.6\,\mathrm{s}$, respectively. Table 2.2 lists the possible temporal orderings. In general relative errors in measured intervals are unbounded and sign inversions can happen!

2.2. Characterization of Clocks

Any time measurement device must somehow realize the second. Practical clocks use an oscillating device, to determine (a constant fraction of) the second, and a counter to accumulate these time intervals (cf. fig 2.1).[12] The vast majority of computer clocks uses crystal oscillators. Embedded systems use ceramic resonators and RC-oscillators too.

2.2.1. Mathematical models of oscillators and clocks

The instantaneous output voltage $v(t)$ of an oscillator is

$$v(t) = [V_0 + \Delta V(t)]\cos[2\pi\nu_0 t + \varphi(t)] \tag{2.1}$$

[9] This was proposed as UTC-SLS (UTC with Smoothed Leap Seconds).
[10] Provided the machine in question was previously synchronized by other means, e. g., with a one-shot clock adjustment program like the `ntpdate` utility.
[11] It breaks Lamport's *Happened-Before* relation described in section 3.2.1 on page 33.
[12] Other types of clocks exist, but play no practical role in time measurement.

where constants V_0 and ν_0 represent nominal amplitude and frequency, respectively. Amplitude fluctuations can be converted to phase fluctuations, but quality oscillators usually have small fractional amplitude fluctuations that are neglected.

$$\frac{\Delta V(t)}{V_0} \ll 1 \qquad \text{and therefore } \Delta V(t) \equiv 0 \text{ is assumed} \tag{2.2}$$

The instantaneous frequency $\nu(t)$ is the derivative of the phase.

$$\nu(t) = \nu_0 + \frac{1}{2\pi}\,\frac{\mathrm{d}\varphi(t)}{\mathrm{d}t} \tag{2.3}$$

The dimensionless instantaneous fractional frequency deviation $y(t)$ is defined as

$$y(t) \;=\; \frac{\Delta f}{f} = \frac{\nu(t)-\nu_0}{\nu_0} = \frac{1}{2\pi\nu_0}\,\frac{\mathrm{d}\varphi(t)}{\mathrm{d}t} = \frac{\mathrm{d}x(t)}{\mathrm{d}t} \tag{2.4}$$

Measuring instantaneous frequency is impossible, because it would require measurement equipment with infinite bandwidth. Frequency measurement *always* involves *two* oscillators and some sampling/averaging time τ.[13] In practice fractional frequency deviation is averaged as

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

When we consider an oscillator as a clock, the fractional phase fluctuation $x(t)$ represents the accumulated *time error* over the interval $[0,\, t]$ [14]

$$x(t) = \int_0^t y(t')\,\mathrm{d}t' = \frac{\varphi(t)}{2\pi\nu_0} \tag{2.6}$$

There are many ways how an oscillator can be interfaced to a counter to build a clock; hence the counter frequency can be different from the oscillator frequency. Analog (PLL) and digital techniques (prescaler, adder) are used for clock rate correction [Loy97, p. 30–34]. Most computer clocks either count whole oscillator cycles (e. g., a PCC) or divide the oscillator frequency by a constant factor with a fixed or programmable prescaler.

Almost all authors model clocks as *continuous* monotonic functions $C(t)$ that map from real time to *clock time*.[15] To make the distinction between real time and clock readings clear, capital letters are used for timestamps.

$$T_i = C(t_i) \tag{2.7}$$

The rate R of a clock is the first derivative of the clock function

$$R(t) = \frac{\mathrm{d}C(t)}{\mathrm{d}t} \tag{2.8}$$

[13] The sampling time τ could e. g., be the gate time of a counter.
[14] Some authors use $x(t)$ to denote the *random* part of time error only.
[15] Also known as *virtual time* or *logical time* in the literature.

A perfect clock would have no rate deviations, i. e., $R(t) \equiv 1$. A clock with $R > 1$ is called *fast* or *early*, and it is said to *gain* time. A clock with $R < 1$ is *slow*, *late*, and *looses* time. The rate is modeled as

$$R(t) = 1 + S + Dt + E_i(t) + \upsilon(t) \tag{2.9}$$

where S is the fractional frequency offset or *skew*, D is the linear fractional frequency *drift* rate due to aging, $E_i(t)$ is the frequency dependency on environmental conditions (temperature, etc.) and $\upsilon(t)$ is the random rate fluctuation (frequency modulation noise).[16] Linear frequency aging D is a simplification, but higher order terms are hardly ever used.

The most commonly used model to represent clock noise in the frequency domain is the *power-law noise* model. The power spectral density (PSD) of $\upsilon(t)$ is modeled as a linear combination [Bre97]

$$S_y = \sum_{\alpha=-2}^{2} h_\alpha f^\alpha \qquad f \le f_h \tag{2.10}$$

where f is the Fourier frequency and the five coefficients $h_{-2}, \ldots, h_2$ are device dependent parameters. The upper cutoff frequency f_h depends on the low-pass filtering of the oscillator, its output buffer amplifier and the bandwidth of the measurement system.[17] The relation between the PSDs of frequency deviation (2.4) and time error (2.6) is [SAHW90, p. TN-34]

$$S_x = \frac{1}{(2\pi f)^2} S_y = \frac{1}{4\pi^2} \sum_{\alpha=-2}^{2} h_\alpha f^{\alpha-2} \qquad f \le f_h \tag{2.11}$$

Most oscillator data sheets however show a plot of the SSB phase noise to carrier power ratio[18] in dBc/Hz [Ril03]

$$\mathscr{L}(f) = 10\log\left[\frac{1}{2}S_\phi(f)\right] = 10\log\left[\frac{1}{2}(2\pi\nu_0)^2 S_x(f)\right] \tag{2.12}$$

Future clock values can be predicted as

$$C(t_1) = C(t_0) + \int_{t_0}^{t_1} R(t')\,\mathrm{d}t' = T_0 + (1+S)(t_1 - t_0) + \frac{1}{2}D(t_1 - t_0)^2 + \int_{t_0}^{t_1} [E_i(t') + \upsilon(t')]\,\mathrm{d}t' \tag{2.13a}$$

Practical application of equation 2.13a requires a lot of knowledge about the clock in question, its reactions to environmental conditions, and needs careful control and/or monitoring of those conditions. Over short to medium intervals $\Delta t = (t_1 - t_0)$ and for most clocks, the contributions of clock aging D and noise $\upsilon(t)$ to the time error $(T_1 - T_0 - \Delta t)$ are typically minuscule compared to skew S and temperature sensitivity. A much simpler model of short-time clock prediction is

$$C(t_1) \approx C(t_0) + [1 + S(t_0)]\,(t_1 - t_0) \tag{2.13b}$$

In contrast to equation 2.13a it is assumed that the rate $R(t)$ has only minor variation over interval $[t_0, t_1]$. This is reasonable, provided that $(t_1 - t_0)$ is small enough, so that environment conditions are stable and aging and noise are insignificant.

[16] Unfortunately, the literature uses much inconsistent terminology. Especially the terms *skew* and *drift* denote many different concepts.

[17] In practice there is also a lower cutoff frequency f_l due to the finite duration of the measurement.

[18] Although $\mathscr{L}(f)$ is not recommended by the literature.

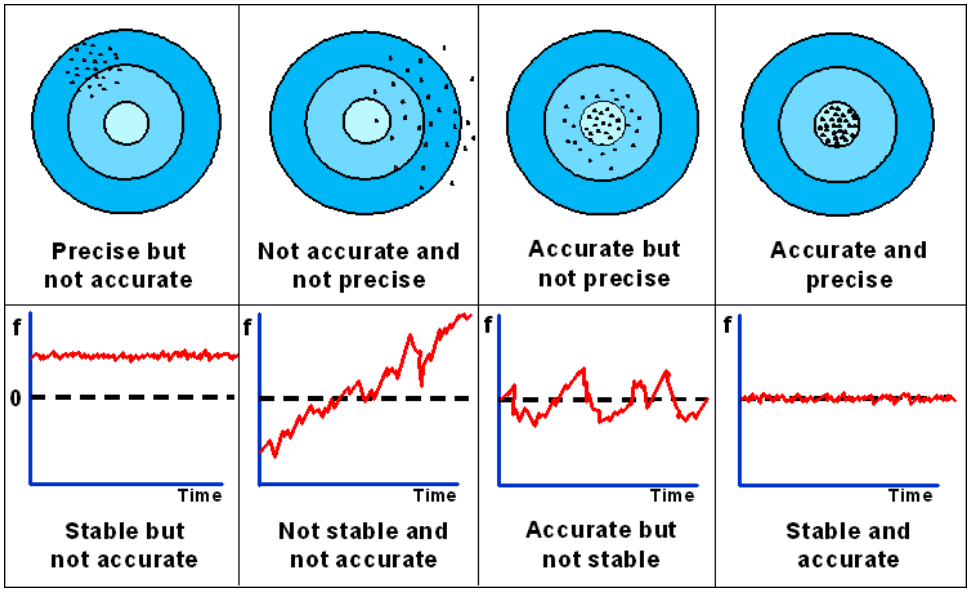


Figure 2.2.: Accuracy, Precision, and Stability [Vig07, p. 4-2]

2.2.2. Accuracy, precision, resolution, and stability

A graphical explanation of the terms *accuracy*, *precision*, and *stability* is depicted in fig. 2.2.

Accuracy A measurement is *accurate* if the result is close to the true value of the measurand.

Precision Measurements are *precise*, if repeated measurements produce small variation in results. Precision is the degree of specified detail which can be observed. It corresponds to the number of *significant* digits in measurement results, which can be obtained repeatably and reliably.[19]

Resolution is the granularity of a measurement result, i. e., the minimum (digital) non-zero difference between readings.

Stability is the quality of being free from change or variation. Stability is a property of an observed quantity; it is not a property of its measurement.

Calibration can compensate a lack of frequency accuracy (i. e., a frequency offset). Good resolution is accomplished with high counter frequency. Synchronization—setting the clock to the same time as a reference clock—establishes time accuracy. Frequency stability determines, how long a clock can keep the time error within specified bounds. Unstable clocks need much shorter resynchronization/recalibration intervals than more stable clocks (cf. table 2.4).

Various variances and deviations (square roots of variances) are used to characterize the fluctuations of a frequency source in the time domain. Riley lists 13 types of variances in [Ril08, p. 11].

[19] NTP uses the term precision for *the smallest possible increase of time that can be experienced by a program*, i. e., the elapsed time to read the system clock from userland.

The classic N-sample or standard variance

$$s^2 = \frac{1}{N-1}\sum_{i=1}^{N}(\bar{y}_i - \bar{y})^2 \qquad \text{where } \bar{y} = \frac{1}{N}\sum_{i=1}^{N}\bar{y}_i \tag{2.14}$$

should *not* be used, because it is non-convergent for some common noise types (cf. eq. 2.10).[20]

Five quantities are used by standardization bodies for characterization of time stability [IT96, p. 13] [IEE99]. These are

1. Allan Deviation (ADEV) σ_y is the most common time domain measure of frequency stability. It can be computed from the first differences of M (averaged) frequency samples $\bar{y}_i$ or, equivalently, from $N = M+1$ second differences of phase samples x_i.

$$\sigma_y^2(\tau) = \frac{1}{2}\left\langle \left(\Delta y\right)^2 \right\rangle \cong \frac{1}{2(M-1)}\sum_{i=1}^{M-1}(\bar{y}_{i+1} - \bar{y}_i)^2 \tag{2.15a}$$

$$\sigma_y^2(\tau) = \frac{1}{2\tau^2}\left\langle \left(\Delta^2 x\right)^2 \right\rangle \cong \frac{1}{2(N-2)\tau^2}\sum_{i=1}^{N-2}(x_{i+2} - 2x_{i+1} + x_i)^2 \tag{2.15b}$$

2. Modified Allan Deviation (MDEV) Mod. σ_y involves an additional phase averaging step

$$\begin{aligned}\text{Mod.}\,\sigma_y^2(m\tau_0) &= \frac{1}{2(m\tau_0)^2}\left\langle \left(\Delta^2 \bar{x}\right)^2 \right\rangle \\ &\cong \frac{1}{2m^4\tau_0^2(N-3m+1)}\sum_{j=1}^{N-3m+1}\left[\sum_{i=j}^{j+m-1}(x_{i+2m} - 2x_{i+m} + x_i)\right]^2\end{aligned} \tag{2.16}$$

   where $m = 1, 2, \ldots, \lfloor N/3 \rfloor$.

3. Time Deviation (TDEV) σ_x

$$\sigma_x^2(\tau) = \frac{\tau^2}{3}\text{Mod.}\,\sigma_y^2(\tau) \tag{2.17}$$

4. Root mean square of Time Interval Error

$$TIE_{rms}(m\tau_0) = \sqrt{\left\langle \left[x(t+m\tau_0) - x(t)\right]^2 \right\rangle} \tag{2.18}$$

5. Maximum Time Interval Error (MTIE) The maximum time interval error MTIE(τ) is defined as a specified percentile, β, of the random variable X.

$$X = \max_{0 \le t_0 \le T-\tau}\left(\max_{t_0 \le t \le t_0+\tau}[x(t)] - \min_{t_0 \le t \le t_0+\tau}[x(t)]\right) \tag{2.19}$$

   Fig. 2.3 explains the MTIE definition graphically.

[20] The problem with (2.14) is that the average $\bar{y}$ is not stationary for $\alpha < 0$ in (2.10) or $D \neq 0$ in (2.9) [Ril08, p. 14].

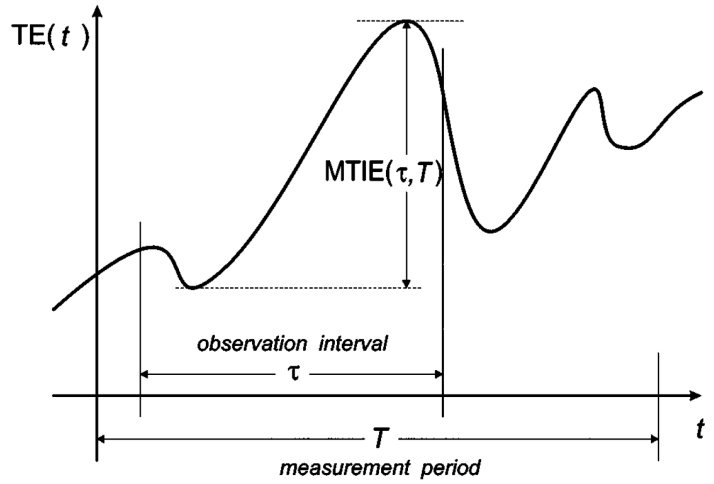


Figure 2.3.: Graphical explanation of MTIE(τ, T) [BM00]

Double logarithmic plots of the (Modified) Allan Deviation over τ are used to identify power law noise processes in clocks, i. e., the exponents α which dominate S_y in (2.10) can be determined by observing the slope of the plot [IEE99, p. 12]. Allan Deviation σ_y does not distinguish between white ($\alpha = 2$) and flicker phase noise ($\alpha = 1$) types. The Modified Allan Deviation and the Time Deviation do not have this ambiguity [AAH97].

The Allan deviations are sensitive to systematic effects like diurnal variations, which might mask noise effects [IT96]. Cyclic disturbance causes a distinctive pattern of maxima and minima at the half period and period of the stimulus [Ril03, p. 47]. Unavoidable systematic effects must be adequately filtered before calculating σ_y.

TIE_{rms} and MTIE are mainly used by the telecommunication industry [Bre97]. MTIE measures peak time deviation and is therefore very sensitive to transients and outliers [Ril08, p. 33]. Straightforward computation of MTIE scales as $O(n^2)$, but an optimized algorithm achieves $O(n \log n)$ [BM00].

2.3. Hardware Clocks

2.3.1. Crystal oscillators

The crystal oscillator is by far the most important frequency control device (cf. table 2.3). Even inexpensive quartz crystals for wrist watches can have a frequency accuracy of 1 ppm and even better stability σ_y [Lev99]. Low cost, small size, low energy consumption, robust design, and long life are important considerations too.

Since the 1960-ies man-grown single crystals with relatively high purity are used as raw material for resonators. Quartz is a highly anisotropic material. The electromechanical properties of a resonator depend not only on the exact geometry of the resonator, but also strongly on the angles of cut relative to the crystal lattice. An angular difference of one arc-minute makes a significant difference.

Technology	Units per year	Avg. unit price $	Worldwide market $ / year
Quartz crystal resonators & oscillators	3×10^9	1	4×10^9
Rubidium cell frequency standard	50000	2000	100×10^6
Caesium frequency standard	500	50000	25×10^6
Hydrogen maser	20	100000	2×10^6

Table 2.3.: Frequency control device market (Estimates for 2006) [Vig07, p. 1-2]

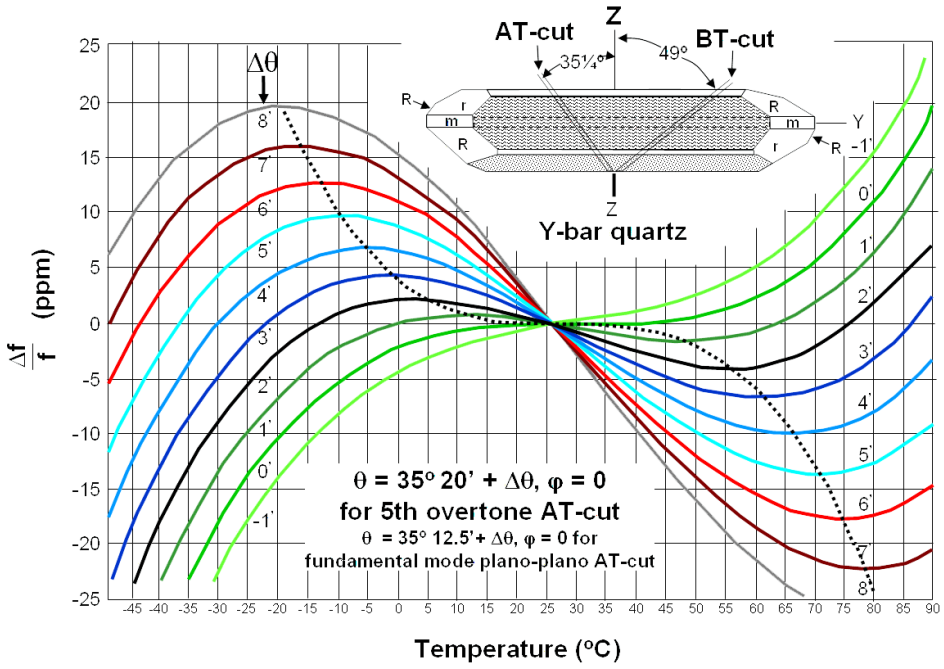


Figure 2.4.: Frequency-Temperature vs. Angle-of-Cut for AT-cut crystal [Vig07, p. 4-44]

Figure 2.4 displays temperature dependent frequency error curves of AT-cut crystals with angle of cut as parameter.

Environmental effects (temperature, humidity, pressure, acceleration, vibration, electromagnetic fields, ionizing radiation...) on frequency have been studied thoroughly [ABC+92, WG92]. Because temperature is usually the dominating factor, only temperature compensated cuts are used for frequency control devices. The AT-cut is most popular.[21] Its frequency dependency on temperature is a cubic parabola

$$\frac{\Delta f}{f} = a(T - T_0) + b(T - T_0)^2 + c(T - T_0)^3 \tag{2.20}$$

where T_0 is 25 °C and coefficients a, b, c depend on angle of cut. The inflection point is conveniently near room temperature (25 °C... 35 °C).

[21] The letter 'T' in the AT- and BT-cuts (and others) stands for "temperature compensated".

The acronyms XO and SPXO denote simple packaged crystal oscillators without any temperature compensation or control. A number of schemes have been developed to mitigate their inherent frequency instability caused by temperature changes.

Commonly used are the following:

TCXO use temperature sensitive reactances (thermistor/resistor networks and varactor diodes) to compensate the frequency vs. temperature variations of the crystal. Peak to peak frequency deviations are reduced by a factor of about 100, yielding about ± 0.5 ppm over a temperature range of -55 °C... 85 °C.[22]

MCXO utilize self-temperature sensing (dual mode) resonators to virtually eliminate thermometry related errors.[23] A microcomputer and digitally stored calibration coefficients are used to control output frequency. About ± 0.03 ppm ($= 30$ ppb) over a temperature range of -55 °C... 85 °C are achieved.

OCXO For best frequency stability the crystal temperature must be stabilized. In an oven controlled XO the crystal and other temperature sensitive components are enclosed in a thermally insulated container along with a heating element and a temperature sensor.[24] The oven is adjusted to a temperature where the f vs. T graph of the crystal has zero slope.[25] OCXOs reduce frequency variations by a factor > 1000, but at the cost of much higher power consumption. About ± 10 ppb temperature instability are common. High-end SC-cut units stay within ± 0.1 ppb over a wide temperature range, have short time stability $\sigma_y(1\,\mathrm{s}) = 10^{-12}$ and aging of 10^{-11} / day.

Several mechanisms (mass transfer due to contamination, stress relief in the mounting and bonding structure, quartz out-gassing, diffusion, etc.) cause frequency aging [Vig07, p. 4-6]. High quality OCXO and MCXO have considerable less aging than cheaper designs. Table 2.4 shows some typical values and required resynchronization/recalibration intervals for a guaranteed maximum clock error of 25 ms.

2.3.2. Other frequency sources

Crystal oscillators are not well suited for applications where high frequency accuracy or long-term frequency stability are important [Lev99]. The mechanical resonance frequency of a crystal depends on the exact geometry of the artifact and is therefore hard to replicate.

Atomic frequency standards use atomic or molecular resonances. Their stability performance is compared in fig. 2.5. Hydrogen masers provide best medium-time stability while caesium clocks offer best long time stability. There is some overlap in stability and unit price between high end quartz and low end rubidium devices. Rubidium frequency aging is small in comparison with quartz. Caesium devices do not suffer from frequency aging at all.

[22] In practice frequency calibration (to compensate aging) degrade f vs. T performance significantly [Vig07, p. 4-52].

[23] The fundamental mode (f_1) and third overtone are excited simultaneously. The beat frequency $f_\beta = 3f_1 - f_3$ depends nearly linearly on crystal temperature. In principle two separate resonators in close thermal contact could be used too [SCF+08].

[24] High performance units use a *double oven* design, where the outer oven stabilizes the ambient temperature of the inner oven.

[25] Usually SC-cut (stress compensated) crystals are used. The inflection point of their f vs. T graph is about 95 °C. SC-cut crystals have several advantages over AT-cut crystals. They are more expensive to produce because of their double-rotated cut.

Osc. type	Temp. Stability	Aging / Day	Resynchr.	Recalibr.
SPXO	50×10^{-6}	1×10^{-8}	5 min	9 years
			8 min	200 days
TCXO	1×10^{-6}	1×10^{-8}	10 min	10 years
			4 h	80 days
OCXO	2×10^{-8}	1×10^{-10}	6 h	50 years
			4 d	1.5 years
MCXO	2×10^{-8}	5×10^{-11}	6 h	94 years
			4 d	3 years
RbXO	2×10^{-8}	5×10^{-13}	6 h	not needed
			4 d	300 years

Table 2.4.: Resynchronization and recalibration intervals (based on [Vig07, p. 8-10])

Research in optical frequency standards suggests, that reproducible relative frequency accuracies at the $10^{-17}\ldots10^{-18}$ level should be achievable within a few years [Gil05]. Another current research area is the chip-scale atomic clock for applications requiring atomic timing in portable battery-powered devices [LRV+07].

Ceramic resonators perform worse than quartz crystals. Frequency tolerances at 25 °C and frequency variation over the operating temperature range are both typically a few thousand ppm. They should only be used for timing applications where very small savings in unit cost matter more than accuracy.

Several microcontrollers can optionally generate their clock signal from on-chip integrated RC-oscillators.[26] Frequency accuracy is limited to about 2 % (20000 ppm).

2.3.3. Computer clocks

Clocks built into computer systems are almost always of the crystal oscillator & counter type depicted in fig. 2.1. Frequency accuracy and stability requirements for computers are typically low. Low price bulk AT-cut crystals with large frequency tolerances are common.

The Intel x86 PC platform (and hence the general purpose processor of I-SENSE nodes) has several timing sources:

RTC The real time clock is battery backed and keeps time when the PC is powered off. Since the internal counter is not software accessible, the resolution is only one second. The RTC can periodically interrupt the CPU. The interrupt frequency is programmable from 2 Hz to 8192 Hz in powers of two.

PIT The Intel 8254 Programmable Interval Timer has three independent 16 bit counters. It has a nominal frequency of 1193181.81 Hz.[27] The nominal frequency can be divided by a programmed 16 bit value to generate periodic interrupts with frequencies down to 18.2 Hz.

[26] Among them the popular Atmel AVR and Microchip Technology PIC families of microcontrollers.

[27] This is one third of the NTSC color subcarrier frequency—reminiscent of the 1981 color graphics adapter (CGA).

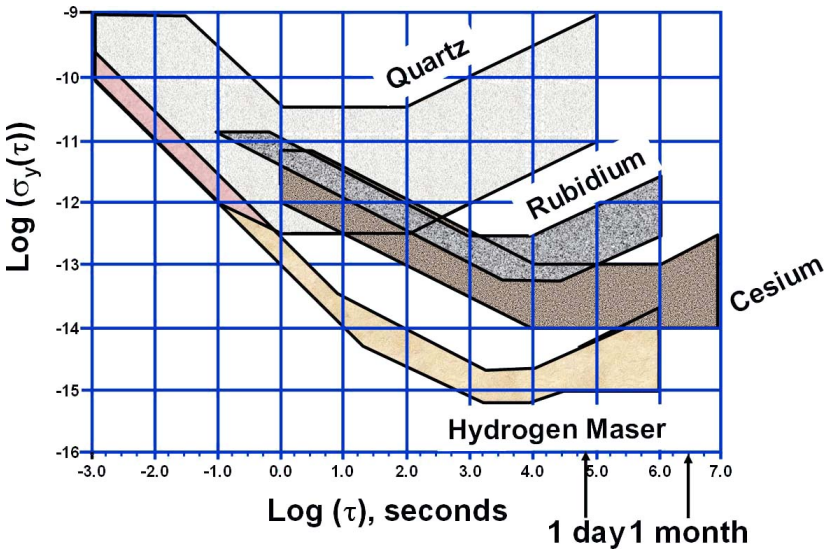


Figure 2.5.: Short term stability ranges of various frequency standards [Vig07, p. 7-5]

ACPI The Advanced Configuration and Power Interface Specification requires a 24 bit or 32 bit power management timer running with a fixed frequency of 3.579545 MHz (NTSC-M color subcarrier frequency).

TSC The Time Stamp Counter is 64 bit wide and available on all x86 processors since the Pentium. It is a processor cycle counter (PCC). Its high resolution and the ability to read it in a single machine instruction[28] seem to make it the ideal timer on the PC platform. Unfortunately, power saving measures like idle states and CPU throttling and can make the TSC frequency highly variable.[29] Moreover on multiprocessor systems the per-processor TSCs can proceed at different rates. The TSC cannot be programmed to cause interrupts, which makes it unsuitable for a scheduler clock.

APIC The local APIC (Advanced Programmable Interrupt Controller) timer can interrupt its associated processor when a programmed count is reached. The timer is 32 bit wide. Its frequency is derived from the processor bus clock, divided by a programmable value. It can be configured for one-shot or periodic operation. Dependent on the actual CPU model, the timer may or may not run at a constant rate in different power states and during power state transitions.

HPET The High Precision Event Timer (a. k. a. Multimedia Timer before 2002) is a monotonic 64 bit counter running with at least 10 MHz. At least three comparators and match registers and one periodic capable timer are provided. The specification permits large frequency instabilities of ± 500 ppm over intervals ≥ 1 ms and ± 2000 ppm over intervals ≤ 100 µs. HPET is not available on older hardware and unsupported by older operating system releases.

Availability, resolution, width, interrupt features, and access/reprogramming speed vary a lot between the above hardware timers. Besides, several of the timers are BIOS managed and quite a lot of broken

[28] RDTSC or RDTSCP. The latter instruction prevents out of order execution.

[29] Spread-spectrum clocking—a technique for electromagnetic interference reduction—introduces clock frequency modulation too. [HFB94]

implementations have been reported. It is therefore impossible to choose an optimal timer without detailed knowledge of the actual hardware/BIOS combination and/or testing. Some operating systems consequently choose the best system timer during the start-up process after probing the hardware [Kam02].

Built-in clock hardware on other platforms varies, but the combination of crystal oscillator and counter is always present. There is great variation in other hardware details like programmable prescalers or PLLs, availability and number of comparators, one shot and periodic interrupt facilities, and speed and level of software access to timer hardware registers. The timer hardware details of I-SENSE signal processor boards are described in section 4.1.1 on page 47.

Many microcontrollers have even more elaborate timer hardware, like up/down counters, external event counters, and PWM output modes. A highly useful feature for timekeeping purposes, which is present in many microcontrollers,[30] is the capture register. By capturing timestamps of (external) events in hardware, the timing uncertainty associated with interrupt latency is avoided. Unfortunately, neither commercial off the shelf (COTS) PC hardware nor the I-SENSE DSP boards are equipped with capture registers.[31]

There is little recent literature about the quality of COTS computer clocks. Most of it does neither include temperature data nor specify whether the machines have been operated in air conditioned rooms.

Marouani and Dagenais report accuracy and stability data on the CPU clocks of some 30 Intel and AMD systems with nominal CPU frequencies between 266 MHz and 2.4 GHz [MD08]. CPU frequency offset from nominal frequency was a few thousand ppm. Frequency variation between eight equal Pentium IV models spans 29 ppm. A temperature change from 28 °C to 47.25 °C caused a -8.3 ppm change of CPU frequency, diurnal variation was 0.74 ppm.[32]

Kohno et al. used TCP timestamps to *remotely* measure clock frequency [KBC05]. They found that individual machines show only 1–2 ppm clock skew over time, but found some 50 ppm variation between individual machines, even identical models.

The static rate difference calibration mentioned in section 1.1 resulted in a maximum rate difference of 142 ppm between a management PC and four I-SENSE DSP cards.

The author's measurement of an I-SENSE node in a laboratory *without* air conditioning is depicted in fig. 2.6. Frequency variation over 24 hours stayed below 2 ppm but was larger over several days. The lower (blue) curve and the scale on the right y-axis show the normalized rate difference between the system clock of the general purpose processor and a DSP node. Both processors/oscillators were located within the same case and thus thermally coupled. This caused an effect similar to common mode rejection—the change in skew *between* both clocks was only about 0.1 ppm.[33]

[30] E. g., the Atmel AVR or TI MSP430 controllers.

[31] A main reason why PC based precision timing applications always need more additional hardware (timing cards) than only an external reference clock.

[32] Only one system has been measured.

[33] A constant bias of 22 ppm has been removed. The observed effect is helpful for inter-node synchronization, but it depends entirely on the *coincidental* matching of the f-versus-T characteristics of both crystals. (cf. fig. 2.4 on page 15 and [SCF+08].)

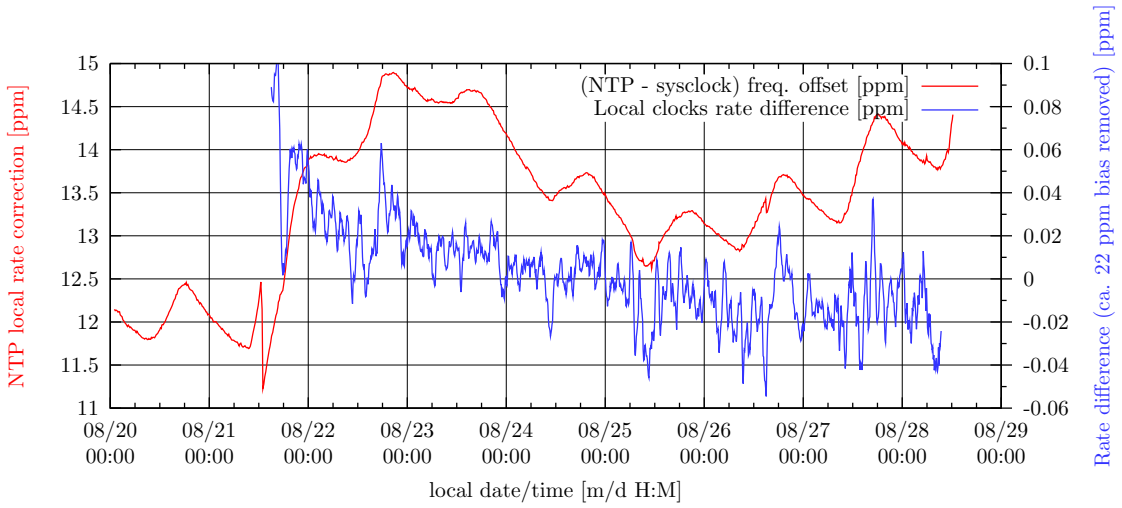


Figure 2.6.: Frequency variation of a computer clock over a week.

2.3.4. External reference clocks

Synchronization of computer networks to external timescales like UTC requires injection of reference time into the network. The points of injection are external reference clocks that are interfaced to network nodes. Many national time standard laboratories attach atomic clocks to public NTP servers in order to inject their realizations of UTC into the Internet. Most external clocks are either radio controlled clocks or GPS time receivers.

Radio clocks are synchronized to terrestrial time signals. Many countries operate longwave time signal transmitters like WWVB at 60 kHz in the U. S., DCF77 at 77.5 kHz in Germany, and TDF at 162 kHz in France. The signal range is between 2000 km (DCF77 50 kW) and 3500 km (TDF 2 MW). Small indoor antennas are usually adequate. Cheap, narrow bandwidth receivers have only accuracies in the 5–25 ms range. Commercial correlation receivers achieve about 50 µs when they are calibrated for the transmitter to receiver distance.

GPS receivers can operate worldwide but antennas need good sky view (and therefore rooftop access is frequently required). Even cheap units have 1 µs accuracy. Commercial quality units achieve about 50 ns accuracy and are equipped with Time Receiver Autonomous Integrity Monitoring (TRAIM) to protect against faulty satellite signals.

LORAN (LOng RAnge Navigation) is a terrestrial longwave navigation system that uses high peak power pulses at 100 kHz [RSJ+05]. It was introduced in 1957 and is used mainly by the U. S., Europe, and Japan. GPS performs much better than the present LORAN-C system, however the U. S. have modernized the system to achieve better navigation and timing performance. The enhanced LORAN (eLORAN) system can be used as a timing source with about 100 ns accuracy.[34] The eLORAN time and frequency accuracy can support almost all civilian applications. E-Loran can be used as a local backup/complement to GPS.

[34] This requires receiver and antenna calibration and LORAN data channel (LDC) corrections.

2.3.5. Interfacing external clocks

At the physical layer the clock-computer interface is frequently realized as pulse per second (PPS) signals. Only minimal add-on hardware (a cable and sometimes a level converter) is needed, because most computers can be programmed to trigger an interrupt when a signal flank arrives at certain parallel or serial port pins.[35] A PPS API has been defined and free implementations for some POSIX systems exist.[36][MMB+00, MK00] Since PPS signals only mark the start of a second but do not tell which second begins, an additional serial or USB connection is needed.

Cheap radio clocks and several navigation-only GNSS receiver models (through the NMEA protocol) use a serial link alone. Because of the relatively slow communication speed and (hardware and software) buffering of serial data streams a large timing uncertainty of tens of milliseconds results.

Precision timing hardware (like PCI timing cards) normally uses IRIG-B time code signals to synchronize with external reference clocks [IRI04].

2.4. Software Clocks

This section deals with clocks from the viewpoint of software. Application programs use time related APIs rather than accessing timer hardware directly. System software isolates applications from idiosyncrasies of actual timer hardware by providing clock abstractions.

2.4.1. Operating system view

Operating system provided clocks can be categorized into three classes:

Counter register read access is the most simple API. Examples are library wrappers around PCC read instructions or the `QueryPerformanceCounter` function in Microsoft Windows. The counter frequency is either available from an API[37] or must be obtained from a hardware specification. Since the epoch is the time of the last system reset, these clocks are rather used for time interval measurements than for providing UTC. Because of low clock access overheads and high frequencies, these clocks are ideal for execution time profiling and similar performance measurements.

Periodic timer interrupts are used by virtually all operating systems. Windows system time is implemented this way. The interrupt service routine increments a software counter. Both the interrupt events and the intervals between them are conventionally called *ticks*. The timer interrupt frequency limits clock precision. Most older Unix/Linux systems used 100 Hz (10 ms). As processor speeds have increased, higher frequencies up to 1000 Hz are nowadays more

[35] On commodity PCs either the DCD pin (carrier detect) of a serial port or the ACK pin of a parallel port is used.

[36] The NTP reference implementation supports the PPS API on Windows too.

[37] On many Windows systems the `QueryPerformanceFrequency` function returns 3.579545 MHz—the ACPI power management timer frequency.

common.[38] At the time of this writing the Microsoft Windows system clock still uses either 100 Hz or 64 Hz.

Interpolation schemes combine the two previously described approaches in order to improve clock resolution. A periodic interrupt is used for basic time keeping, and a high frequency counter is used to interpolate between ticks.

To achieve increased resolution the Windows port of the NTP reference implementation does system clock interpolation at the application layer (i. e., outside the kernel). The implementation uses the multimedia timer API to get 1000 callbacks per second from the kernel. The performance counter is used to interpolate between callback events. A drawback of this approach is that callback invocation may be delayed for several milliseconds during phases of heavy system load.[39]

2.4.2. Clock phase and rate adjustment

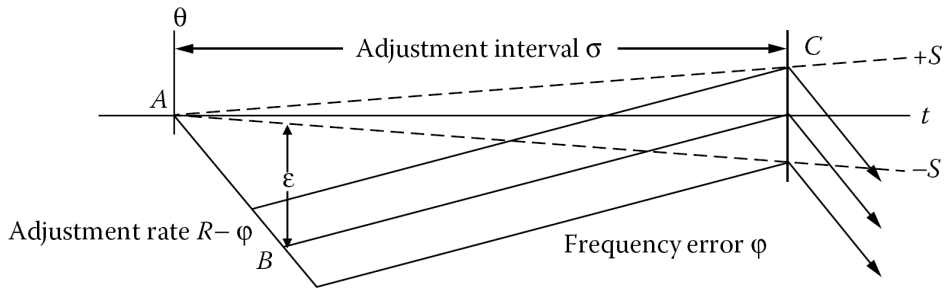


Figure 2.7.: Clock adjustment sawtooth error [Mil06b, p. 180].

The frequency offsets of computer hardware clocks are in a range from a few to a few hundreds ppm. Since even a rate offset of 1 ppm leads to a time error of 86.4 ms/day, some means of correction is needed—at least for clocks tracking wall clock time.[40]

Setting the time to a new value is problematic, because it violates assumptions about a continuous timescale. Stepping time backwards is known to confuse application programs. Clock steps are hence avoided and (small) phase corrections are made by changing the clock rate instead. The time a clock needs to run with increased or decreased rate to achieve a given phase correction is called the *amortization interval*.

Unix systems use the `adjtime` system call to make small adjustments to the system time. This amortizes the given signed time offset by using an increment that is slightly larger or smaller than normal. The kernel reverts automatically to the standard increment as soon as the adjustment is complete. Unix typically increases or decreases the rate during adjustments by 500 ppm, i. e., it

[38] Periodic interrupts on an otherwise idle machine cause wakeups from power-saving states, which is detrimental to power efficiency. Clock ticks also add to "system noise". The Linux kernel is therefore moving away from this traditional design. [TEFK05, SPV07]

[39] A slightly modified version of this interpolation clock is used for the I-SENSE synchronization process on general purpose processor nodes.

[40] Clocks that measure the system uptime(e. g., `GetTickCount` on Windows) usually cannot be phase or rate adjusted.

takes 2000 s (more than 33 minutes) to amortize one second. Only three clock rates are available, the unadjusted rate with a frequency error (line $\overline{BC}$ in fig. 2.7), the same rate reduced by -500 ppm (line $\overline{AB}$), and increased by -500 ppm. The $\overline{ABC}$ sawtooth of fig. 2.7 has to be repeated permanently (and `adjtime` calls made at A) in order to keep the offset error low.

The Windows scheme for clock adjustment offers more flexibility for rate control, but there is no API for amortized offset corrections. The `SetSystemTimeAdjustment` function sets the increment added to the time-of-day clock at each clock interrupt. This allows arbitrary system clock rates, but since Windows system time has 100 ns resolution, the granularity of possible rates is only 10 ppm/6.4 ppm at the common 100 Hz/64 Hz tick rates.

POSIX systems with kernel support for NTP[41] allow very fine grained rate settings and offset corrections with the `ntp_adjtime` system call.

2.4.3. Application view

On general purpose operating systems direct access to the timer hardware is not possible.[42] Clocks that are provided by the OS are used.

Table 2.5.: Time representations in various APIs

OS	Time format	Size	Resolution	Precision	Wrap around	
Windows	System Time	8×16 bit	1 ms	ca. 10 ms	65536 years	
	File Time	2×32 bit	100 ns	ca. 10 ms	58454 years	
	MS-DOS Time	2×16 bit	2 s	2 s	128 years	
	Windows Time	32 bit	1 ms	ca. 10 ms	49.7 days	[a]
	MM Timers	32 bit	1 ms	$\geq$1 ms	49.7 days	
	HR Timer	64 bit	279.4 ns	279.4 ns	163300 years	[b]
DSP/BIOS	Highres Timer	32 bit	13.33 ns	13.33 ns	57.27 sec	[c]
	Lores Timer	32 bit	1 ms	1 ms	49.7 days	[d]
NTP	internal formats	64...128 bit	1 µs *or* 1 ns	$\geq$1 ns		[e]
	wire format	64 bit	232.8 ps		136 years	
POSIX	struct timeval	2×32 bit	1 µs	$\geq$1 µs	136 years	[f]
	struct timespec	64 bit+32 bit	1 ns	$\geq$1 ns	$>10^{11}$ years	[f]

[a] Epoch is system boot, i. e., the system uptime is returned.
[b] Assuming `QueryPerformanceFrequency()` = 3.579545 MHz
[c] The timer frequency is $f_{CPU}/4$ or $f_{CPU}/8$ depending on CPU model. The values shown are for an I-SENSE DSP node with a 600 MHz TMS320C64X CPU.
[d] The low-resolution timer period is configurable with a 1 ms default value.
[e] Either micro- or nanosecond resolution is used dependent on a status flag.
[f] The standard does not specify the size of the integer seconds part (`time_t`).

[41] Solaris, *BSD, Linux, and Darwin.
[42] Unless one goes through the hassles of developing, maintaining, and deploying a device driver.

From the application programmer's view, it is irritating that so many different date-time and time interval representations with varying resolution and precision are provided. Table 2.5 shows some examples. The I-SENSE time synchronization software has to deal with several of the tabled time formats. Conversion between formats must scale resolutions and consider different epochs (cf. table 2.6).[43]

Timescale	Epoch
Windows	1601-01-01
MS-DOS	1980-01-01
NTP	1900-01-01
POSIX	1970-01-01
.NET	0001-01-01
MacOS	1904-01-01

Table 2.6.: Epochs of some computer timescales

API documentation is frequently lacking important specifications. Answers to the following questions often require additional research or testing:

Precision What is the minimum time interval between clock increments?

Monotonicity Is the clock (strictly) monotonic?

Accuracy Is the clock disciplined, i. e., rate and offset controlled? If yes: How does this affect monotonicity and rate accuracy? Can the synchronization state be obtained?

Obtaining answers to questions about clock *reliability*[44] and clock *access overhead* most often requires running test programs on the target hardware.

2.5. General Clock Synchronization Model

Since clocks (and ensembles of clocks) are dynamical systems, methods and terminology from *control theory* can be applied to the synchronization problem. Synchronization solutions can be described as closed-loop control systems. Essential components of the feedback control loop are detailed in the following subsections.

[43] Local time zones, synthetic time scales like profiling clocks, and time related APIs mandated by programming language standards complicate the picture further. Java APIs e. g., use the POSIX timescale with 1 ms resolution but intervals can be timed with 1 ns resolution (`System.nanoTime`).

[44] There are reports on the Internet about Windows systems with broken/erratic `QueryPerformanceCounter` implementations.

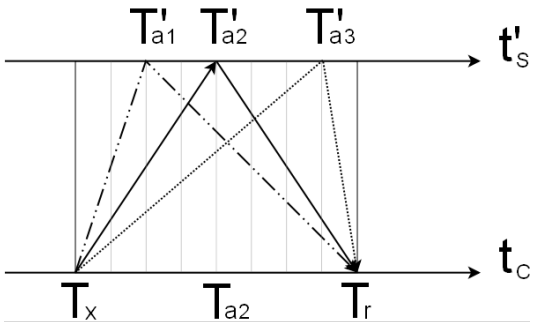


Figure 2.8.: Two-way time transfer

2.5.1. Measurement of the time difference between a local clock and a remote clock

Determining the time difference of a local clock relative to a remote clock is called *time transfer* in metrology [Lev99, Lev08]. All time distribution methods depend on the accuracy of communication channel delay. Dependent on time signal topology several time transfer techniques can be distinguished:

One-way (broadcast) The reference clock broadcasts its time. Receivers must know the channel delay. After receiving the broadcast they calculate the time of reception by adding the channel delay to the transmit time of the sender. Radio controlled clocks and single channel GPS time receivers make use of the one-way method. A real-world limitation is, that channel delay varies over time, and that this variation is not entirely predictable.

Multi-hop paths over (possibly congested) Internet paths have large unpredictable delay fluctuations. The one-way method is therefore unsuitable across the Internet. Broadcast message propagation delay on LAN or single hop WLAN links is much more predictable.

Two-way (round trip) The channel delay can be estimated using a query/response scheme. The local client sends a time request to a time server that sends its current time in reply.[45] If the channel is reciprocal, its one-way delay can be estimated as half the round trip time $RTT = T_r - T_x$. The error of the method is proportional to the path asymmetry a. If we define asymmetry with reference to fig. 2.8 as

$$a = \frac{2(T_a - T_x)}{RTT} - 1 \qquad -1 < a < 1 \tag{2.21}$$

then the timing error θ_e due to path asymmetry a will be

$$\theta_e = \frac{a}{2} RTT \qquad |\theta_e| < \frac{RTT}{2} \tag{2.22}$$

In the depicted example the case of the faster request path (T'_{a1}) would cause the client clock to be late by one quarter of the RTT, the even faster response path (T'_{a3}) would set it early by ⅜ RTT.

[45] The two-way method does not mandate a client/server architecture. The NTP symmetric mode is a counter-example. Two-way satellite methods (TWSTFT) operate even full-duplex with uncertainty below the nanosecond. [GA05]

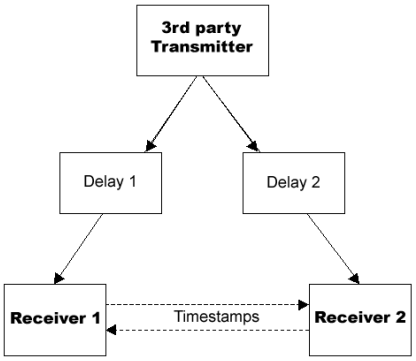


Figure 2.9.: Common-view method

Common-view (third party broadcast signal) The method is depicted in fig. 2.9. Two receivers use their local clocks to measure the arrival of the same third party broadcast signal. The receivers then exchange their measurement results to obtain the mutual clock offset. The method requires the receivers to be approximately equidistant from the third party, to make the channel delays experienced by the receivers nearly equal.[46] One advantage over the one-way method is, that knowledge of absolute path delay is not required. Another advantage is, that fluctuations in the channel delay cancel out if the paths are similar.[47] The sender needs no precise clock; it does not even need to "know" that it is being used for time transfer.

The common view method is only applicable to computer networks with a physical broadcast channel. Although the technique can be used on a LAN,[48] it is especially attractive on WLAN, because the medium access delay of the sender does not matter [EGE02].

Two other time transfer techniques known in the metrology literature are the *All-in-View* and the *portable clock* method.[49] They are not relevant for computer network synchronization.

Decomposition of path delay

Message passing in distributed systems involves many more delays than only signal propagation across a medium between two nodes. The speed of electromagnetic waves is about 300 m/µs in the atmosphere and roughly 200 m/µs along cables. End-to-end message delivery delays of hundreds of microseconds between (W)LAN nodes are dominated by processing rather than physical signal propagation.

Processing of a one-way message from the sender application layer down the network protocol stack and vice versa in the receiver protocol stack is sketched in table 2.7. Additional software layers like virtual machines or middleware transport services would add even more unpredictable delays.

[46] In high precision applications some correction for the residual path difference is usually required.
[47] Typically because the baseline between the receivers is small compared to the sender distance.
[48] For example by utilizing already available broadcast packets like ARP requests as the third party signal.
[49] The *All-in-View* or *Melting-Pot* technique uses synchronous one-way measurements of a clock ensemble. An example is a multi-channel GPS time receiver. [Lev08]

Software layer	Processing step
Sender application	call time-of-day function, add timestamp to message
Sender application	call send-message function
Sender OS	copy message to kernel memory, invoke network stack
Sender stack	assemble packet by several layers of network stack
Sender NIC driver	wait for medium access
Sender NIC driver	transmit packet, probably re-transmit in case of collision (CSMA/CD)
(Medium)	Packet propagation
Intermediate router(s)	Enqueue, process, and forward packet
Receiver NIC	Store packet in NIC buffer, interrupt CPU
Receiver kernel	wait until higher priority interrupts processed
Receiver NIC driver	copy packet to kernel, invoke network stack
Receiver stack	process packet by several network stack layers
Receiver OS	wake up/notify receiving process
Receiver OS	scheduling delay until receiving process is assigned CPU
Receiver OS	context switch to receiving process
Receiver OS	probably extra delay due to virtual memory paging
Receiver application	read message (receive-message function returns)
Receiver application	call time-of-day function to get receive timestamp

Table 2.7.: Message delivery processing steps

Additional processing delays are not, per se, a problem for synchronization, since they can be abstracted into the end-to-end communication channel. The drawback of application-to-application delay measurements stems from the unpredictable delays due to resource contention that may occur at most of the steps in table 2.7. The combined delay uncertainties create a large overall uncertainty when delay is measured at application level. The importance of taking timestamps at the lowest possible layer is further substantiated in section 3.1.8.

2.5.2. Estimation of the time and frequency error of the local clock

Isolated offset measurements are only a rough estimate, because of the uncertainty associated with remote clock reading. Only at system startup when the clock has to be set quickly, a single remote clock reading (or the average of a few ones) is used as-is.[50] In general, protocols try to reduce the uncertainty of single measurements by applying mathematical methods to sets of measurements [Joh04].

One effective method to remove outliers is based on eq. 2.22. In many cases, the samples with minimal RTT have suffered from the smallest random delays and have little path delay asymmetry. Selecting the low RTT samples therefore tends to remove outliers and decrease overall jitter [Mil06b, p. 43f].

[50] The `ntpdate` program coming with the NTP reference implementation does this.

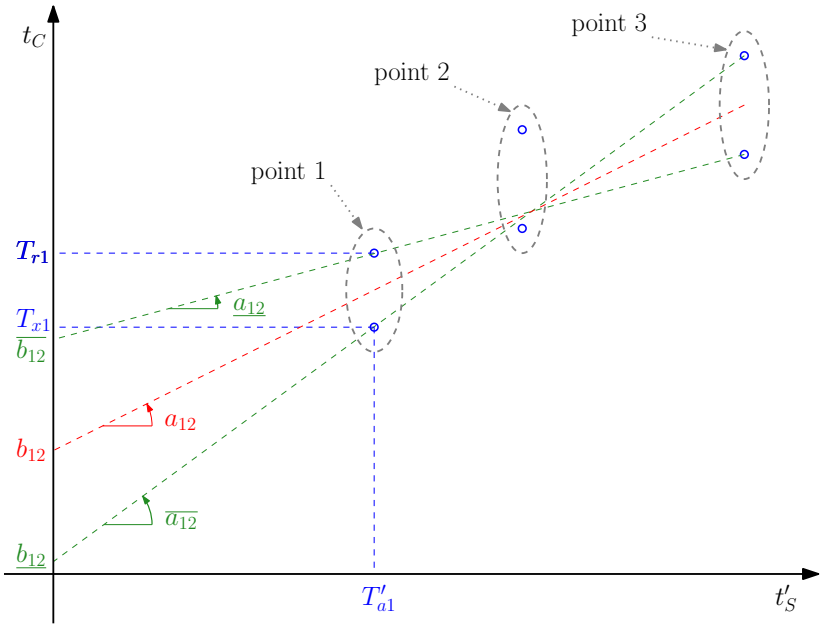


Figure 2.10.: Offset and skew estimation in the Tiny-Sync protocol [YVS07]

This characteristic of Internet paths is also present on I-SENSE intra-node links (between general purpose processor and DSPs) and is depicted in the scatter diagram of fig. 4.6 on page 58.[51]

Other proposed methods are simple first order low-pass filtering, sliding averages, linear least squares regression (LLR) techniques, statistical models in combination with parameter fitting, Kalman filters, convex-hull, and linear programming approaches [ZLX02, Joh04, SM08].

As an example of a linear programming approach the algorithm used by the Tiny-Sync protocol [YVS07] is graphically explained in fig. 2.10. Based on the assumption of linear clock functions (cf. eq. 2.13b on page 11) the client clock t_C is a linear function of the server clock t'_S. Transformation of a server timestamp T'_i to a client timestamp T_i requires two parameters: the client time b_{12} at $t'_S = 0$ and the relative rate $a_{12} = \frac{1+S_c}{1+S_S}$. For each two-way measurement of fig. 2.8 the inequalities

$$T_x < a_{12} T'_a + b_{12} < T_r \tag{2.23}$$

must hold. New measurements may or may not reduce the possible parameter intervals. In fig. 2.10 data point 3 clearly raises the lower bound $\underline{b_{12}}$ and lowers the upper bound $\overline{a_{12}}$ but it changes $\overline{b_{12}}$ and $\underline{a_{12}}$ only slightly. To avoid unbounded computational and storage requirements, Tiny-Sync stores only four selected constraints at any time.

All estimation methods are based on assumptions about the properties of communication channels and clocks. Making good assumptions about real-world network paths is hard, since they often show complex non-Gaussian delay distributions. Varying network conditions are another complication, as they may cause substantial change of delay distribution.

[51] Minimum RTT filtering was hence effective in this case too.

Multiple clocks and fault tolerance

Multiple time sources add another layer of complexity. Clocks with different properties, some of them possibly failing in Byzantine ways, create new challenges for protocols. Interval-comparison based selection algorithms (cf. Marzullo's algorithm in section 3.2.4), clustering algorithms [Mil06b, p. 50ff], and weighted combinations are common building blocks for fault-tolerant averaging or convergence functions.

The *fault-tolerant midpoint algorithm* [LL84] is a convergence function, which is e. g., used for time synchronization in the automotive network communications protocol *FlexRay* [Fle05, pp. 169–193]. The algorithm works in rounds. In each round the list of n remote clock readings is first sorted, then the f highest and the f lowest values are thrown away, and finally the time is estimated as the arithmetic average of the highest and the lowest remaining value.[52]

Schneider [Sch87] compares eight fault-tolerant convergence functions. He points out, that all fault-tolerant clock synchronization protocols can be viewed as refinements of a single paradigm. He also formalizes four required properties of convergence functions: monotonicity, translation invariance, precision enhancement, and accuracy preservation.

2.5.3. Adjusting the local clock to reduce future time differences

Not all protocols adjust the local clock, sometimes timescale transformations (cf. section 3.1.3) are used instead.

In the common case of real-time synchronization, new time and frequency error estimates can be computed when a new offset sample arrives. An overly simplistic clock adjustment algorithm would correct the rate and phase of the clock immediately using APIs from section 2.4.2 above. Frequent clock steps in both directions would result. In practice, however, clock steps are avoided and a smooth and stable rate is desired to keep short interval measurement errors low.

The time constant used for rate smoothing is always a compromise. Optimizing the clock for good interval measurement performance requires a rather large time constant to keep rate changes low. Optimizing for phase accuracy needs a shorter time constant to reduce time offset errors more quickly. The requirements for rate and phase performance—i. e., clock stability and accuracy—are conflicting.

NTP alleviates the conflict with a dynamic time constant. The time constant is enlarged when the estimated network jitter is below a threshold and shrinked otherwise [Mil06b, p. 71f]. This scheme adapts to varying network conditions, but it does not help in the case of a drifting local clock, e. g., due to a temperature step [Mil98].

A solution to the stability-versus-accuracy tradeoff problem has been described by Ridoux and Veitch [RV09]. They co-implemented two software clocks based on the same CPU clock hardware oscillator. One clock is optimized for time differences and the other for absolute time.

[52] In FlexRay if $3 \leq n \leq 7$ then $f = 1$, and if $n > 7$ then $f = 2$.

3. Synchronization Protocols for Computer Networks

There is a huge body of work about synchronization protocols for computer networks. Countless protocols have been proposed in thousands of publications. The presentation and citations in this chapter are therefore restricted to a few classical papers and some survey papers. Much of the theoretical work has been published throughout the 1980-ies and early 1990-ies. An (incomplete) annotated bibliography on clock synchronization from that period is [Sch94a], an overview paper is [SWL90].

Currently the subject of time synchronization protocols is again a hot research topic. The main reason is that classical approaches do not fit the highly specific demands of wireless sensor networks very well [ER03]. Pointers to WSN research literature are given in section 3.2.6. Although the WSN design space is vast [RM04], I-SENSE does not fit into it.[1] The advantages of WSN protocols in areas like energy-efficiency, scalability, and ad hoc deployment—while interesting—are of little relevance for the practical part of this thesis (cf. section 1.2).

Section 3.1 below is an annotated compilation of protocol classification criteria. It gives an overview over the synchronization protocol design space. Section 3.2 briefly presents some examples of practical algorithms and gives pointers to literature. Section 3.3 concludes this chapter with a more detailed description of the NTP and SNTP protocols.

3.1. Classification of Synchronization Protocols

3.1.1. Communication model

The communication patterns for synchronizing time sources with clients have been discussed in section 2.5.1. The common-view method requires a physical broadcast channel. The one-way method can use broadcast addressing, but does not require it. The two-way method normally uses unicast, but a broadcast request with unicast replies can be used too.

[1] Current technology cannot support the energy requirements of I-SENSE nodes from reasonable sized batteries. I-SENSE nodes are therefore tethered devices.

3.1.2. Time source

No time source The simplest form of synchronization is only concerned with *event ordering*. These algorithms only compare clocks and do not attempt to synchronize them. Algorithms without any physical clocks at all (cf. section 3.2.1) fall in this category too.

Internal synchronization Without an external time source the goal is *consistency* among the network nodes., i. e., to minimize the differences between the readings of the local clocks.

External Synchronization Time is supplied from outside the network. The injected *reference time* usually refers to a standardized global timescale like UTC. All nodes adjust to this reference time. While internal synchronization *may* be implemented in peer-to-peer fashion, external synchronization is always hierarchical.

3.1.3. Clock correction versus timescale transformation

Clock correction Most methods control rates and offsets of all clocks. The clock of each node is corrected to achieve network synchronization.

Untethered clocks Each clock is free running, i. e., not adjusted or steered. Because each clock implements its own timescale, timestamps must be converted when passed between nodes. These *timescale transformations* need parameters relating the clocks against each other. The storage overhead for these parameters and the computation overhead for maintaining parameters and doing timestamp transformations are a disadvantage of the approach. It is nonetheless attractive in the context of WSNs, where energy efficiency is paramount and wireless communication is the most energy consuming activity. Timescale transformation is attractive in these scenarios, because it significantly reduces the communication overhead for synchronization.

3.1.4. Master-slave versus peer-to-peer

Master-slave Master nodes require CPU and bandwidth resources proportional to the number of slaves. Large networks therefore use hierarchical topologies, where certain higher level slaves are masters for the next level.[2]

Peer-to-peer Any node is allowed to talk to any other node directly. Peer-to-peer protocols are more flexible and node-failure tolerant than hierarchical protocols, but also more difficult to control.

[2] In North-American telecommunication networks (cf. ANSI/T1.101-1987) the hierarchical synchronization levels are called *strata* (sg: *stratum*). The NTP documentation uses that term too, but, unlike the ANSI standard, a NTP stratum carries only topological significance and no accuracy and stability requirements are defined for the levels.

3.1.5. Probabilistic versus deterministic

Probabilistic synchronization Probabilistic clock synchronization algorithms do not require guaranteed bounds on message delays. Probabilistic protocols cannot guarantee a maximum clock offset, but the probability of exceeding an offset can be bounded or determined.[3]

Deterministic synchronization Deterministic algorithms can guarantee bounds on clock offset. Most algorithms in the literature are of this type. In order to make deterministic guarantees these protocols have to make stringent assumptions about the properties of the involved clocks and communication channels, e. g., lower and upper bounds for the channel delay and clock skew must be known and guaranteed.

3.1.6. Time instants versus time intervals

Time instants A time instant (like $t = 3$) specifies a zero dimensional quantity with no margin for error. Since measurement uncertainty is unavoidable, a time instant can never be *exactly* accurate. Both time instants and intervals can be refined by quality statements. The processing of detailed quality statements (like the probability distribution of offset error) would make protocols quite complex. Therefore symmetric or asymmetric intervals together with a probability for the correctness of the given interval are preferred.

Time intervals Using guaranteed intervals instead of instants allows to combine the intervals efficiently and unambiguously by intersection. Guaranteed time bounds for sensor-data permit to guarantee bounds on fusion results too.

3.1.7. Lifetime and scope

Most protocols are designed for *continuous synchronization*. For sensor network applications this may be suboptimal. If inter-event intervals are long then *on-demand synchronization* may be much more efficient. An example is *post-facto synchronization* where synchronization and timestamp generation happen only after an event has been recorded [EGE02].

Only the collocated *subset* of network nodes that observes the triggering event needs to participate in the synchronization.

3.1.8. Low level access

Standard approach Time stamping is done at the application layer. This avoids modifications to low level software, but inevitable system noise degrades timestamp quality. Some operating systems have built-in timestamping facilities implemented in the network stack.[4] When these facilities are available and used then application scheduling delay is entirely removed from timestamps.

[3] Some authors make a further distinction between *probabilistic* and *statistical* algorithms. [AP98]

[4] Among them Solaris, Linux, and the open source BSDs. Windows lacks this feature. The reference implementation of NTP makes use of OS-provided UDP packet timestamps where available.

MAC-layer utilization Timestamping at application or network stack level is agnostic to media access delay. Several WSN time synchronization protocols rely on direct access to the *Media Access Control* (MAC) layer.

Low Level Hardware Timestamps Creating timestamps at the lowest possible software layer (device driver / MAC) still incurs the timing uncertainties associated with interrupt latency. Sub microsecond accuracies are possible with special network interface hardware. Examples of this approach are the SynUTC project where a special UTCSU-ASIC (Universal Time Coordinated Synchronization Unit, see [Loy97]) and an asymmetrical interval based protocol [Sch00] were developed. A newer example is the IEEE 1588 standard, which specifies the Precision Time Protocol (PTP) [IEE08]. It is commonly implemented with special Ethernet hardware, which snoops the IEEE 802.3 MII-bus to the PHY transceiver chip [Hor04, pp. 55–60]. IEEE 1588 is aimed at real time applications in automation [HSK03].[5]

3.2. Synchronization Protocol Survey

3.2.1. Logical clocks

Lamport points out in [Lam78] that the concept of time is derived from the more basic concept of event ordering. He models distributed systems as collections of processes. Each process consists of a sequence of events, e. g., the execution of a machine instruction or sending a message to another process. He formalizes the concept of event ordering with the definition of the "happened-before" relation, denoted by "$\rightarrow$" and defined as follows:

Definition 1. The relation "$\rightarrow$" on the set of events of a system is the smallest relation satisfying the following three conditions:

1. If a and b are events in the same process, and a comes before b, then $a \rightarrow b$.
2. If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$.
3. If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

Two distinct events a and b are said to be *concurrent*, denoted by $a \parallel b$, if $a \nrightarrow b$ and $b \nrightarrow a$.

No event can happen before itself, therefore $a \nrightarrow a$ holds for any event a. Thus the "happened-before" relation is irreflexive, transitive, and antisymmetric. It defines a partial ordering on the set of all events.

Another meaning of $a \rightarrow b$ is that event a may *causally affect* event b, and $a \parallel b$ means that neither can causally affect the other.[6] Two distributed computations are equivalent (have the same effect) if they only differ by the order of concurrent operations.

[5] See `http://www.ieee1588.com/` and `http://ieee1588.nist.gov/`. Software only implementations of PTP exist too [CBB05].

[6] The above definition gives a causality relationship for message passing systems. In shared memory systems two operations on the same data item, one of which is a write, are causally related too.

Scalar clocks

Lamport introduces logical clocks in [Lam78] as just a way of assigning numbers to events, i. e., each process P_i has a clock C_i which assigns a number $C_i(a)$ to any event a in P_i. For any event b we define its timestamp $C(b)$ to be $C_j(b)$ if b is an event in P_j. A reasonable correctness condition for logical clocks is that if event a happens before event b, then the clock value (timestamp) of a must be less than the clock value of b. We state this more formally as follows:[7]

Consistency condition. If $a \rightarrow b$ then $C(a) < C(b)$

Logical clocks satisfying this condition are remarkably easy to implement. Only per-process counters and a timestamp field attached to messages are needed. The following two implementation rules are sufficient:

1. Each process P_i increments C_i between successive events.
2. If a is the sending of a message m by process P_i to P_j, then the message contains a timestamp $T_m = C_i(a)$. When P_j receives m it sets $C_j := \max(C_j, T_m + 1)$.

The partial ordering by "$\rightarrow$" can be completed to a total ordering "$\Rightarrow$". We can e. g., define that $a \Rightarrow b$ if and only if either $C(a) < C(b)$ or $C_i(a) = C_j(b)$ and $i < j$.[8] A total ordering of events has many uses in distributed systems, it can e. g., be used to implement synchronized access to shared resources.

Vector clocks

Logical clocks satisfying the consistency condition have a limitation. Comparing timestamps cannot confirm a "happened before" relation. While $C(a) < C(b)$ implies $b \nrightarrow a$ it does not tell us whether events a and b are causally related ($a \rightarrow b$) or concurrent ($a \parallel b$). With the following stronger consistency condition this is possible:

Strong consistency condition. If $a \rightarrow b$ then $C(a) < C(b)$ and if $C(a) < C(b)$ then $a \rightarrow b$

Vector clocks satisfying the stronger consistency condition have been independently developed by Fidge, Mattern, and Schmuck in 1988 [Fid88, Mat89]. With n processes each process P_i maintains a vector $vt_i[1..n]$. The own clock of P_i is the $vt_i[i]$ element. Element $vt_i[k]$ where $k \neq i$ represents the latest knowledge of P_i about the logical clock value of process P_k. Again only two implementation rules are required:

1. Process P_i updates its own clock before each local event $\quad vt_i[i] := vt_i[i] + d \quad (d > 0)$
2. Each process that is sending a message attaches the whole vector to the outgoing message. The receiving process P_i updates its vector clock as follows before it delivers the message:

$$vt_i[k] := \max(vt_i[k], vt_{msg}[k]) \qquad \text{where } 1 \leq k \leq n$$
$$vt_i[i] := \max(vt_i[k]) + d$$

[7] Adding the converse condition—if $a \nrightarrow b$ then $C(a) \not< C(b)$—would imply that any two concurrent events have to occur at the same logical time.

[8] The rule to handle the case when $C(a) = C(b)$ is arbitrary. Any relation $P_i \prec P_j$, which orders processes totally, is sufficient.

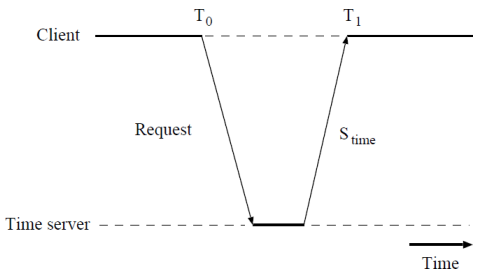


Figure 3.1.: Cristian's remote clock reading method [SBK05]

Comparison of two vector timestamps *vh* and *vk* is defined as follows.

$$\begin{aligned} vh &\leq vk \quad \text{if} \quad \forall x : vh[x] \leq vk[x] \\ vh &< vk \quad \text{if} \quad vh \leq vk \text{ and } \exists x : vh[x] < vk[x] \\ vh &\parallel vk \quad \text{if} \quad vh \not\leq vk \text{ and } vk \not\leq vh \end{aligned}$$

Straightforward implementations of vector clocks scale badly with system size n, because each message piggybacks n integers. More efficient implementations of vector clocks are known.

The differential technique of Singhal and Kshemkalyani sends only changed elements of the vt_i clock vector in the form of $(index, new\ counter)$ tuples. A direct implementation would need $O(n^2)$ storage to keep the latest vectors sent to each process. The scheme in [SK92] uses only two vectors[9], a "last sent" vector $LS[1..n]$ and a "last updated" vector $LU[1..n]$. Process P_i keeps in $LS_i[j]$ his $vt_i[i]$ timestamp from the last message sent to P_j. $LU_i[k]$ contains the $vt_i[i]$ value from the last update to $vt_i[k]$. Process P_i piggybacks only $\{(k, vt_i[k]) \,|\, LS_i[j] < LU_i[k]\}$ to messages it sends to P_j.

Matrix clocks

The matrix clock is a third type of logical clock besides the scalar and vector clock. A matrix clock not only keeps state about the current logical time of other processes, it also stores information about what those other processes know about each others clocks. This is useful to identify (and delete) obsolete information. Matrix clocks therefore have applications in areas like distributed checkpointing and garbage collection. Efficient implementations exist, which reduce the $O(n^2)$ storage and bandwidth complexities of a straightforward approach.

3.2.2. Cristian's algorithm

Cristian describes in [Cri89] a simple probabilistic algorithm. The basic principle is depicted in fig. 3.1. The *remote clock reading* method assumes that round-trip times are short compared to the required accuracy. In order to read the clock of another node, a process initiates at local time T_0 a timestamp request. The queried time server replies with a message containing the server side timestamp S_{time}.

[9] Not counting the clock vector vt_i itself.

When the reply is received the round-trip time is calculated as the difference $T_1 - T_0$. The receiver then estimates the server time S_1 corresponding to T_1 as

$$S_1 = S_{time} + \frac{T_1 - T_0}{2} \tag{3.1}$$

This assumes that the server timestamp was created half in between the round-trip at local time $T_0 + (T_1 - T_0)/2$.

Cristian observed that with his method the offset error θ is bounded by the half round-trip time.[10]

$$|\theta| \leq \frac{(T_1 - T_0)}{2} \tag{3.2}$$

In Cristian's scheme several clients are synchronized to an accurate time server using the remote clock reading method. To improve accuracy several round-trips are made and only the one with the least round-trip time is used.

3.2.3. The Berkeley algorithm

The Berkeley algorithm [GZ89] is like Cristian's algorithm a master-slave protocol intended for use within intranets. All computers run an instance of the `timed` daemon.[11] Through an election algorithm one of them is chosen as the master.[12]

The master reads the clocks of the slaves with Cristian's remote clock reading method. A fault tolerant averaging function is then used on the $C_{slave} - C_{master}$ clock offsets. Fault tolerance is achieved by averaging only the largest set of client clocks, which do not differ by more than a small constant γ from each other. Finally the master asks each client to adjust its clock for the $C_{avg} - C_{slave}$ difference. The algorithm is repeated in regular intervals. The clock of the master plays no privileged role in this internal synchronization algorithm.

3.2.4. Marzullo's algorithm

Marzullo and Owicki considered distributed systems with clocks that differ in rate accuracy [MO83]. Their augmentation of timestamps with an indication of their accuracy lead to an interval based algorithm.

Their error model assumes known upper bounds for clock skew S (cf. eq. 2.9) and for one-way message delay. Because of skew the confidence intervals of clocks grow linearly with the time since the last clock reset. To minimize the maximum error, interval intersection is used. If all clocks are correct, the intersection of their confidence intervals must give a nonempty interval. If not, a nonempty intersection interval is sought including all but a minimum number of incorrect clocks.

[10] Rate differences between the clocks increase $|\theta|_{max}$ slightly.

[11] In Unix (and similar operating systems) a *daemon* is a server process running in the background without interactive user control. By convention daemons have names ending with the letter 'd'.

[12] Should the master fail, a new election takes place after a timeout.

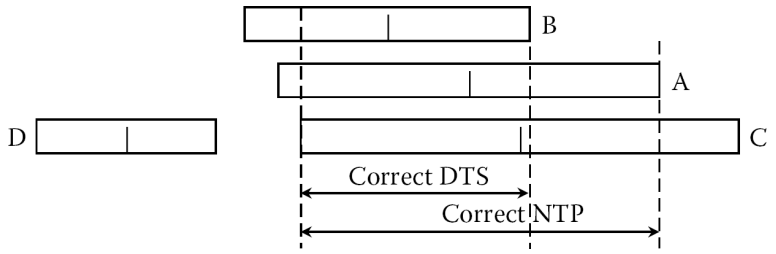


Figure 3.2.: Marzullo's interval intersections [Mil06b, p. 48]

An example with $m = 4$ confidence intervals is depicted in fig. 3.2. The algorithm finds the smallest intersection interval that contains points of each of $m - f$ confidence intervals, where f is the number of incorrect clocks or *falsetickers*. The algorithm requires that $f < m/2$.

There are some border cases where Marzullo's algorithm produces anomalistic results. A modified version of the algorithm is thus used in NTP. The NTP version requires that the intersection of $m - f$ intervals contains at least $m - f$ midpoints. The intersection generated by the NTP selection algorithm is shown in fig. 3.2 too. The modified algorithm always results in an interval that includes the interval produced by the original algorithm.

The asymptotic space usage of the algorithm is $O(m)$. The time efficiency is $O(m \log m)$ for the first run, and $O(m)$ when only one source interval is updated.

3.2.5. Fault tolerant protocols

Many fault tolerant clock synchronization protocols have been proposed. Some examples are described in [LL84, DHS84, LMS85]. Most of the literature refers to the Byzantine generals problem described in [LSP82].

In analogy to the solutions given in [LSP82], without authenticated timestamp messages fault tolerance is possible if and only if less than one third of the clocks fail.

The proposed deterministic protocols need a relatively high communication overhead in order to achieve fault tolerance and a sufficient level of precision.

3.2.6. Protocols for wireless sensor networks

Wireless sensor networks (WSN) differ in many aspects from traditional distributed systems [RM04]. Protocols like NTP, which work well in the Internet, are not suitable for most WSN environments. NTP was designed for large-scale networks with a rather static topology. NTP servers have to process synchronization requests at all times. Energy efficient time division schemes, which would allow servers to enter a low-power state outside assigned time slots, are therefore not possible with NTP.

The IEEE 801.11 standards already provide a time synchronization mechanism for the so-called infrastructure mode. The mechanism uses a master/slave model of communication. The fixed master node is the access point. To coordinate medium access for all reachable stations, the access point

periodically sends a high-priority *beacon frame*.[13] The beacon frame includes a timestamp for synchronizing the clocks of the slaves.

The IEEE 801.11 scheme uses the one-way method described in section 2.5.1. This adds the uncertainty of the time interval between the taking of the timestamp at the access point and the transmission of the beacon frame to the time transfer. This uncertainty is orders of magnitude larger than the physical signal propagation delay of WLANs. The sender-side uncertainty is eliminated by the common-view method. Several authors have proposed common-view based synchronization protocols for WSNs, e. g., the Reference Broadcast Synchronization Protocol (RBS) of Elson et. al [EGE02].

Many other protocols (using all three methods of section 2.5.1) for WSNs have been published. In general they focus on energy efficiency, low storage and processing resources, and suitability for dynamical and/or ad-hoc topologies. Two survey papers are [SY04] (a comparison of the RBS, TPSN, Tiny-Sync/Mini-Sync, and LTS protocols) and [SBK05], the latter is a survey and analysis of nine existing clock synchronization protocols for wireless sensor networks. A section of [RBM05]—a good introductory text—also examines ten current synchronization algorithms.

3.3. The Network Time Protocol

3.3.1. NTP classification

NTP can be classified according to the criteria of section 3.1 as

- using either one-way or two-way time transfer or both (dependent on configuration)
- doing external synchronization
- adjusting the operating system clock
- using (mostly) hierarchical master/slave topologies[14]
- using probabilistic algorithms
- using maximum likelihood and worst case intervals
- operating continuously
- utilizing lower level timestamps where provided by the OS.

[13] The beacon frame marks the switch from a *contention period* with CSMA based medium arbitration to a *contention free period* with centralized arbitration.

[14] The NTP symmetric mode is used between two equal peers.

3.3.2. History and background

NTP is claimed to be the longest running, continuously operating, distributed application in the Internet [Mil06b, p. 253]. A first protocol specification appeared 1981 in RFC 778 [Mil81]. As the protocol evolved revised specifications appeared.

NTPv0 The (now so called) NTP version 0 was implemented an documented in 1985 [Mil85].

NTPv1 Version 1 of the NTP specification appeared three years later and defined client/server and symmetric modes [Mil88].

NTPv2 In 1989 the NTP version 2 specification introduced a formal model and state machine describing the protocol. The NTP Control Message Protocol for management of NTP servers and clients had been added. Another novelty was a cryptographic authentication scheme based on symmetric key cryptography [Mil89].

NTPv3 The NTP version 3 specification appeared 1992 as RFC 1305. A modified version of Marzullo's interval-based agreement algorithm ([MO83]) had been integrated. The appendix includes a formal error analysis. The error budget calculation includes all error contributions between the reference clock over intervening servers to the eventual time service client. Based on this error model NTPv3 implementations provide maximum error and expected error statistics. These statistics are used by the protocol as a metric for selecting the best server from a group of available servers. Also a broadcast mode of operation (intended for use on LANs) had been added. NTPv3 and NTPv2 implementations can inter-operate in a time distribution network [Mil92].

NTPv4 The NTP protocol has evolved since NTPv3. New features and algorithm revisions have been added while interoperability with older versions has been preserved. NTPv4 has not been officially adopted by the IETF yet.

NTPv4 accommodates Internet Protocol version 6 by means of a modified protocol header. It has also got a new security model and a self-configuring protocol called Autokey [Mil06a]. While the NTPv2 authentication has worked well, it suffers (like all symmetric key schemes) from complicated key distribution. Autokey uses a combination of public key cryptography for signing timestamps and a computationally less expensive pseudo-random keystream for authenticating packets relative to the signed timestamp values.[15]

Other additions are a manycast mode, which permits client to discover nearby servers autonomously, improvements to the clock discipline algorithm and (on some operating systems) implementation of parts of it directly in the kernel [Mil06c].[16]

[15] The rationale behind this design is to minimize the impact of CPU time (spent in asymmetric cryptography routines) on the quality of timekeeping.

[16] With a kernel PLL interfaced to an external reference clock the residual time error is on the order of 50 ns [MK00].

3.3.3. NTP implementations

The official NTP reference implementation along with documentation is available as open source code releases (for UNIX-like systems) from the NTP project homepage[17]. The code is published by the University of Delaware under a permissive BSD-style open source license. Dave Mill's many NTP-related publications are available on his homepage[18].

The OpenNTPD project[19] is a subproject of the OpenBSD project. The goal of the project is a free, easy to use implementation of the Network Time Protocol. The project has a focus on security, simplicity of implementation and configuration, and reasonable accuracy.[20]. There are two OpenNTPD teams. One team does strictly OpenBSD-based development. Another team then takes the OpenBSD version and makes it portable to other POSIX systems. Currently 10 operating systems are supported.

Other free third-party implementations, including pre-compiled versions for Windows NT / 2000 / XP, Windows Server 2003, and Vista, are available (e. g., through links at the `ntp.org` homepage) too.

Microsoft Windows versions since Windows 2000 include the Windows Time Service (W32Time). How Windows Time Service works is documented on Microsoft TechNet.[21] The Microsoft implementations in Windows 2000 and Windows XP work only as SNTP clients. Windows versions since Windows 2003 are documented by Microsoft to be compliant to [Mil92] (NTPv3). Windows systems do not interpolate time between timer interrupts (ticks). This alone limits accuracy to ± 16 ms (64 Hz). Moreover a Microsoft support document titled *"Support boundary to configure the Windows Time service for high accuracy environments"*[22] states:

> We do not guarantee and we do not support the accuracy of the W32Time service between nodes on a network. The W32Time service is not a full-featured NTP solution that meets time-sensitive application needs. The W32Time service is primarily designed to do the following:
>
> - Make the Kerberos version 5 authentication protocol work.
> - Provide loose sync time for client computers.
>
> The W32Time service cannot reliably maintain sync time to the range of 1 to 2 seconds. Such tolerances are outside the design specification of the W32Time service.

3.3.4. NTP sub-algorithms

The presentation of this section follows [Mil06b, Mil06c] and describes NTPv4.

A NTP architecture overview is shown in fig. 3.3. The depicted process decomposition reflects the internal software organization only. It is *not* mapped to operating system level processes. The NTP reference implementation uses a single operating system process.

---

[17] `http://www.ntp.org/`
[18] `http://www.eecis.udel.edu/~mills/`
[19] `http://www.openntpd.org/`
[20] The project webpage states: "We are not after the last microseconds."
[21] `http://technet.microsoft.com/en-us/library/cc773013(WS.10).aspx`
[22] `http://support.microsoft.com/kb/939322`

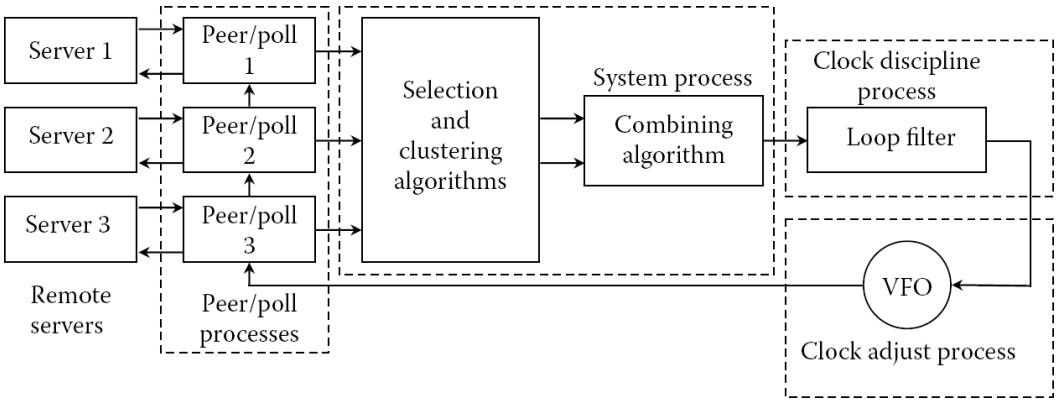


Figure 3.3.: NTP architecture overview [Mil06b, p. 19]

The communication with remote servers and local reference clocks[23] is handled by peer/poll processes. Incoming clock samples on UDP/IP port 123 are filtered by the clock filter algorithm. The selection algorithm separates the server population into *truechimers* and *falsetickers*. If more than three truechimers are left the clustering algorithm casts off *outliers*. Offsets from the remaining *survivors* are combined by the combining algorithm into the final system offset Θ that is input to the clock discipline algorithm.

NTP timestamp exchange protocol

The exchange of NTP timestamps is shown in fig. 3.4. Depicted is the most general form with symmetric modes. Other than Cristian's remote clock reading method (cf. fig. 3.1) four timestamps are taken per round trip and the scheme is symmetric. In client/server mode the server only copies two timestamps and adds a new one, but it does not need to save any client state.

The round trip communication channel delay (RTTD) δ is

$$\delta = (T_{n+3} - T_n) - (T_{n+2} - T_{n+1}) \tag{3.3}$$

and with the usual assumption of a symmetric channel the offset θ is

$$\theta = \frac{1}{2}\left[(T_{n+1} - T_n) + (T_{n+2} - T_{n+3})\right] \tag{3.4}$$

Time stamp subtractions are done in 64 bit integer arithmetic to preserve precision. This leaves a signed 31 bit integer seconds value. A NTP client must therefore know the time to within ± 68 years by means outside the NTP protocol. The reference implementation uses floating point arithmetic for all further processing of the resulting time differences.

[23] Local reference clocks are not shown in fig. 3.3. Drivers present them to the core software as additional servers with zero channel delay.

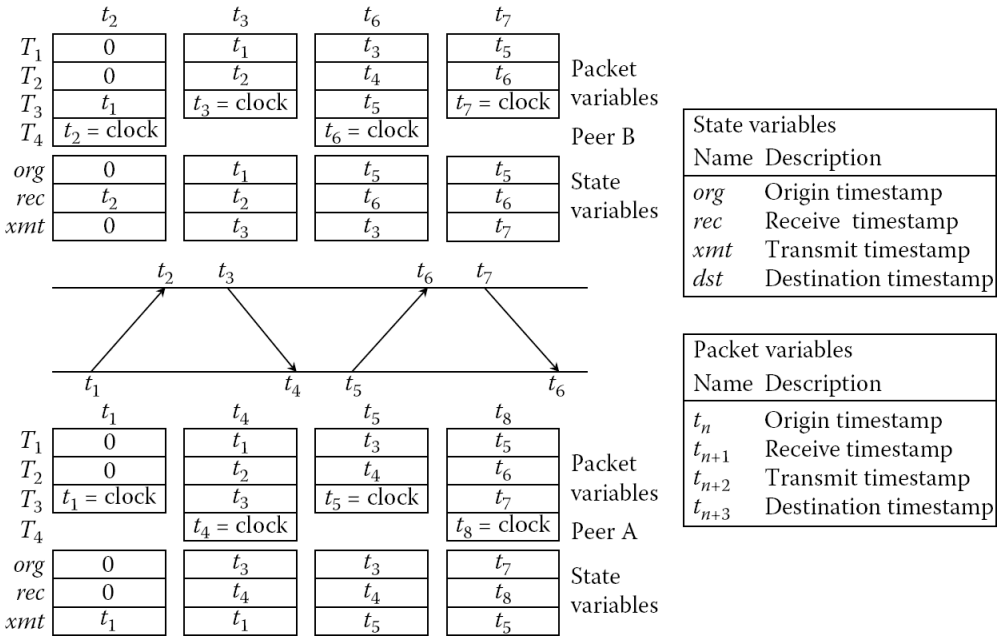


Figure 3.4.: NTP timestamp exchange [Mil06b, p. 42]

NTP clock filter algorithm

The NTP clock filter algorithm stores the latest eight valid samples of each peer in a shift register. Its output is the sample with the minimum RTTD δ. The method is based on eq. 2.22 and is explained in section 2.5.2 above.

The 8-stage shift register actually stores $(\theta, \delta, \varepsilon, t)$ tuples. The *dispersion* $\varepsilon(t)$ represents the maximum error due to frequency tolerance and clock reading precision.

$$\varepsilon(t) = \rho_R + \rho + \Phi(t - t_0) \tag{3.5}$$

Here ρ is the minimum time needed to read the system clock and called *precision* in NTP terminology.[24] The maximum frequency tolerance of clocks Φ is a configurable constant and defaults to 15 ppm.[25] The peer precision ρ_R is taken from the NTP packet header. The dispersion grows linearly with the elapsed time since t_0, i. e., since the timestamp in question was taken.

The clock filter algorithm sorts the eight tuples by increasing δ and averages the sampled ε values to the *peer dispersion*[26]

$$\varepsilon = \sum_{k=1}^{8} \frac{\varepsilon_k}{2^k} \tag{3.6}$$

[24] This definition of precision is unfortunately different from the definition used in metrology and given in the glossary.

[25] The 15 ppm number comes from the specification of Digital Alpha machines. It is quite arbitrary as undisciplined computer clock tolerances can be up to 500 ppm in extreme cases. [Mil06b, p. 177]

[26] As in the NTP documentation, the time dependency of ε is not made explicit by the notation.

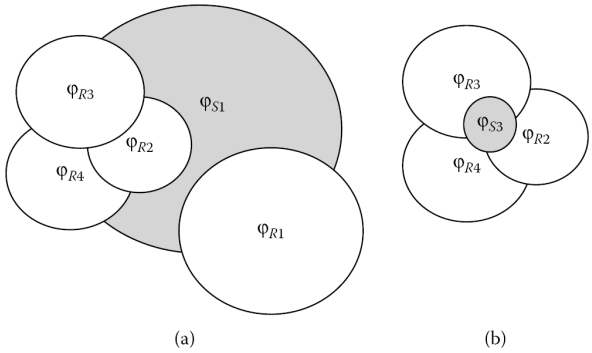


Figure 3.5.: NTP clustering algorithm example [Mil06b, p. 52]

The sorted list of n valid tuples is further used to compute the *peer jitter* φ_P

$$\varphi_P = \sqrt{\frac{1}{n-1}\sum_{k=2}^{n}(\theta_k - \theta_1)^2} \tag{3.7}$$

NTP selection algorithm

The NTP selection algorithm is variant of Marzullo's algorithm and has been sketched in section 3.2.4. The correctness intervals that are processed by the algorithm are of the form $[\theta - \lambda, \theta + \lambda]$ where λ is the *peer synchronization distance*

$$\lambda = \frac{(\Delta_R + \delta)}{2} + E_R + \varepsilon + \varphi_P \tag{3.8}$$

The variables Δ_R and E_R are the accumulated *root delay* and *root dispersion*, respectively. They are provided by the server in packet header fields.

NTP clustering algorithm

The truechimers from the selection algorithm are put on a *survivor* list and processed in a series of rounds by the clustering algorithm. Each round removes a statistical *outlier* until either only $NMIN = 3$ survivors are left or no further improvement is possible.

For each of the n survivors the *selection jitter* $\varphi_{S,i}$ is computed

$$\varphi_{S,i} = \sqrt{\frac{1}{n-1}\sum_{k=1}^{n}(\theta_k - \theta_i)^2} \tag{3.9}$$

Then $\varphi_{max} = \max(\varphi_S)$ and $\varphi_{min} = \min(\varphi_P)$ are determined. If $\varphi_{max} < \varphi_{min}$ or $n < NMIN$ no further improvement is possible and the algorithm terminates. Otherwise, the peer with the highest selection jitter $\varphi_{S,i}$ is removed from the survivor list, n is decremented, and a new round starts.

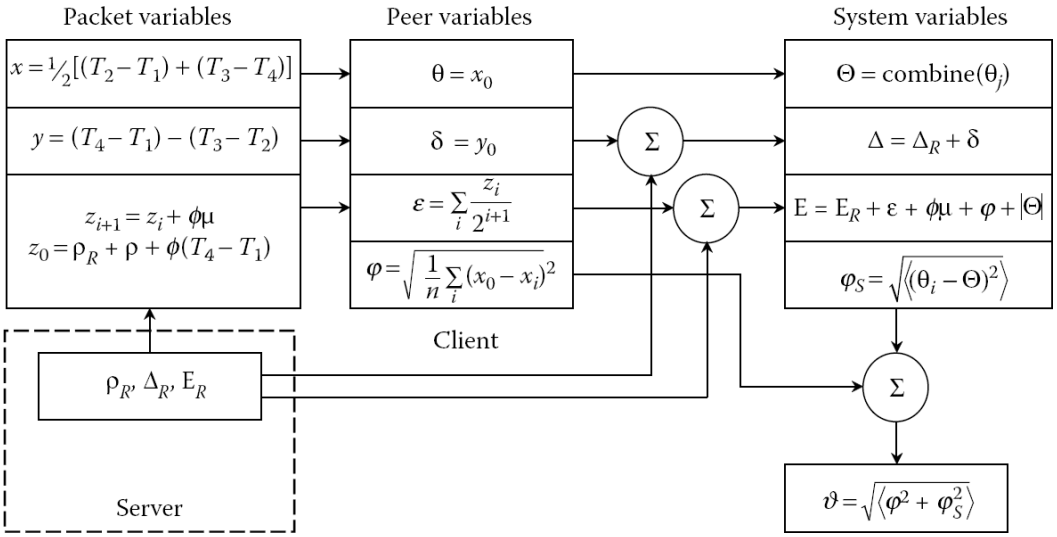


Figure 3.6.: NTP error budget calculations [Mil06b, p. 184]

The algorithm is illustrated by fig. 3.5. The diameters of the white circles represent φ_P and the gray circle represents φ_S. In fig. 3.5a the gray φ_{max} is greater than the smallest φ_P and peer 1 is removed. The largest remaining selection jitter in fig. 3.5b is smaller than each φ_P, so the algorithm would even terminate with $NMIN < 3$.

NTP combining algorithm

The survivors are used to form the combined *system offset* Θ

$$\Theta = q \sum_i \frac{\theta_i}{\lambda_i} \tag{3.10}$$

and combined peer jitter φ_r

$$\varphi_r = \sqrt{q \sum_i \frac{\varphi_{P,i}^2}{\lambda_i}} \tag{3.11}$$

where q is the normalizing factor

$$q = \left(\sum_i \frac{1}{\lambda_i} \right)^{-1} \tag{3.12}$$

The survivor u with the lowest stratum and peer synchronization distance λ provides the best statistics for performance evaluation and is promoted to the *system peer*. The *system jitter* ϑ is then computed as

$$\vartheta = \sqrt{\varphi_r^2 + \varphi_{S,u}^2} \tag{3.13}$$

The system jitter represents the best clock offset error estimate (or *nominal error* statistic) and is available to application programs.

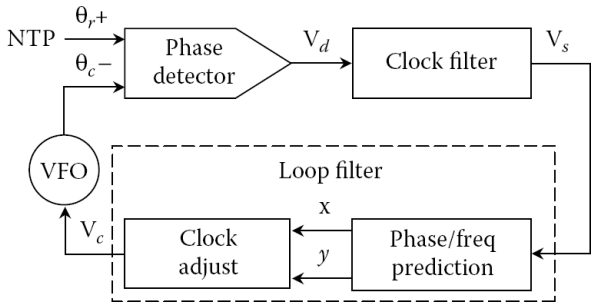


Figure 3.7.: NTP clock discipline algorithm [Mil06b, p. 65]

The computation of the system variables is summarized in fig. 3.6. The *root delay* Δ

$$\Delta = \Delta_R + \delta \tag{3.14}$$

is the sum of all delays from the stratum 1 synchronization source down the path through the time distribution hierarchy. The *root dispersion* E (with sampling interval μ) is

$$E = E_R + \varepsilon + \Phi\mu + \varphi_{P,u} + |\Theta| \tag{3.15}$$

and is also inherited along the path from the primary server. Both Δ and E are passed down the hierarchy to the next stratum as packet header variables Δ_R and E_R, respectively (cf. eq. 3.8).

Finally the *system synchronization distance* Λ

$$\Lambda = \frac{\Delta}{2} + E \tag{3.16}$$

is provided to dependent applications as the *maximum error* statistic.

NTP clock discipline algorithm

The NTP clock discipline algorithm synchronizes the computer clock with respect to the system offset Θ. In principle, it could be used with any protocol that provides periodic time corrections. On some POSIX systems, parts of the clock discipline have been implemented for highest accuracy inside the operating system kernel.[27] The algorithm has evolved to a complex self-adapting state machine and has been extensively tested with network simulators. This section gives only a coarse overview. More detail is available from [Mil06b, pp. 63–76] and [Mil98].

The structure of the clock discipline is shown in fig. 3.7. The loop filter is implemented using two sub-algorithms, a phase and a frequency locked loop. This hybrid PLL/FLL design had been originally suggested by Levine [Lev95] and was adopted by NTP. It is based on the observation that a PLL usually works better with shorter update intervals when network jitter dominates, while a FLL works better with long intervals when oscillator frequency wander dominates.

[27] This part is often referred to as the *kernel PLL*.

NTP uses poll intervals that are powers of two. NTPv3 uses default *poll exponents* τ between 6 (64 s) and 10 (1024 s). In NTPv4 τ can range from 4 (16 s) to 17 (131072 s). The poll exponent is dynamically adjusted. A counter is incremented when $\Theta < 4\varphi_P$ and decremented otherwise.[28] When the counter reaches an upper or lower threshold, τ is incremented or decremented, respectively. This scheme adapts to varying network conditions but does not catch oscillator wander.[29]

The clock discipline has to use relatively large time constants T_c. The time constant depends on the poll interval 2^τ. PLL stability requires that T_c is greater than twice the total loop delay. The 8-stage shift register of the clock filter algorithm can cause a loop delay of $8 \times 2^\tau$. With the default initial poll interval of 64 s it follows that $T_c \geq 2 \times 8 \times 64 = 1024$ s. With that T_c, a 63 % PLL response to a frequency step takes 4.25 hours. After the NTP daemon has started, it could take many hours to adapt to the frequency offset of the local clock.[30] The reference implementation therefore saves the frequency offset in a file once each hour.

Since time constants are large and POSIX system clocks typically provide only 1.8 s/hour amortization (500 ppm), the reference implementation defines a *step threshold* and a *stepout threshold*. The clock will be stepped (even backwards) when a system offset Θ greater than the step threshold persists for a duration greater than the stepout threshold. The default for the step threshold is 128 ms, and the default for the stepout threshold is 15 min.

3.3.5. The Simple Network Time Protocol

The NTP reference application is a relatively complex application with many elaborate sub-algorithms. The Simple Network Time Protocol (SNTP) is a subset of NTP that can be used when the performance of a full NTP implementation is neither needed nor justified. The current version is SNTPv4 defined in RFC 4330 [Mil06d], which obsoletes SNTPv3 defined in RFC 1769 [Mil95]. SNTPv4 implementations can inter-operate with both NTPv3 and NTPv4.

The major simplification of SNTP is that exactly one time source is used. SNTP servers are typically dedicated products that include a reference clock, usually a GPS time receiver or a radio clock. SNTP clients rely on a single NTP or SNTP server and cannot have clients of their own. Because there is only one time source, SNTP does not need any of the selection, clustering, and combining algorithms.

SNTP clients are not fault tolerant. Failure of the server or the network path to the server cause desynchronization.

An SNTP clients can receive the address of its server automatically, either an IPv4 address via the DHCP protocol [AD97] or an IPv6 address via DHCPv6 [Kal05].

Several free SNTP implementations are available. A new reference implementation of the SNTPv4 client was written 2008. It is available as a part of the NTP reference implementation and shares much code with it.

[28] The coefficient 4 has been determined through simulations.

[29] Moderate clock skew does not increase the measured peer jitter much, hence poll interval reduction is delayed until the clock has accumulated an offset error $\Theta \geq 4\varphi_P$.

[30] At the time of this writing the `ntp.org` development version—but not the stable version—has a quick-start algorithm that computes a first clock skew estimation after 15 minutes.

4. Design and Implementation of Time-Synchronization for the I-SENSE Framework

4.1. I-SENSE Architecture Overview

4.1.1. Hardware architecture

The I-SENSE platform is a network of geographically distributed sensor nodes which are connected via a common communication medium. Typically wired Ethernet is used, but other network technologies such as WLAN can be used as well [KRT06]. The structure of an I-SENSE node is shown in fig. 1.1 on page 3. The sensor nodes are heterogeneous multi-processor systems. Digital signal processors do most of the sensor data processing and a general purpose CPU handles communication and management tasks.

Fig. 4.1 shows a photo of an I-SENSE node. The general purpose component is an embedded Pentium-M PC conforming to the PICMG[1] 1.2 specification. It provides PC-typical interfaces (Video, USB, Serial, Parallel, PS/2 mouse and keyboard, ATA), two 100Base-Tx Ethernet controllers and a Compact Flash disc.

The video encoder card contains two TMS320DM642 video/imaging fixed-point digital signal processors. Each DSP can process up to six ITU-R BT.656 video streams without multiplexing. The signal processors share only the power supply, clock signal generator, and the PCI bridge. The memory, buses and peripherals are not shared [Put04].[2] Each DSP has access to 128 MiB of on-board DRAM.[3]

The DSPs have no network interface hardware attached. All intra-node inter-processor communication uses the PCI bus. Inter-node (Ethernet) communication is carried out only by general purpose processors.

Timer hardware

The hardware clock facilities of the general purpose component are those of the PC platform and are described in section 2.3.3 on page 17. Direct access to the timer hardware is reserved for the operating system running on the x86 CPU.

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

[2] The general purpose processor can access the on-board memory via PCI (cf. section 4.1.3).

[3] Several other models of TMS320C64X boards with TMS320C6416 processors or DSPs with floating point units are supported by I-SENSE too.

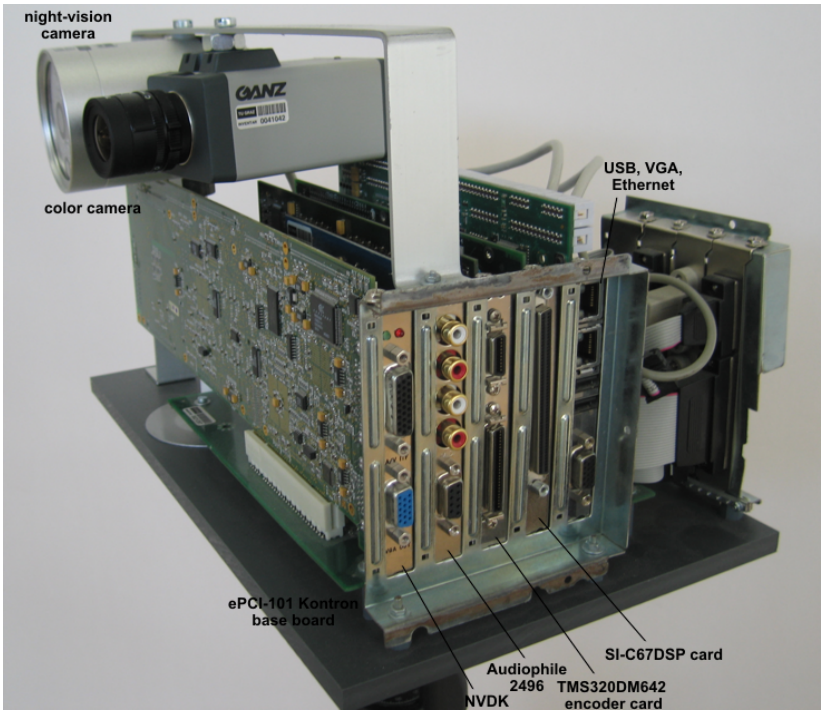


Figure 4.1.: I-SENSE node

The TMS320C64X processors have three 32 bit hardware timers [Tex05]. The timers are either driven from an internal clock source—the CPU clock divided by eight—or from an external signal. A programmable timer period register can cause a reset of the timer count register and the generation of a hardware interrupt when the programmed counter state has been reached. The TMS320C64X timers have no hardware facility to capture the timer count upon an event. One of the timers is used by the DSP/BIOS OS to drive the system clock and scheduling [Tex04c]. In the default configuration it generates 1000 clock tick interrupts per second. The other two are free for application use.

4.1.2. Software architecture

The current I-SENSE middleware runs atop the Microsoft Windows XP embedded operating system on the general purpose processors,[4] and on DSP/BIOS on the TMS320C64X processors.

DSP/BIOS is a scalable real-time multi-tasking kernel, designed specifically for the TMS320C6000, TMS320C5000, and TMS320C28x DSP platforms [Tex00]. DSP/BIOS provides standardized APIs across the supported platforms [Tex04b]. It is designed as a collection of configurable modules to minimize memory footprint. DSP/BIOS development for I-SENSE has been done with the Code Composer Studio IDE, which includes a graphical tool for statical configuration of kernel modules and objects. Where required, dynamical configuration using operating system calls has been used too.

[4] Actually other Windows variants like Windows XP or Windows 2000 work too, but their resource usage cannot be trimmed down to the extent that is possible with the modular Windows XP embedded system.

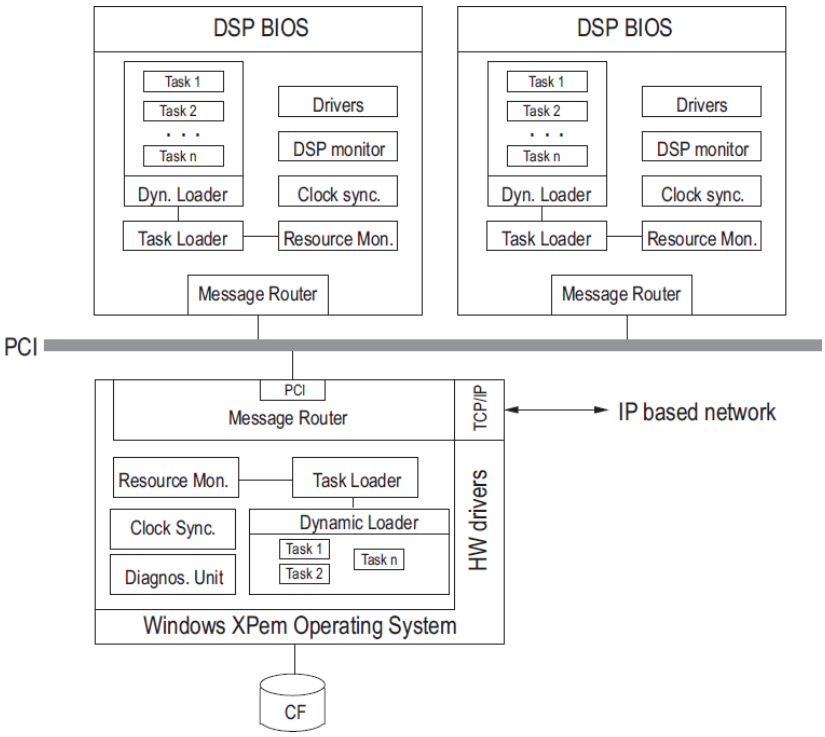


Figure 4.2.: I-SENSE middleware services [TKR07]

Processor model dependent hardware details—especially the on-chip peripherals—are supported through the C6000 Chip Support Library (CSL). It is a software layer below DSP/BIOS and also available to applications [Tex04a].

The major I-SENSE middleware services are shown in fig. 4.2. This chapter is mainly concerned with the implementation of Clock Synchronization Service on both operating systems.

4.1.3. The I-SENSE message subsystem

The design of any time synchronization solution and the achievable performance level depend on the properties of the used clocks and communication channels. The (time related) properties of the I-SENSE Message Router Service are therefore important.

The I-SENSE message subsystem provides the following features [Ten08] :

Physical network independence I-SENSE fusion tasks communicate with each other in a uniform way, regardless of the underlying media (PCI bus, Ethernet, . . .)

Transport layer independence The I-SENSE message subsystem provides an overlay transport layer. Fusion tasks communication is independent of the underlying operating system and transport facility.

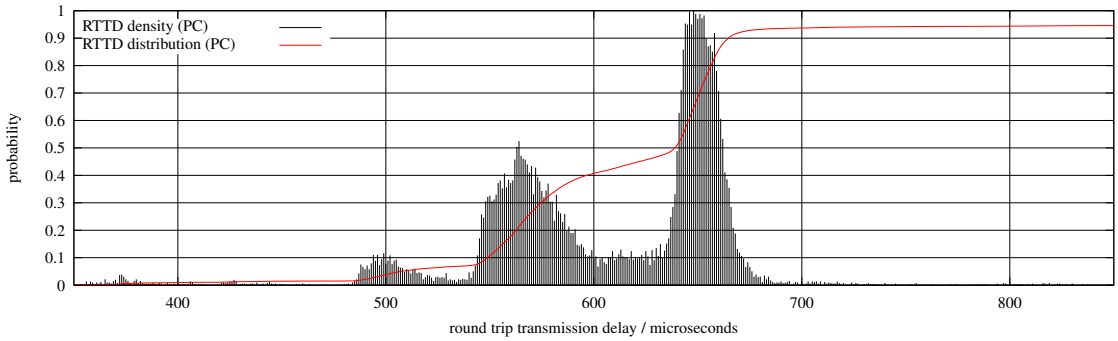
Connection independence Underlying transport connections are dynamic. Connection changes at runtime are transparent to fusion tasks.

Location independence Fusion tasks may be migrated at runtime.[5] After migration the framework updates the communication links of the task with new connections, notifies linked nodes about the tasks new location, and installs a temporary message forwarding proxy at the old location.

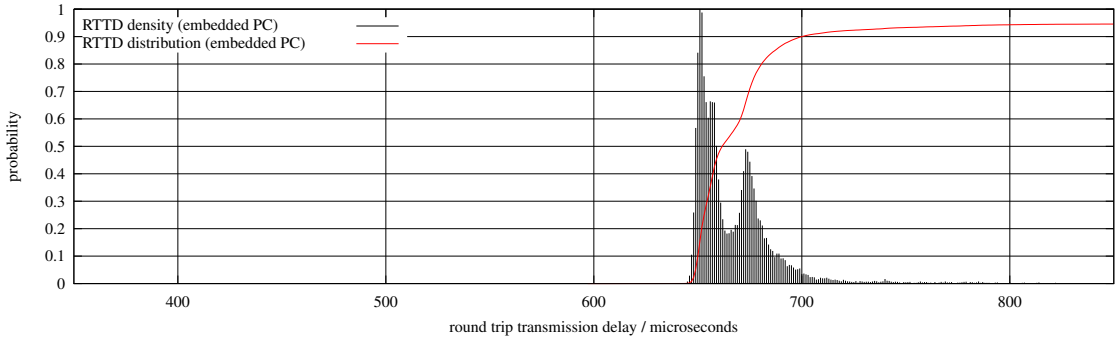
Internally the I-SENSE message subsystem uses FIFO buffers and dedicated TCP/IP and PCI threads. The threads handle unidirectional traffic, i. e., there are always send/receive thread pairs.

The message subsystem is not only a necessary component for time synchronization—it is also a consumer of synchronized time. The message headers contain a timestamp field, which is used internally for message ordering.[6]

Message passing over the PCI bus is implemented with two shared memory ring buffers. To avoid the copy-in and copy-out overhead an alternative method for large messages is implemented too. The indirect method uses the ring buffer to inform the receiver about the message size. The receiver responds with the address of a suitable memory buffer, and the message contents is then efficiently transfered via DMA. The message size threshold for DMA transfer is currently 256 bytes.



(a) Industry PC⇌DSP channel delay distribution



(b) Embedded PC⇌DSP channel delay distribution

Figure 4.3.: DSP→x86→DSP round trip time histograms

[5] Fusion task migration is not transparent, i. e., upon migration notification a task must save its state and suspend processing.

[6] The pre-synchronization workaround mentioned in section 1.1—i. e., resetting all clocks at middleware startup—avoids the chicken-and-egg problem.

fraction	δ_{PC} / µs	δ_{ePC} / µs
lowest value	329	597
bottom 25 %	< 569	< 653
median	639	662
top 25 %	> 653	> 677
top 5 %	> 1060	> 961
top 1 %	> 12675	> 17475
highest value	38678	78391

Table 4.1.: I-SENSE inter-node communication channel delay

When considering the buffering and scheduling inside the I-SENSE message subsystem, a relatively large variation in message delivery delay can be expected. Measurement results from 26450 message exchanges between a DSP and the general purpose CPU of its sensor node are shown in fig. 4.3.[7] Histogram 4.3a shows measurement results from the development PC `fitipc150`, the second histogram 4.3b was taken with a less powerful embedded PC. Identical DSP board models have been used. The DSPs processed the same fusion tasks during the measurements. On both operating systems the message exchanging threads were run with elevated scheduling priorities.[8]

Because the high-delay tails of the histograms are surprisingly long, about 5 % of the samples had to be cut off on the right side. Some numerical data about the delay distributions is given in table 4.1.[9]

4.2. Design Decisions and Their Rationale

Each design process involves choices. This section gives a high level view of the I-SENSE synchronization service design. Instead of giving only a description of the architecture, the design is explained as a consequence of a few high level design decisions. All decisions have several pros and cons, which are discussed only briefly.

4.2.1. Choice of transport layer

Any time synchronization protocol relies on some transport mechanism(s) for exchanging timestamp and protocol information. For an I-SENSE DSP, there is no alternative to the I-SENSE communication subsystem, because no other transport layer exists.[10] In contrast, for inter-node protocol exchange there is the choice between using either the middleware facilities or directly the underlying Windows XPe network stacks.

[7] The histograms are vertically scaled to unit height. Depicted with the same scale, histogram (b) would be taller by a factor of 2.82 —such that the filled areas were almost equal in both histograms.

[8] On Windows the `THREAD_PRIORITY_ABOVE_NORMAL` priority was used. On DSP/BIOS `TSK_MAXPRI` (the top priority for normal tasks) was used [Tex00]. The same priorities are used by the inter-node synchronization implementation.

[9] Note that although the embedded machine needed 268 µs more for the best round trip, there is only 23 µs difference in the median value!

[10] Below the I-SENSE message subsystem there is only low level PCI communication support from device drivers.

It was decided to use the network stack of the operating system to avoid reduced performance. Time synchronization is essentially reduction in time measurement *uncertainty*. Sections 2.5.1 and 3.1.8 explain why timestamps should be taken at the lowest possible layer to avoid unpredictable delays. Internal buffering and context switching by the I-SENSE message subsystem would add to the system noise and degrade timestamp quality. Figure 4.3 confirms that there is significant jitter even on local PCI links.

4.2.2. Choice of implementation layer

Decision 4.2.1 leads to another pair of alternatives. Since inter-node synchronization messages do not use the I-SENSE message subsystem, inter-node synchronization can be implemented below the middleware layer rather than inside it.

Implementation outside the middleware has been chosen, because it allows to leverage existing state-of-the-art solutions.

4.2.3. Inter-node protocol and implementation selection

Many aspects of the I-SENSE software environment on the general purpose processors are similar to typical small server: A Windows variant on hardware that resembles a small COTS PC, Ethernet connectivity, permanent operation, and no energy scarcity. The standard Internet time synchronization protocols NTP and SNTP are known to work well under this circumstances.[11]

NTP has been chosen, because it is widely regarded as one of the most advanced and time-tested protocols [RBM05] and good quality open source implementations are available.

The current stable release of the NTP reference implementation has been selected. Main reasons for this choice were source and binary code availability, extensive documentation, flexible (although complex) configuration possibilities, and valuable logging options.

Resource consumption was deemed acceptable. The binary has a size of 508 KB. The daemon uses—even with an extensive ensemble of configured time servers—less than a megabyte of memory.[12] CPU usage is not noticeable.

4.2.4. Intra-node protocol and implementation

DSPs must synchronize their clocks with their local general purpose processor. The DSP (slave) part of the protocol implementation should have a small memory footprint. Floating point arithmetic operations are undesirable, as they require inefficient emulation by library code. Since no networking hardware is attached to DSPs, compatibility with packet formats of existing protocols is irrelevant.

[11] But NTP—and presumably most other application level protocols too—could still perform better on Windows, if there were operating system support for low level network packet timestamping and a high frequency system clock.

[12] As reported by the Windows task manager.

A custom protocol has been designed, which can be viewed as a modified version of SNTP. It uses the I-SENSE communication subsystem instead of UDP/IP. Enhancements with respect to SNTP are an improved clock filter algorithm, which was necessitated by the difficult I-SENSE channel delay characteristics, and the ability to synchronize to more than one timescale. Among the simplifications are the restriction to client/server mode, omission of the error model calculations, fixed polling intervals, and a less complex clock discipline algorithm.

The implementation is object oriented, uses integer and fixed point arithmetic only, and has low resource demands.

4.3. NTP Configuration

NTP supports five modes of operation and has a lot of tunable parameters. The NTP reference implementation features a multitude of configuration options. It is optimized for a stable temperature environment and some default settings favor frequency stability and low communication bandwidth over accuracy. This section summarizes some of the lessons learned during the practical part of this thesis, and gives configuration recommendations, which are tailored for the I-SENSE platform.

Two avoid a single point of failure, more than one server should be available inside the I-SENSE network. Two or three nodes should together act as the (local) top level of the time distribution hierarchy. They must be configured at the same stratum and synchronize with each other in symmetric active mode.[13] Lightly loaded machines are commonly recommended, because high system load is known to degrade time server performance.[14] This hint is probably less useful for I-SENSE, because the middleware distributes fusion task load dynamically. The other I-SENSE nodes are synchronized to the local server ensemble in client mode, i. e., they are in the next stratum.

If external synchronization is desired, UTC sources must be available to the local servers. If there is Internet connectivity, thousands of public NTP servers are available. Otherwise commercial external reference clocks can be attached to the server nodes[15] or embedded LAN products can be deployed. At least four or five time sources are generally recommended, to benefit from the NTP selection and clustering algorithms. Without UTC sources, or when all such sources fail, there is only internal synchronization.

The NTP default poll intervals and the hysteresis built into their dynamic adjustment algorithm limit LAN performance. The resulting long clock discipline time constants (cf. section 3.3.4) delay the response to time offsets too much. When minimization of inter-clock time offsets is more important than local clock frequency stability, then it is best to clamp the poll interval exponent τ to the minimum.[16]

[13] Symmetric passive mode allows time injection from unknown sources and is therefore a security risk, except on very controlled network environments or when used with cryptographic authentication.

[14] NTP itself generates very little load. Fast COTS PCs can handle thousands of NTP clients.

[15] The NTP reference implementation incorporates some 30 reference clock drivers.

[16] The minimum τ (without recompilation) was 4 (16 s) in older stable versions and is at the time of this writing 3 (8 s).

4.4. Custom Intra-node Synchronization Protocol

4.4.1. Implementation classes overview

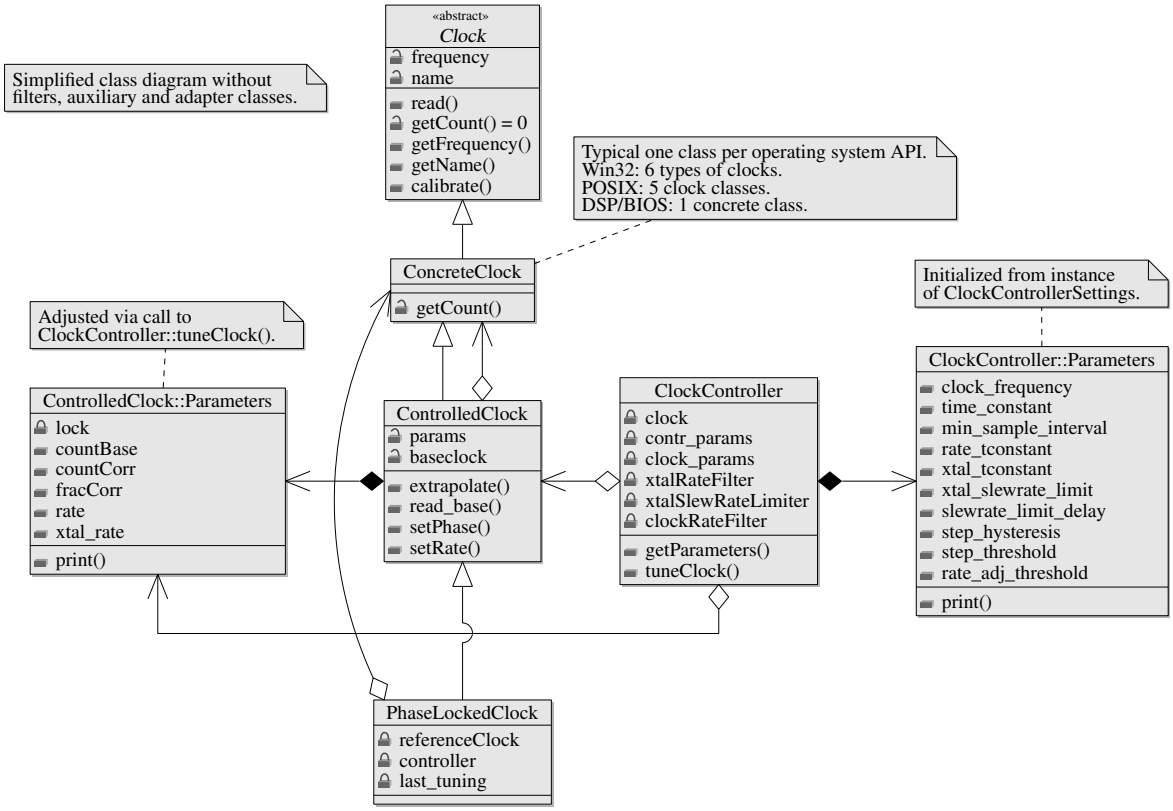


Figure 4.4.: Additional I-SENSE framework classes

Figure 4.4 shows a condensed UML class diagram of the implementation. Almost all C++ code is shared between the Windows and DSP/BIOS implementations. Operating system dependencies have been factored out into specific implementations of abstract base classes.

Basic clocks

Section 2.4 pointed out that there is an irritating number of time formats and clock APIs—even within a single operating system.[17] Selection of the best available clock is hard, because clock performance and reliability depend on the actual hardware, firmware, and system software combination. The abstract `Clock` class shields the implementation from the idiosyncrasies of the underlying hardware and system software. It represents the basic concept, that a clock is a combination of a fixed frequency oscillator with a counter (cf. fig. 2.1). The implementation uses the *Template Method* design pattern.

[17] Confer table 2.5 on page 23!

Most *ConcreteClock* classes are extremely terse. They implement the abstract `getCount` method—typically by calling the underlying clock API and returning a 64 bit value. On DSP/BIOS one of three 32 bit hardware timers is encapsulated by the `DSPClock` class. On Windows there are more concrete clock classes:

SysClock The windows system time (`GetSystemTimeAsFileTime`) is queried. Windows updates the system time only 64 or 100 times per second.

GT1Clock This clock uses a multimedia timer API (`timeGetTime`) to get the system time with a precision of 1 ms.

TCClock This clock has a resolution of 1 ms (`GetTickCount`) but it tracks the time since the system was booted instead of UTC.

PCClock The performance counter is queried. This is the highest resolution clock available on Windows and recommended for short interval measurements. Unfortunately, the performance counter behaves erratic on some systems.

MMClock The multimedia timer is the primary time source. The performance counter is used to interpolate between the multimedia timer callback events. The same scheme is used by the Windows port of the NTP reference implementation.

Encapsulation of clock APIs offers much flexibility. Switching to an alternative concrete clock is simply a matter of changing one constructor call.[18]

Adjustable software clocks

The concept of an adjustable clock is captured by the `ControlledClock` class. Since most of the concrete basic clocks are read only, the implementation of adjustability is done in software. As a consequence, a controlled clock is not only a specialization of a concrete clock, it also holds a reference to an underlying concrete clock instance. This immutable *baseclock* member provides the time information and the controlled clock instance only applies rate and offset corrections when it is read.

Piecewise linear extrapolation relative to a settable reference time (T_0, T_0') with a rate adjustment r_c is used.[19]

$$T_A'(T_B) = T_0' + (1 + r_c)(T_B - T_0) \tag{4.1}$$

The rate correction r_c is a signed 32 bit value, which represents parts per 2^{32}. This allows rate corrections of $\pm 50\,\%$ with a granularity of 0.233 ppb.[20] Sawtooth errors (cf. sec 2.4.2 on page 22) are hereby avoided.

[18] Run-time switching between clock implementations (as described in [Kam02]) could be implemented too. To be useful, code for automatic clock sanity checks and performance evaluation would be needed though.

[19] The primes ($'$) in equation 4.1 mark timestamps on the timescale of the adjusted clock.

[20] Greater frequency ratios are handled by the `PrescaleClock` adapter class.

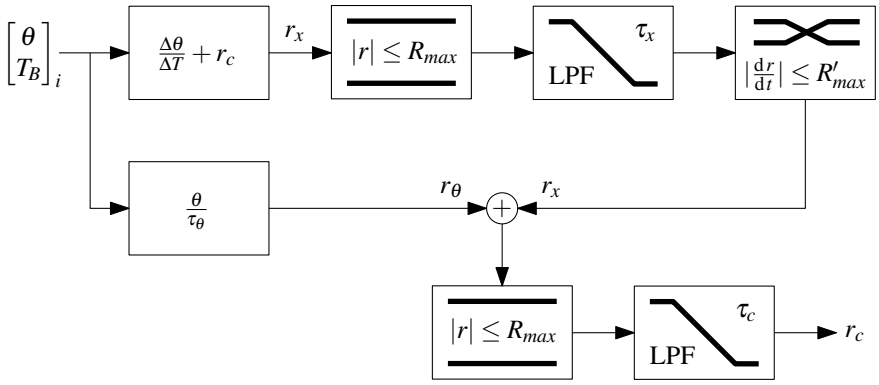


Figure 4.5.: Clock controller implementation

Filters

Only a few basic filter types are used to filter $(timestamp, value)$ tuples. They are subclasses of the `TimedFilter` class. The classes are generic, i. e., arbitrary arithmetic data types may be used for the timestamp and value type, respectively.[21] The `MinMaxFilter` and `MedianFilter` return the minimum/maximum/median of the last N tuples, where N is a parameter to the constructor. The `SlewRateLimitFilter` limits *value* change rates. The `ExpAvgFilter` filter type is a first order low pass filter. The filter output a_i is

$$a_i = \begin{cases} v_0 & \text{if } i = 0, \\ \dfrac{\tau_{avg}\, a_{i-1} + (t_i - t_{i-1})\, v_i}{\tau_{avg} + t_i - t_{i-1}} & \text{if } i \ge 1, \end{cases} \tag{4.2a}$$

and for faster start-up behavior the exponential averaging interval τ_{avg}

$$\tau_{avg} = \min(\tau_K, t_i - t_0) \tag{4.2b}$$

is used instead of simply the constant filter parameter τ_K.

Clock controller

If a `ControlledClock` instance is to be synchronized with another clock, it must be part of a control loop. Output from the clock filter (cf. fig. 4.7) is fed in the form of (T_B, θ) tuples (baseclock timestamp and offset sample) into the clock controller. Numerous approaches for clock offset and rate estimation with varying implementation complexity exist; section 2.5.2 lists some of them. The `ClockController` class currently uses a low-complexity approach. The computations behind the `tuneClock` method are depicted in fig. 4.5.

The clock controller assures a minimum time interval between any two samples used for tuning the controlled clock. If we use equation 2.13b for clock prediction, then the accumulated difference in offset error $\Delta\theta$ since the last sample ΔT ago, must be due to the clock skew difference ΔS. If we add

[21] They are implemented as C++ templates with type parameters `TimeT` and `ValueT`.

the previous rate correction r_c, i. e., the rate adjustment that has been applied to the baseclock during the last interval (cf. eq. 4.1), then we obtain the rate difference r_x between the sampled reference clock and the baseclock.

$$r_x = \frac{\theta_i - \theta_{i-1}}{T'_i - T'_{i-1}} + r_c \tag{4.3}$$

Inevitable measurement noise introduces an error that is inversely proportional to $\Delta T'$. The estimated rate difference r_x is therefore limited, exponentially averaged, and change-rate limited in turn.[22] Applying the resulting r_x rate adjustment to the baseclock, would only correct the rate difference between the controlled clock and the reference clock. To reduce the offset an additional term

$$r_\theta = \frac{\theta_i}{\tau_\theta} \tag{4.4}$$

is added.

The r_x term models the clock skew difference ΔS between the reference clock and the baseclock. It is therefore averaged with a relatively large time constant τ_x. The actual r_x value is correlated with the local crystal oscillator temperature and can be retrieved from the controller object. As a future extension, r_x can be modified according to local temperature sensor data in order to improve controller responsiveness to temperature changes.

Large offsets are reduced by a superimposed two-point controller, which applies a constant rate of $\pm R_{max}$ (± 500 ppm) while $|\theta| > \theta_{thresh}$. Like in NTP, huge offsets are corrected by stepping the controlled clock, because continuous amortization would last too long.

4.4.2. Timestamp format and timescales

For any given timestamp size there is a tradeoff between resolution and maximum interval length. There are many examples of timestamp formats that have been defined too small by "historical accident".[23] The I-SENSE time synchronization service therefore uses and provides 64 bit timestamps. An integer format with a resolution of 1 µs has been chosen, because that is computationally efficient (compared with the split-resolution structure types of table 2.5) and human-friendly as well. Since 2^{64} µs is about 585×10^3 years, there are no wrap-around issues to be expected in the foreseeable future. Applications that are concerned with short intervals only, are free to use the least significant 32 bits instead, which gives a wrap around interval of 1 h11 m35 s.

The preliminary synchronization service placeholder had used a 32 bit format with a 10 µs resolution that wrapped around in less then 12 hours. That resolution would have limited achievable performance, since accuracies of a few tens of microseconds are possible for COTS PCs connected over a fast LAN under ideal circumstances [Mil06b, p. 10] [RV09].

Applications have different timescale requirements. Unfortunately, the timescale properties of accuracy, stability, and (strict) monotonicity require implementation tradeoffs. An experimental second timescale (besides UTC) has therefore been added to the intra-node synchronization protocol. This *interval* or *scheduling* timescale guarantees strict monotonicity and offers better frequency stability over short

[22] The slew rate limiter is activated after a delay, to allow the low pass filter output to settle first.

[23] For example the Y2K problem or the year 2038 problem caused by 32 bit POSIX `time_t` types.

and medium time periods than the UTC realization. The timescale uses the Windows performance counter as its source.

Due to the object oriented design, the addition of a second timescale was very easy. Basically an additional timestamp had been added to the protocol data unit and another clock object with its associated controller had been instantiated. Since those objects keep only very little internal state, the impact on memory use is minimal.

Alternatively, the second clock can be connected to the same timescale. With this setup different clock filters and/or controllers—either implementations or parameter settings—can be evaluated and compared concurrently in real time.

4.4.3. The custom algorithm

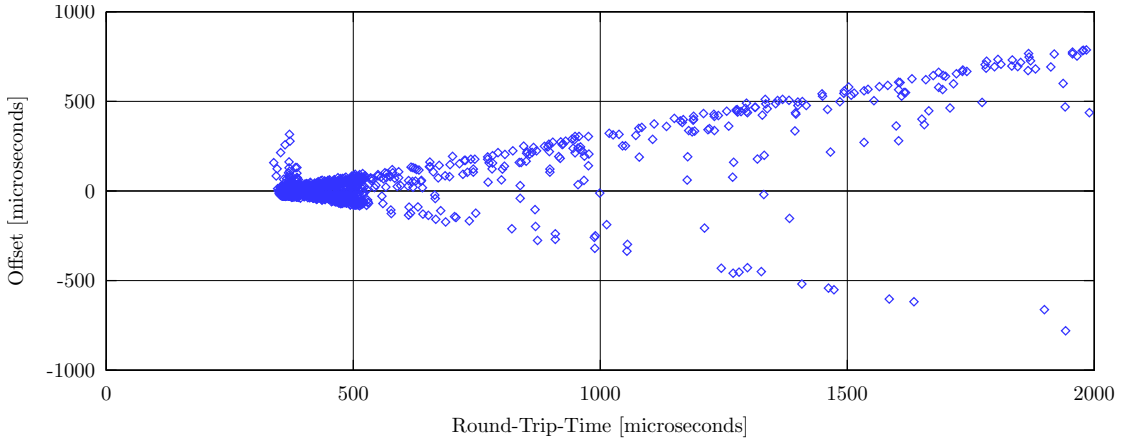


Figure 4.6.: Offset versus round-trip time scatter diagram.

The modified SNTP algorithm has a structure similar to fig. 3.7. Two-way time measurement implements the phase detector. The measurement error of the phase difference samples θ_i depends on the transmission delay δ and is bounded by equation 2.22. By using the NTP clock filter, samples from the left (low error) part of the scatter wedge in fig. 4.6 are selected. The scatter diagram was taken on `fitipc150` and corresponds with the histogram 4.3a and the middle column of table 4.1, respectively.

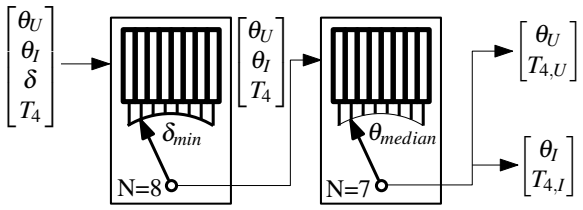


Figure 4.7.: Clock filter implementation

protocol field	clock	size / bytes
origin timestamp T_1	H	8
receive timestamp $T_{2,I}$	I	8
receive timestamp $T_{2,U}$	U	8
Processing delay $d = T_{3,I} - T_{2,I}$	I	4
sequence number	-	4
Protocol version	-	8
sum		40

Table 4.2.: Inter-node synchronization message format

Analysis of a few 10^5 samples showed, that there was only little incremental improvement for $N > 5$ filter stages. Nevertheless $N = 8$ stages (like in NTP) are used. Plots of the clock filter output revealed that it contained a small percentage of outliers, sometimes even adjacent ones.[24] These outliers are removed by the seven stage median filter shown in fig. 4.7. The median filter output is not chronological ordered. Reordered samples are ignored by the clock controllers.

A `ClockController` instance corresponds to the loop filter block of fig. 3.7. It adjusts the rate of its controlled clock object. The clock object corresponds to the VCO.

The protocol data unit is shown in table 4.2. If the software is compiled for debugging and evaluation,[25] internal state variables and PPS timestamps are also present in the PDU and its size increases to 156 bytes. The T_1 timestamp would be only required in symmetric mode and is currently not used by the (x86) server side of the protocol. The three involved clocks are the DSP hardware clock (H), the UTC clock (U), and the interval clock (I). The hardware clock is the base clock for the other clocks on the DSP. On the general purpose processor, the UTC clock is the NTP-controlled Windows system clock and the interval clock is derived from the performance counter.

The client and servers synchronization loops are shown in fig. 4.8 and fig. 4.9, respectively. The offset and delay calculations are equivalent to the NTP timestamp exchange protocol (cf. fig. 3.4 and equations 3.3 and 3.4). They appear different because two minor optimizations have been made:

1. Instead of transmitting both timestamps $T_{3,I}$ and $T_{3,U}$ separately the processing delay of the server $d = T_{3,I} - T_{2,I} = T_{3,U} - T_{2,U}$ is replied.
2. On the DSP all three clocks are read synchronously by first reading the hardware baseclock, and then passing the obtained value to the `extrapolate` method of the controlled clock instances.

[24] The NTP reference implementation contains a so called *popcorn spike filter* [Mil06b, p. 72]. This filter uses a dynamic suppression threshold, which is $3\varphi_P$. It cannot suppress adjacent outliers reliably.

[25] With the `DEBUG_CLOCKSYNC` preprocessor symbol defined to a non-zero value.

```
initialization ;
while true do
    sleep (poll-interval) ;
    prepare synchronization message ;
    T_{1,H} := hardware-clock.read () ;
    send_to_x86 () ;
    (T_{2,I}, T_{2,U}, d) = receive_from_x86() ;
    T_{4,H} := hardware-clock.read () ;
    T_{4,I} := interval-clock.extrapolate (T_{4,H}) ;
    T_{4,U} := UTC-clock.extrapolate (T_{4,H}) ;
    rtt := T_{4,H} − T_{1,H} ;                                  // round trip time
    δ := rtt − d ;                                              // sum of both channel delays
    rcv-to-rcv := (rtt + d)/2 ;                                 // interval between x86 and DSP recv
    θ_I := T_{2,I} + rcv-to-rcv − T_{4,I} ;
    θ_U := T_{2,U} + rcv-to-rcv − T_{4,U} ;
    if first message then
        interval-clock.add_offset (θ_I) ;
        UTC-clock.add_offset (θ_U) ;
        θ_I := 0 ;
        θ_U := 0 ;
    endif
    (T_{4,H,min}, θ_{I,min}, θ_{U,min}) := minimum-δ-filter.process (δ, T_{4,H}, θ_I, θ_U) ;
    if new δ minimum then
        pass (T_{4,H}, θ_I, θ_U) to median-θ-filter;
        pass filter output to clock controllers ;
    endif
endw
```

Figure 4.8.: Client (TMS320C64X) synchronization loop

```
initialization ;
while true do
    receive_from_DSP () ;
    T_{2,I} := interval-clock.read () ;
    T_{2,U} := UTC-clock.read () ;
    prepare reply message ;
    d := interval-clock.read() − T_{2,I} ;                      // processing delay of server
    send_to_DSP (T_{2,I}, T_{2,U}, d) ;
    if compiled for packet logging then
        store packet data in ring buffer ;
        if ring buffer more than half full then asynchronously notify logger thread ;
    endif
endw
```

Figure 4.9.: Server (Pentium M) synchronization loop

5. Evaluation

The protocols have been evaluated first separately and then together. NTP performance is evaluated in section 5.1 and the custom protocol in section 5.2. An end-to-end evaluation, which compares the clocks of signal processors in separate I-SENSE nodes, is presented in section 5.3.

Because of its large time constants, NTP converges only slowly. NTP startup behavior does not affect the inter-node and end-to-end results, because the displayed data has been recorded after several hours of continuous operation.[1] In contrast to NTP, the intra-node protocol converges quickly. Its startup transients are displayed in the section 5.2 diagrams (i. e., figures 5.4, 5.5, and 5.6).

5.1. Inter-node Evaluation

Evaluation of inter-node synchronization was complicated by the fact, that external precision time and time-interval measurement equipment[2] was not available. The lack for low level timestamping support in Windows was another drawback.

Although the NTP reference implementation is widely used, no suitable scientific publications about its performance over a LAN between Windows machines could be located. All assessed NTP related publications are either in the context of Unix/Linux operating systems or look at the Microsoft implementation. A paper comparing the NTP performance of the NTP reference implementation on Linux with the Microsoft implementation concludes [SA06]

- that the synchronization accuracy on Linux is limited by the clock discipline time constant, and
- that the MS Windows implementation is severely restricted by the coarse granularity of the Windows system time clock.

In June 2009 experimental support for the PPS API [MMB+00] was added to the Windows port of the `ntp.org` development version. The results of two experiments, which took advantage of the PPS timestamping facility, are shown in figures 5.2 and 5.3.

In both experiments the PPS output of a GPS time receiver[3] was connected to the Data Carrier Detect (DCD) pin on a serial-line interface of both PCs. The triggered interrupts are timestamped by a patched Windows serial line driver. Alternative hardware had to be used, because in the ITI VLSI laboratory unobstructed sky view, which is required for GPS signal reception, was not available within antenna cable range. To match the capabilities of embedded hardware, two older COTS PC models were used and interconnected with a 100 Mbit/s Ethernet. The machines `mark` and `space` are equipped

[1] It is assumed that (re)starts are rare events, since sensor networks are typically operated continuously.
[2] Like an external reference clock and one of the many commercial PCI time and frequency measurement products.
[3] The used Motorola Oncore UT+ GPS receiver provides output pulses that are ± 50 ns ($1\,\sigma$) accurate to UTC.

with Intel Celeron processors with clock frequencies of only 800 MHz and 900 MHz, respectively. They were operated without ambient temperature control in a domestic environment and only lightly loaded during the experiments. The general purpose processors of current I-SENSE nodes are Intel Pentium-M models with clock frequency options up to 1.6 GHz—limited only by passive cooling.

In the first experiment both machines were running the latest NTP reference implementation on Windows XP. Machine `mark` was synchronized to the GPS receiver and served its time via NTP to `space`. On the client the PPS signal was used for evaluation only, not for synchronization.

The black and blue curves of fig. 5.2 display clock offset from UTC for the server and client, respectively. They have have been measured with low level PPS timestamps. The roughly 0.1 ms (peak to peak) jitter that is visible on both curves comes from the Windows kernel. In order to work around the low resolution of the Windows system time clock the NTP reference implementation uses an interpolated clock like the `MMClock` described on page 55. The Windows kernel delivers the timer event callbacks with limited precision and this causes the jitter. Operating systems with a high resolution system clock (e. g., Linux and many Unix variants) do not have this limitation.[4]

The red curve labeled "Client - Server" fig. 5.2 displays the filtered offset samples that have been made by NTP, i. e., the output of the clock filter algorithm. This is the client's notion of its clock offset relative to the server. Because of the clock filter delay some samples are stale, i. e., they are used several minutes after their measurement.

The experiment revealed a bug in the Windows system clock implementation on client *space*. During the six hour experiment it occurred 10 times that a millisecond was lost—presumably due to lost interrupts. This problem makes fig. 5.2 quite instructive, because the response to a 1 ms offset step is clearly visible. Until 06:35 the protocol was run with default settings. Then the client daemon was reconfigured with a fixed poll interval of 8 s and restarted. The reduced clock discipline time constant caused a much quicker response to offset errors. Also visible before 06:35 is the effect of the dynamic poll interval adjustment algorithm, i. e., the sampling interval varies (cf. page 46).

For the second experiment FreeBSD 7.2 was installed on `space`, which resolved the problem with lost milliseconds. This time `space` was synchronized to UTC with the GPS receiver and provided its system time via NTP across the LAN to client `mark`. Again, the client poll interval was clamped to 8 s. Results are shown in fig. 5.3.[5] The good timekeeping support of the FreeBSD kernel is reflected by less than 3 µs RMS short time jitter of the black "Server - UTC" curve. Again, most of the roughly 100 µs peak-to-peak jitter of the blue "Client - UTC" curve is presumably caused by application level clock interpolation (cf. section 2.4.1 on page 21), since interrupt latencies are considerable smaller in magnitude and PPS signal jitter was less than 1 µs. The even higher jitter of the red "Client - Server" NTP clock filter output—i. e., the input to the client side NTP sub-algorithms—results from accumulated timing uncertainties during the NTP timestamp exchanges (cf. table 2.7). The fairly constant ≈ 50 µs *average* offset between client and server shows that the path-reciprocity assumption behind equation 3.4 was only approximately correct. On average, the delay of the request path was approximately 100 µs larger than the delay of the response path (cf. eq. 2.22).

[4] The magnitude of the jitter is small compared to the 5 ms accuracy required by I-SENSE. However under very heavy system load the Windows kernel may delay the delivery of timer event callbacks by several milliseconds. This may cause transient clock reading errors of corresponding magnitude.

[5] Note the change in y-axis (time offset) scale!

During the shown six hours, both absolute time offsets from UTC and relative time offsets between client and server have been below half a millisecond by a wide margin. With a Windows server (instead of Unix) the ≈ 0.1 ms clock interpolation jitter of the server clock would increase offset errors slightly.

Because NTP time constants are large, it can take a few hours from service start-up until the clock discipline settles. During this period offsets are generally larger.

A series of similar experiments[6] showed that maximum absolute offset is positively correlated

1. with the speed of temperature change, and
2. with the length of the clock discipline time constant.[7]

Configured with shortened time constants, the maximum absolute offsets between clients and a server stayed below 2.5 ms during two weeks of experimentation. Desktop PCs in a domestic network had been used. The experiments took place in Winter. Ambient temperature drops from about 25 °C to 10 °C had been induced by extensive airing of heated rooms. An old 100 MHz PC with only a 10 Mbit/s NIC showed the largest offsets.

5.2. Intra-node Evaluation

The setup for evaluation of the custom intra-node protocol needed facilities for timestamping of external signals on Windows and DSP/BIOS. Both had to be created first.

No suitable Windows parallel or serial port device driver with low level timestamping support could be located. A RFC 2783 compatible PPS API [MMB+00] was therefore implemented.[8] This implementation does not require kernel level support. A user level thread with a time critical scheduling priority is notified about selected serial line status changes via the Windows `WaitCommEvent` function and creates timestamps. Compared with a kernel level implementation, there is some loss of precision because of scheduling latency. In practice the facility performs quite well. Few outliers occur and it is easy to identify them on a plot. About 20 µs jitter have been removed from the plots by computing a sliding window average of 11 samples (5.5 s).

On the DSPs interrupt handlers had been added, which fetched a timestamp from the hardware clock and stored it for later retrieval with the `getLastTimingPulseEvent` function. The hardware interface consisted of an optocoupler connected to an open-collector interrupt line on an external extension board connector.

A 2 Hz rectangular signal (2PPS) interrupted all CPUs synchronously. The signal was created by dividing the 32.768 kHz output of a simple crystal oscillator with a 14-stage binary counter. Because the *common-view* technique is utilized, the timing characteristics of the 2PPS signal are irrelevant (cf. section 2.5.1). A schematic diagram of the used external hardware is shown in fig. 5.1.[9]

[6] This experiments had a similar setup, but used only Unix/Linux machines and recorded temperature data from on-board and CPU sensors too.

[7] The time constant cannot be made arbitrarily short, as this would cause an instable and oscillating control loop.

[8] The implementation used in section 5.1 did not exist at the time of the evaluation.

[9] The remote DSP board was not connected for the intra-node evaluation. The voltage level for logical zero on the DCD pin

Parameter	UTC clock	Interval clock
τ_x	15 min	30 min
τ_θ	45 sec	2 min
τ_c	15 sec	30 sec
R_{max}	500 ppm	500 ppm
R'_{max}	10 $\frac{\text{ppm}}{\text{min}}$	1 $\frac{\text{ppm}}{\text{min}}$

Table 5.1.: Clock controller parameters

Machine `fitipc150` was used for the experiment. Results of the first hour are shown in fig. 5.4. A sample interval of 457 ms and the clock controller parameters of table 5.1 have been used (cf. fig. 4.5). The parameters had been obtained empirically.

The black curve of fig. 5.4 shows the true offset error as measured with the common external signal. The DSP clock was between 0.1 ms and 0.3 ms early compared to the Pentium clock.[10] The red curve shows the same offset as measured without external hardware, i. e., by two-way timestamp exchanges. It is the offset that is "seen" by the protocol and that the controller tries to reduce.

The shift of $\approx 170\,\mu$s between the red two-way (protocol) and the black 2PPS (hardware assisted) measurements reveals non-reciprocity of paths. The DSP clocks are ahead of the x86 clocks, because the reply path to the DSPs is faster. Probably the *relatively* large asymmetry is due to the fact that there was much more traffic on the DSP-to-x86 leg because of video streaming.[11]

The blue curve of fig. 5.4 shows two-way measurement results of the interval clock offset between DSP and Pentium M. The red UTC clock offset curve is less smooth than the blue interval clock offset curve, because of the following reasons:

- The source clock on Windows has a more complicated implementation with callbacks and interpolation,
- the source clock is adjusted once per second by NTP, and
- the UTC clock controller on the DSP uses shorter time constants.

Internal clock controller state (cf. fig. 4.5) during the experiment is plotted in fig. 5.5. Both rates—the correction r_c applied to the baseclock and the controllers notion r_x of the baseclock skew—are displayed for both clocks. The interval clock needed only a small ≈ 1 ppm swing for a few minutes before it settled to a very smooth rate. The UTC clock controller keeps correcting with about ± 2 ppm rate adjustments.

Time offset between two DSPs is shown in fig. 5.6. The curve is very precise, since interrupt latency variation is the largest remaining cause of measurement error. There is very good common mode noise rejection, as both DSPs use the same software over the same physical communication channel (PCI bus) to query the same time server.[12] After the controllers have settled, offset stayed below

does not conform to the RS-232 standard, which mandates -3 V to -15 V. In practice this did not cause any problem.

[10] The few isolated spikes are measurement errors due to scheduling latency as mentioned above.

[11] This question cannot be decided with the available data, because fusion tasks were running on *all* DSPs in *all* experiments.

[12] However each DSP communicates independently with the Pentium M processor.

approximately 20 µs. Relative synchronization between DSPs within the same I-SENSE node is excellent!

5.3. End-to-end Evaluation

The end-to-end experiment used a similar hardware setup as the intra-node experiment. A second optocoupler was used to interrupt the DSPs on an embedded I-SENSE node.[13] The infrared-emitting diodes of the optocouplers had been connected in series, to ensure that all DSPs were interrupted synchronously (cf. fig. 5.1). Because no Windows XP embedded license was available at the time of the tests, Windows 2000 had been installed on the embedded I-SENSE node as a workaround. Also an *earlier version* of the clock controller without the output low pass filter (cf. fig. 4.5) was used.[14]

A six hours window of the results is shown in fig. 5.7. The data has been sub-sampled with a 2 min interval.

Maximum offset was ≈3.15 ms. Although this *intermediate* result is a bit worse than what could be expected by extrapolation of the separate NTP and intra-node experimental results,[15] it fulfills the goals of the project as set in section 1.2.

[13] Such an additional wired infrastructure could be used for synchronization too. An example of such an approach is described in [NPON02].

[14] The experiment could not be repeated with the final design, because no embedded I-SENSE node was available then.

[15] Adding the inter-node and two times the intra-node offset errors suggests that end-to-end offsets below 1 ms should be possible. Intra-node offsets with the same sign (i. e., both positive or both negative) would even partially cancel each other.

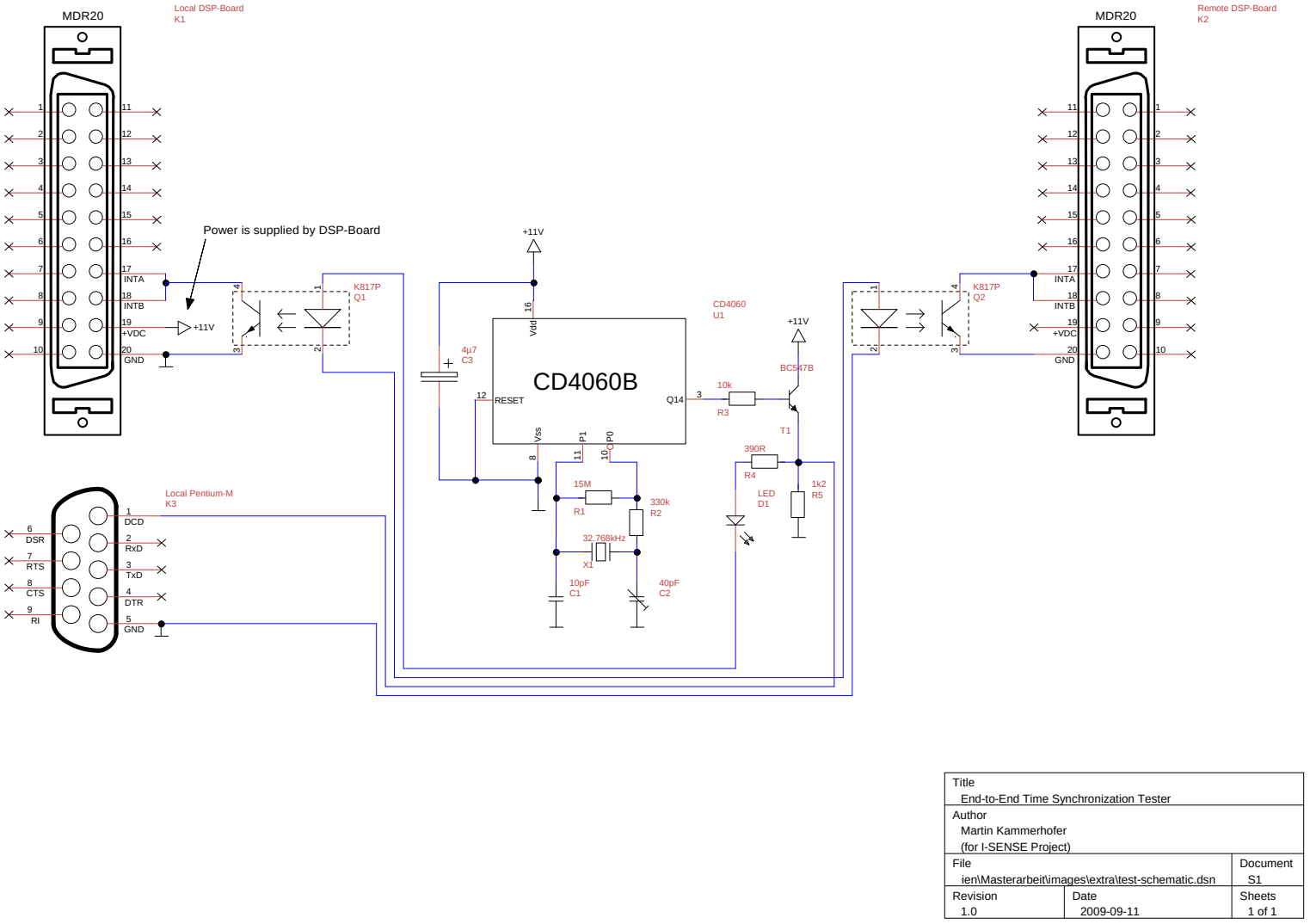


Figure 5.1.: Schematic circuit diagram of external test hardware

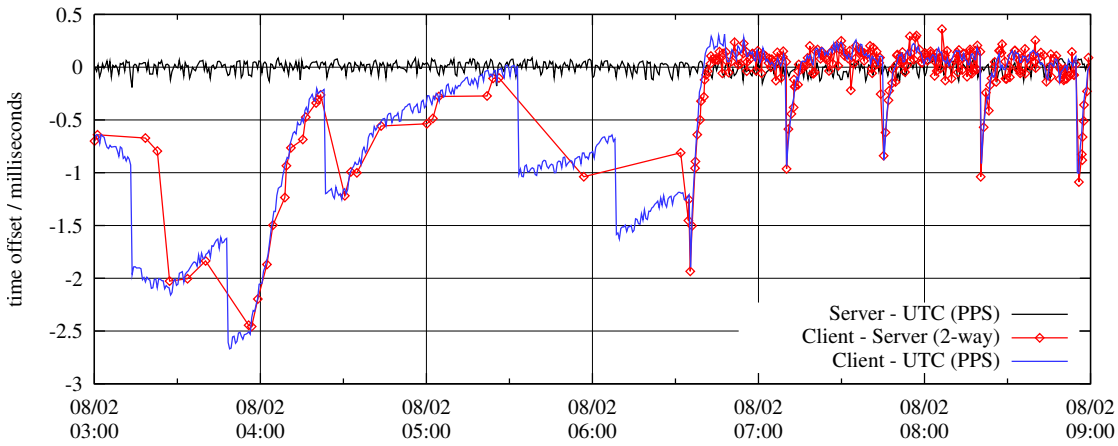


Figure 5.2.: Client clock with spurious 1 ms steps

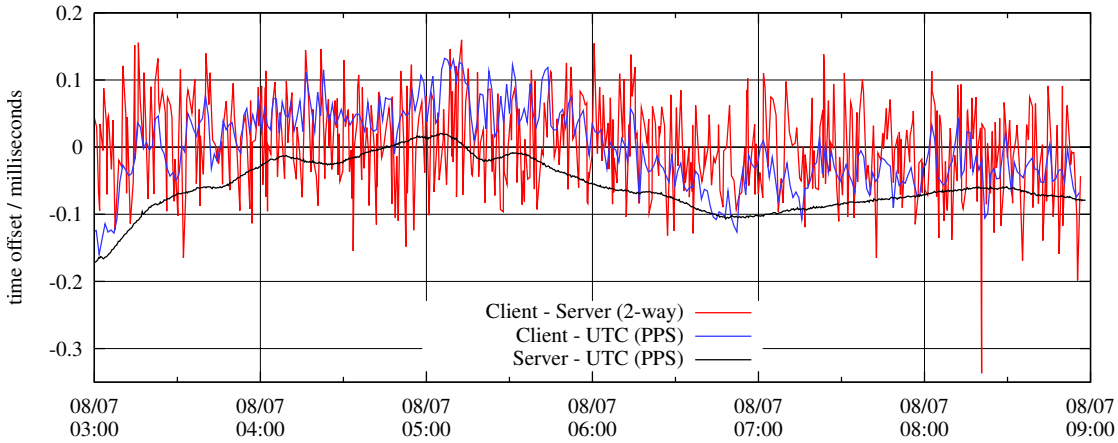


Figure 5.3.: NTP with stratum 1 server on same LAN

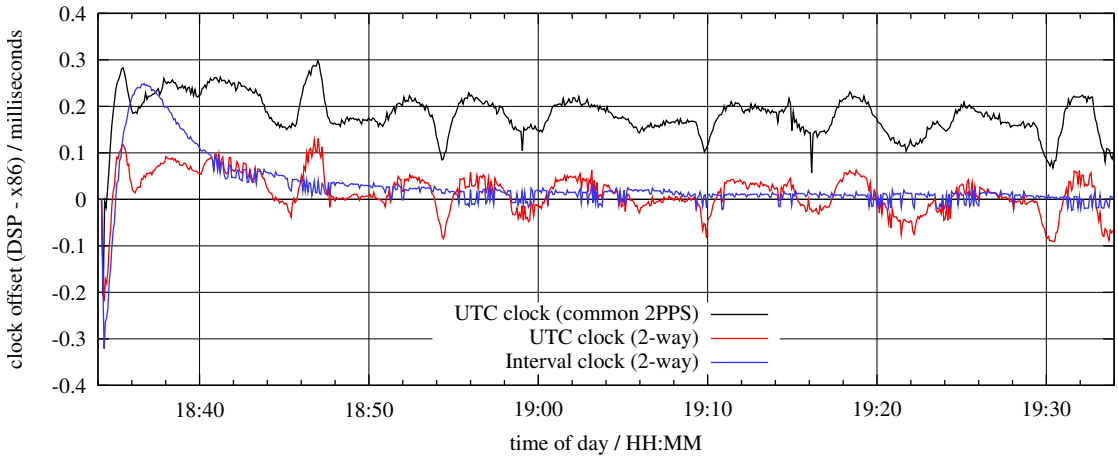


Figure 5.4.: Clock differences between x86 CPU and DSP

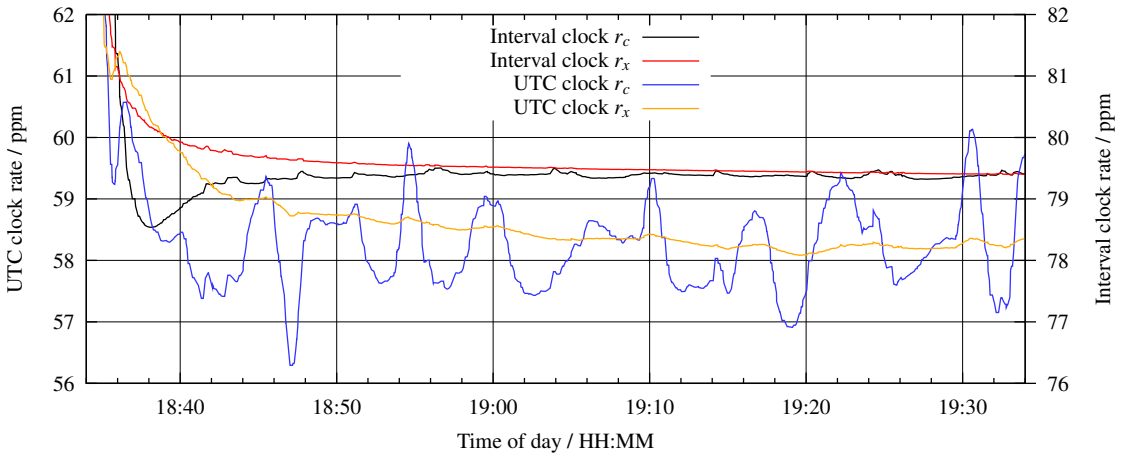


Figure 5.5.: Clock rates

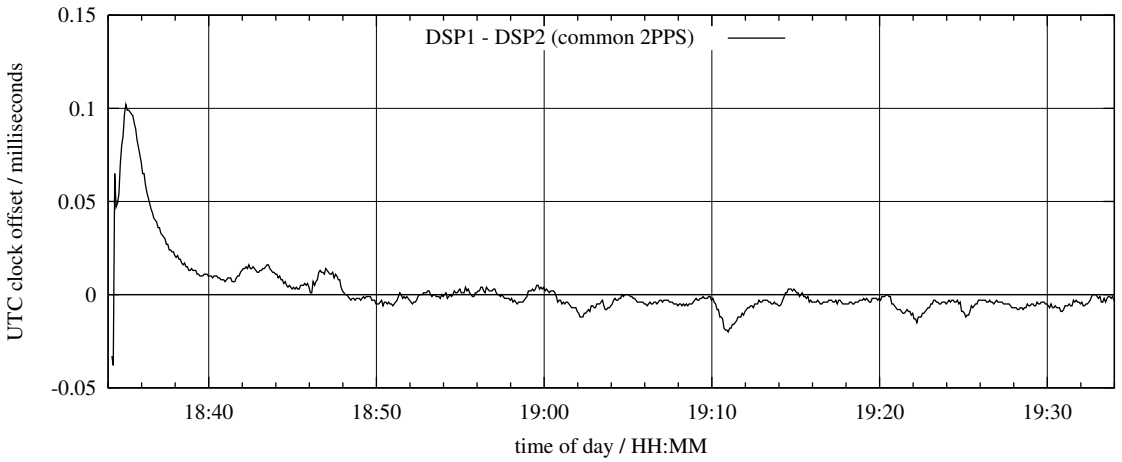


Figure 5.6.: Clock difference between two local DSPs

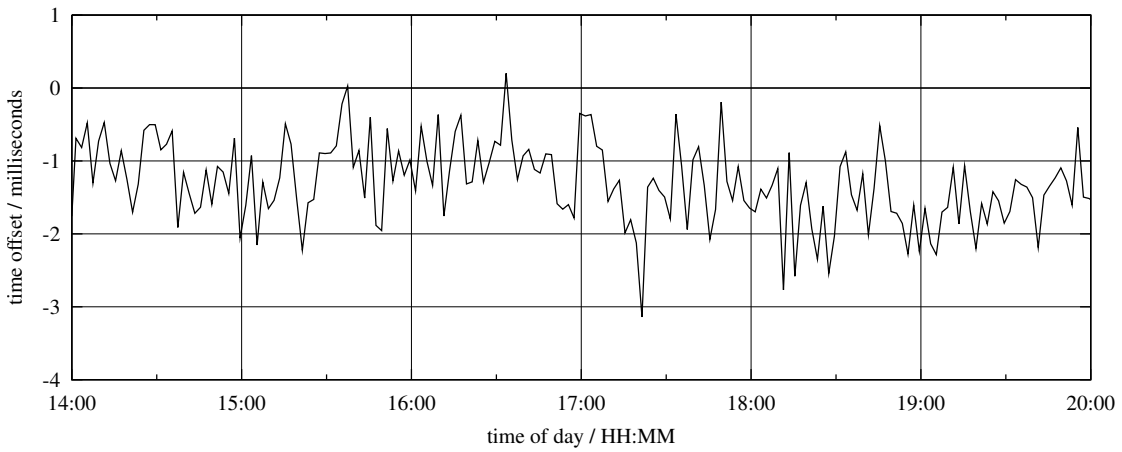


Figure 5.7.: Clock difference between DSPs in distant sensor nodes

6. Conclusion

This thesis presents an overview to the recurring problem of time synchronization. The fundamentals of time, timescales, clocks, remote clock reading, and synchronization are described. Especially clocks are treated thoroughly from several theoretical and practical perspectives.

An introduction to the wide design space of synchronization protocols is given, selected important contributions to the field are presented, and the sophisticated standard protocol for time synchronization in the Internet is described.

The architecture of the I-SENSE middleware shaped the design of its synchronization service. The design is a two level hierarchy. On the top level, state-of-the-art NTP software synchronizes the system time of the general purpose (Pentium M) processors. Redundant top level servers provide a high degree of fail safety. Internal or external synchronization is possible, with networked or directly attached time sources.

The second hierarchical level synchronizes the TMS320C64X signal processors within each sensor node to the local Pentium M processor. A custom protocol that combines the essentials of SNTP with some extensions is deployed. Its implementation is centered around a few object oriented clock abstractions. Its moderate complexity, the low resource demands, and the restriction to fixed point arithmetic fit well into embedded environments.

First evaluations confirmed that the implementation goal—time offsets between clocks of less than 5 ms—has been met. Most experiments even yielded sub-millisecond offsets.

6.1. Future Work

Evaluation methodology Further evaluation in a hard- and software environment that is closer (or identical) to actual outdoor deployment should be carried out. Measurement uncertainty could be reduced with calibrated time measurement hardware. Temperature data should be taken too, because rapid temperature changes are a challenge for synchronization.

Simulations Effects of ambient temperature changes in outdoor deployments could be evaluated with simulations. Either by full simulation in virtual time, or by injection of clock frequency disturbances into the present implementation in real time.

Confidence intervals NTP computes expected and maximum error intervals. This data is not used at present. If there is demand, intervals could be provided via an API to fusion tasks.

Interval clock The current implementation of the UTC timescale cannot guarantee strict monotonicity. Under rare circumstances (e. g., leap seconds, broken servers, intermittent network connectivity, startup with a broken RTC) backward steps in time can happen. For applications that cannot tolerate timescale discontinuities, a strictly monotonic second clock is provided. Its usefulness needs to be evaluated. Currently interval clocks are only synchronized within each I-SENSE sensor node. Inter-node synchronization could be added.

Low level timestamps At present Microsoft Windows operating systems have no support for low level timestamping of network packets. `WinPCap`, an open source packet capture library,[1] can provide packet timestamps with 1 µs resolution to applications. More accurate timestamps would improve synchronization precision.[2]

Controller design The present intra-node clock discipline is a preliminary ad-hoc design. Although it works well, a more elaborate design could increase synchronization precision.

[1] `WinPCap` is a Windows port of the widely used `libpcap` (packet capture) library for UNIX-like systems.

[2] If that approach is viable, the design decisions of section 4.2 might even be reconsidered.

A. List of Symbols

$\varphi(t)$	oscillator phase deviation [rad]
$\mathscr{L}$	SSB phase noise to carrier power ratio [dBc Hz^{-1}]
$MTIE$	Maximum Time Interval Error [s]
S_x	PSD of time fluctuations [$\mathrm{s}^2\,\mathrm{Hz}^{-1}$]
S_ϕ	PSD of phase fluctuations [$\mathrm{rad}^2\,\mathrm{Hz}^{-1}$]
S_y	PSD of fractional frequency fluctuations [Hz^{-1}]
$\langle X \rangle$	Statistical *expectation value* of X (a. k. a. $E[X]$)
Δ	difference
Δ^2	second order difference
$\lfloor x \rfloor$	floor function
σ_x^2	Time Variance [s^2]
σ_y^2	Two-Sample or Allan Variance [1]
Mod. σ_y^2	Modified Allan Variance [1]
TIE_{rms}	root mean square of Time Interval Error [s]
τ_0	shortest sampling period [s]
τ	(sub)sampling period [s]
θ	clock offset error [s]
V_0	nominal amplitude [V]
$\Delta V(t)$	amplitude fluctuations [V]
$v(t)$	oscillator output voltage [V]
ν_0	nominal oscillator frequency [Hz]
$x(t)$	time error (of oscillator phase) [s]
$y(t)$	fractional frequency deviation [1]
$\bar{y}(t)$	average fractional frequency deviation over a period [1]
$R(t)$	clock rate [s/s]

S	clock skew (fractional frequency offset) [s/s]
D	clock rate drift due to aging [s/s^2]
$E_i(t)$	rate error due to environmental conditions [s/s]
$\upsilon(t)$	random rate fluctuation (FM noise) [s/s]
$\rightarrow$	happened-before relation
$\nrightarrow$	not happened-before
$\Rightarrow$	total order based on happened-before relation
$\parallel$	concurrent
$:=$	assignment operator
$\approx$	approximately equal
δ	NTP round trip communication delay [s]
Δ	NTP root delay [s]
ε	NTP dispersion [s]
E	NTP root dispersion [s]
λ	NTP peer synchronization distance (i. e., max. abs. offset error) [s]
Λ	NTP system synchronization distance [s]
ρ	NTP precision [s]
Φ	NTP frequency tolerance constant = 15 ppm
φ	NTP jitter [s]
Δ_R	NTP root delay [s]
E_R	NTP root dispersion [s]
Θ	NTP combined system clock offset [s]
θ	NTP system jitter [s]
τ	NTP poll exponent [$\log_2$ s]

B. Abbreviations and Glossary

Accuracy	Closeness of agreement between a measured quantity value and a true quantity value of a measurand [BIP08].
ACPI	Advanced Configuration and Power Interface
AM	Amplitude Modulation
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
ASIC	Application Specific Integrated Circuit
ATA	Advanced Technology Attachment
BEV	Bundesamt für Eich- und Vermessungswesen. (The Austrian →NMI.)
BIH	Bureau International de l'Heure
BIOS	The Basic Input Output System is the firmware for x86 PCs.
BIPM	The Bureau International des Poids et Mesures located in Sevres (near Paris) maintains the timescales TAI and UTC.
CDMA	Code Division Multiple Access
CGPM	Conférence générale des poids et mesures
COTS	commercial off the shelf
DNS	Domain Name System
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
DTSS	Digital Time Synchronization Service
DUT1	UT1 – UTC
EIA	Electronic Industries Alliance
FIFO	First-In First-Out
FLL	Frequency Locked Loop
FPGA	Field Programmable Logic Array
FPU	Floating Point Unit

Frequency Instability The frequency change, typically averaged for an interval, t, with respect to another frequency. Generally one distinguishes between frequency drift effects and stochastic frequency fluctuations. Special variances have been developed for the characterization of these fluctuations [AAH97].

GMT Greenwich Mean Time was the official name for world time until 1972.

GNSS Global Navigation Satellite System

GPS Global Positioning System. The 24 satellites of the space segment carry accurate atomic clocks. Their signals allow high precision timing applications.

HPET High Precision Event Timer

IAU International Astronomical Union

IDE Integrated Development Environment

IEK Imaging Evaluation Kit

IERS International Earth Rotation and Reference Systems Service

IETF Internet Engineering Task Force

IGS International →GNSS Service

IP Internet Protocol

IQR Inter Quartile Range

IRIG Inter-Range Instrumentation Group

ISM band The industrial, scientific, and medical radio bands are defined by the →ITU-R.

ISO International Standards Organization

ITU The International Telecommunication Union is located in Geneva, Switzerland. The ITU is made up of three sectors:

ITU-T Telecommunication Standardization Sector,
ITU-R Radiocommunication Sector, and
ITU-D Telecommunication Development Sector.

JD Julian Day number of mean solar days (and decimal fractions thereof) elapsed since JD 0.0 which was at Greenwich mean noon of -4712 January 1.

LAN Local Area Network

LLR Linear Least Squares Regression

LORAN-C LOng Range Aid to Navigation is a terrestrial radio navigation system using low frequency radio.

MAC Media Access Control

MCXO Microcomputer Compensated Crystal Oscillator

Metrology The science of measurement and its application [BIP08].

MII Media Independent Interface

MPEG Motion Picture Experts Group

MJD Modified Julian Day. MJD was defined in the 1950s as (JD - 2400000.5). MJD 0.0 corresponds to 1858-11-17T00:00:00.

MTBF Mean Time Between Failures

MTIE Maximum Time Interval Error

NIC Network Interface Card

NMEA U. S. National Maritime Electronics Association

NMI National Metrology Institute

NIST U. S. National Institute of Standards and Technology

NTP Network Time Protocol

NTSC National Television System Committee

NVDK Network Video Developer's Kit

OCXO Oven Controlled Crystal Oscillator

PCC Processor Cycle Counter

PCI Peripheral Component Interconnect

PDU Protocol Data Unit

PICMG PCI industrial computer manufacturers group

PIT Originally the Programmable Interval Timer was an Intel 8353 or 8354 chip. Now it is integrated into the x86 PC chipsets.

PLL Phase Locked Loop

PM Phase Modulation

POSIX Portable Operating System Interface is a family of related standards specified by the IEEE to define the application programming interface for software compatible with variants of the Unix operating system. Formally designated as IEEE 1003 or ISO/IEC 9945.

PPS Pulse Per Second

Precision Closeness of agreement between indications or measured quantity values obtained by replicate measurements on the same or similar objects under specified conditions [BIP08]. *Caveat: "Precision" means many things throughout the literature. Often used as a synonym for →accuracy. In the context of →NTP [Mil06b]:* Minimum time required to read the system clock.

PRN Pseudo Random Noise

PSD Power Spectral Density

PTP The Precision Time Protocol is specified by the IEEE 1588 standard [IEE08].

PWM Pulse width modulation

QoS Quality of Service

RbXO Rubidium-Crystal Oscillator

Resolution Smallest change in a quantity being measured that causes a perceptible change in the corresponding indication [BIP08].

RFC Request For Comments

RMS Root Mean Square ($\sqrt{\langle X^2 \rangle}$)

Second A basic unit of measurement of time in the International System of Units [BIP06]. It is defined as the duration of 9,192,631,770 cycles of microwave light absorbed or emitted by the hyperfine transition of caesium-133 atoms in their ground state undisturbed by external fields.

RTC Real Time Clock. On the PC platform originally a Motorola MC146818A or Dallas Semiconductor DS122887 chip providing a battery backed time-of-day clock and 50/114 bytes of non-volatile RAM.

RTT Round Trip Time

RTTD Round Trip Transmission Delay, i. e., →RTT minus the interval between receiving the request and sending the response.

SI International System of Units (abbreviated from the French "Le Système international d'Unités")

SPXO Simple Packaged Crystal Oscillator

SSB Single Side Band

SSC Spread Spectrum Clocking

Stability →Frequency Instability

Stratum A level or layer in a hierarchical time or frequency distribution system.

Synchronization The times of clocks are in synchronization if their readings are the same after accounting for reference frame delays and relativistic effects. Synchronization needs to be specified to within some level of uncertainty [AAH97].

Syntonization The rates or frequencies of clocks are in syntonization if the rates are the same after accounting for reference frame corrections and relativistic effects. Syntonization needs to be specified to within some level of uncertainty [AAH97].

TAI International Atomic Time

TCP Transmission Control Protocol

TCXO Temperature Compensated XO

TDMA Time Division Multiple Access

Tick In system programming a *tick* is the event of a periodic timer interrupt which is used for timekeeping and scheduling purposes.

TIE Time Interval Error

Timescale Continuum of monotone-increasing values that denote time in some frame of reference.

Timestamp An unambiguous representation of some instant in time. Timestamps refer to a →timescale.

TRAIM Time Receiver Autonomous Integrity Monitoring

TSC Time Stamp Counter. A →PCC built into Intel x86 CPUs since the Pentium.

TUG Graz University of Technology

TWSTFT Two Way Satellite Time and Frequency Transfer

UDP User Datagram Protocol

UML Unified Modeling Language

UMTS Universal Mobile Telecommunications System

Uncertainty Parameter, associated with the result of a measurement, that characterizes the dispersion of values that could reasonably be attributed to the measurand [BIP08].

USB Universal Serial Bus

UTC Coordinated Universal Time as maintained by the →BIPM.

UT1 A timescale based on the rotation angle of earth and corrected for polar motion as maintained by the →IERS.

VCO Voltage Controlled Oscillator

VLBI Very Long Baseline Interferometry

WAN Wide Area Network

WSN Wireless Sensor Network

XO Quartz crystal oscillator

Bibliography

[AAH97] David W. Allan, Neil Ashby, and Cliff C. Hodge. The science of timekeeping. Technical report, Hewlett Packard Application Note 1289, 1997.

[ABC+92] David W. Allan, James A. Barnes, Franco Cordara, Michael Garvey, William Hanson, Jack Kusters, Robert Smythe, and Fred L. Walls. Precision oscillators: Dependence of frequency on temperature, humidity and pressure. In *Proceedings of the 46th IEEE Frequency Control Symposium*, pages 782–793, Hershey, PA, USA, May 27–29 1992.

[AD97] Steve Alexander and Ralph Droms. DHCP options and BOOTP vendor extensions. RFC2132, March 1997.

[AP98] Emmanuelle Anceaume and Isabelle Puaut. Performance evaluation of clock synchronization algorithms. Technical Report n3526, INRIA Rennes, October 1998.

[Ari05] Elisa Felicitas Arias. The metrology of time. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 363(1834):2289–2305, 2005.

[BIP06] BIPM. The international system of units. `http://www.bipm.org/utils/common/pdf/si_brochure_8_en.pdf`, May 2006. 8th edition.

[BIP07] BIPM. The international system of units. - appendix 2: Practical realization of the definition of the unit of time. Published in electronic form only: `http://www.bipm.org/utils/en/pdf/SIApp2_s_en.pdf`, April 2007.

[BIP08] BIPM. International vocabulary of metrology - basic and general concepts and associated terms (vim). `http://www.bipm.org/utils/common/documents/jcgm/JCGM_200_2008.pdf`, 2008.

[BM00] Stefano Bregni and Stefano Maccabruni. Fast computation of maximum time interval error by binary decomposition. *IEEE Transactions on Instrumentation and Measurement*, 49(6):1240–1244, December 2000.

[Bre97] Stefano Bregni. Clock stability characterization and measurement in telecommunications. *IEEE Transactions on Instrumentation and Measurement*, 46(6):1284–1294, December 1997.

[CBB05] Kendall Correll, Nick Barendt, and Michael Branicky. Design considerations for software only implementations of the IEEE 1588 precision time protocol. In *Conference on IEEE 1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, Zurich, Switzerland, October 2005. NIST.

[Cri89] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[DFH+08] L. Diduch, A. Fillinger, I. Hamchi, M. Hoarau, and V. Stanford. Synchronization of data streams in distributed realtime multimodal signal processing environments using commodity hardware. In *2008 IEEE International Conference on Multimedia & Expo*, pages 1145–1148, Hannover, Germany, June 2008.

[DHS84] Danny Dolev, Joe Halpern, and H. Raymond Strong. On the possibility and impossibility of achieving clock synchronization. In *Proceedings of the sixteenth annual ACM Symposium on Theory of Computing (STOC-84)*, pages 504–511, New York, NY, USA, 1984.

[EGE02] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. *ACM SIGOPS Operating Systems Review*, 36:147–163, 2002.

[ER03] Jeremy Elson and Kay Römer. Wireless sensor networks: A new regime for time synchronization. *SIGCOMM Computer Communication Review*, 33(1):149–154, 2003.

[Fid88] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In K. Raymond, editor, *Proceedings of the 11th Australian Computer Science Conference (ACSC'88)*, pages 56–66, Queensland, Australia, February 1988.

[Fle05] FlexRay Consortium. FlexRay Communications System Protocol Specification Version 2.1 Revision A. Available from `http://www.flexray.com/`, December 2005.

[GA05] Bernard Guinot and Elisa Felicitas Arias. Atomic time-keeping from 1955 to the present. *Metrologia*, 42:20–30, June 2005.

[Gil05] Patrick Gill. Optical frequency standards. *Metrologia*, 42(3):S125–S137, June 2005.

[GZ89] R. Gusella and S. Zatti. The accuracy of the clock synchronization achieved by TEMPO in Berkeley UNIX 4.3BSD. *IEEE Transactions on Software Engineering*, 15(7):847–853, 1989.

[HFB94] K.B. Hardin, J.T. Fessler, and D.R. Bush. Spread spectrum clock generation for the reduction of radiated emissions. In *IEEE International Symposium on Electromagnetic Compatibility*, pages 227–231, Chicago, IL, USA, August 1994.

[Hor04] Martin Horauer. *Clock Synchronization in Distributed Systems*. PhD thesis, Vienna University of Technology, Vienna, Austria, February 2004.

[HSK03] Roland Höller, Thilo Sauter, and Nikolaus Kerö. Embedded SynUTC and IEEE 1588 clock synchronization for industrial ethernet. In *Proc. of the 9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA '03)*, volume 1, pages 422–426, Lisbon, Portugal, September 2003.

[IEE99] IEEE Std 1139-1999. IEEE Standard Definitions of Physical Quantities for Fundamental Frequency and Time Metrology—Random Instabilities, July 1999.

[IEE08] IEEE Std 1588-2008. Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. (Revision of IEEE Std 1588-2002), July 2008.

[IRI04] IRIG Standard 200-04. Serial Time Code Formats. U. S. Army, Inter-Range Instrumentation Group, White Sands Missile Range, New Mexico, September 2004.

[IT96] ITU-T. Recommendation G.810: Definitions and terminology for synchronization networks, August 1996.

[Joh04] Svein Johannessen. Time synchronization in a local area network. *IEEE Control Systems Magazine*, 24(2):61–69, April 2004.

[Kal05] V. Kalusivalingam. Simple network time protocol (SNTP) configuration option for DHCPv6. RFC4075, May 2005.

[Kam02] Poul-Henning Kamp. Timecounters: Efficient and precise timekeeping in SMP kernels. In *Proceedings of the BSDCon Europe*, Amsterdam, the Netherlands, November 2002.

[KBC05] T. Kohno, A. Broido, and K.C. Claffy. Remote physical device fingerprinting. *IEEE Transactions on Dependable and Secure Computing*, 2(2):93–108, April–June 2005.

[KRT06] A. Klausner, B. Rinner, and A. Tengg. I-SENSE: Intelligent embedded multi-sensor fusion. In *Proceedings of the 4th IEEE International Workshop on Intelligent Solutions in Embedded Systems (WISES'06)*, pages 105–116, Vienna, Austria, June 2006.

[KTR08] Andreas Klausner, Allan Tengg, and Bernhard Rinner. Distributed multilevel data fusion for networked embedded systems. *IEEE Journal of Selected Topics in Signal Processing*, 2(4):538–555, August 2008.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Lev95] Judah Levine. An algorithm to synchronize the time of a computer to Universal Time. *IEEE/ACM Transactions on Networking (TON)*, 3(1):42–50, February 1995.

[Lev99] Judah Levine. Introduction to time and frequency metrology. *Review of Scientific Instruments*, 70(6):2567–2596, 1999.

[Lev08] Judah Levine. A review of time and frequency transfer methods. *Metrologia*, 45(6):S162–S174, December 2008.

[Lis93] Barbara Liskov. Practical uses of synchronized clocks in distributed systems. *Distributed Computing*, 6(4):211–219, 1993.

[LL84] Jennifer Lundelius and Nancy Lynch. A new fault-tolerant algorithm for clock synchronization. In *Proceedings of the Third ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing*, pages 75–88, Vancouver, B. C., Canada, August 1984.

[LM00] Judah Levine and David L. Mills. Using the network time protocol (NTP) to transmit international atomic time (TAI). In *Proceedings of the 32nd Annual Precise Time and Time Interval (PTTI) Meeting*, pages 431–437, Reston, VA, USA, November 2000.

[LMS85] Leslie Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, 1985.

[Loy97] Dietmar Loy. *GPS-Linked High Accuracy NTP Time Processor for Distributed Fault-Tolerant Real-Time Systems*. PhD thesis, TU Vienna, Österreichischer Kunst- und Kulturverlag, Vienna, Austria, 1997.

[LRV+07] R. Lutwak, A. Rashed, M. Varghese, G. Tepolt, J. LeBlanc, M. Mescher, DK Serkland, KM Geib, GM Peake, and S. Römisch. The chip-scale atomic clock – prototype evaluation. In *Proceedings of the 39th Annual Precise Time and Time Interval (PTTI) Meeting*, pages 269–290, Long Beach, CA, USA, November 2007.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[MD08] Hicham Marouani and Michel R. Dagenais. Internal clock drift estimation in computer clusters. *Journal of Computer Systems, Networks, and Communications*, vol. 2008, Article ID 583162, 7 pages, 2008.

[Mil81] David L. Mills. DCnet Internet clock service. RFC778, April 1981.

[Mil85] David L. Mills. Network time protocol (NTP). Network Working Group Request for Comments: 958, September 1985.

[Mil88] David L. Mills. Network time protocol (version 1) specification and implementation. Network Working Group Request for Comments: 1059, July 1988.

[Mil89] David L. Mills. Network time protocol (version 2) specification and implementation. Network Working Group Request for Comments: 1119, September 1989.

[Mil92] David L. Mills. Network time protocol (version 3) specification, implementation and analysis. RFC1305, March 1992.

[Mil95] David L. Mills. Simple network time protocol (SNTP). RFC1769, March 1995.

[Mil98] David L. Mills. Adaptive hybrid clock discipline algorithm for the network time protocol. *IEEE/ACM Transactions on Networking*, 6:505–514, 1998.

[Mil06a] David L. Mills. The autokey security architecture, protocol and algorithms. Technical Report 06-1-1, Network Working Group, University of Delaware, January 2006.

[Mil06b] David L. Mills. *Computer Network Time Synchronization – The Network Time Protocol*. Taylor & Francis, Boca Raton, FL, 2006.

[Mil06c] David L. Mills. Network time protocol version 4 reference and implementation guide. Technical Report 06-6-1, NTP Working Group, University of Delaware, June 2006.

[Mil06d] David L. Mills. Simple network time protocol (SNTP) version 4 for IPv4, IPv6 and OSI. RFC4330, January 2006.

[MK00] David L. Mills and Poul-Henning Kamp. The nanokernel. In *Proc. Precision Time and Time Interval (PTTI) Applications and Planning Meeting*, pages 423–430, Reston, VA, USA, November 2000.

[MMB+00] J. Mogul, D. Mills, J. Brittenson, J. Stone, and U. Windl. Pulse-per-second API for UNIX-like operating systems, Version 1.0. RFC2783, March 2000.

[MO83] Keith Marzullo and Susan Owicki. Maintaining the time in a distributed system. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 295–305, New York, NY, USA, 1983. ACM.

[NMM+01] R. A. Nelson, D. D. McCarthy, S. Malys, J. Levine, B. Guinot, H. F. Fliegel, R. L. Beard, and T. R. Bartholomew. The leap second: its history and possible future. *Metrologia*, 38:509–529, 2001.

[NPON02] Jorji Nonaka, Gerson H. Pfitscher, Katsumi Onisi, and Hideo Nakano. Low-cost hybrid internal clock synchronization mechanism for cots pc cluster (research note). In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 121–124, London, UK, 2002. Springer-Verlag.

[Put04] Otmar Putz. Hardware Dokumentation PCI-MF-Encoder Board. Austrian Research Center Seibersdorf. Unpublished hardware documentation (in German), v1.0, Juni 2004.

[RBM05] Kay Römer, Philipp Blum, and Lennart Meier. *Handbook of Sensor Networks: Algorithms and Architectures*, chapter Time synchronization and calibration in wireless sensor networks, pages 199–237. Wiley-Interscience, 2005.

[Ril03] William J. Riley. Techniques for frequency stability analysis. In *IEEE International Frequency Control Symposium*, Tampa, FL, 2003.

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

[RM04] Kay Römer and Friedemann Mattern. The design space of wireless sensor networks. *IEEE Wireless Communications*, 11(6):54–61, 2004.

[RSJ+05] G. Linn Roth, Paul Schick, James Jacoby, Chad Schweitzer, Dean Gervasi, and Eric Wiley. Enhanced or eLoran for time and frequency applications. In *Proceedings of the 2005 IEEE International Frequency Control Symposium and Exposition*, Vancouver, B. C., August 2005.

[RV09] Julien Ridoux and Darryl Veitch. Ten microseconds over lan, for free (extended). *IEEE Transactions on Instrumentation and Measurement (TIM)*, 58(6):1841–1848, June 2009.

[SA06] Katsuhisa Sato and Kazuyoshi Asari. Characteristics of time synchronization response of NTP clients on MS Windows os and Linux. In *Proceedings of the 38th Annual Precise Time and Time Interval (PTTI) Meeting*, pages 175–183, Washington D.C., USA, December 2006.

[SAHW90] D. B. Sullivan, D. W. Allan, D. A. Howe, and F. L. Walls. *Characterization of Clocks and Oscillators*. NIST Technical Note 1337, March 1990.

[SBK05] Bharath Sundararaman, Ugo Buy, and Ajay D. Kshemkalyani. Clock synchronization for wireless sensor networks: A survey. *Ad-Hoc Networks*, 3(3):281–323, March 2005.

[SCF+08] Thomas Schmid, Zainul Charbiwala, Jonathan Friedman, Young H. Cho, and Mani B. Srivastava. Exploiting manufacturing variations for compensating environment-induced clock drift in time synchronization. In *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems (SIGMETRICS'08)*, pages 97–108, Annapolis, Maryland, USA, June 2008.

[Sch87] Fred B. Schneider. Understanding protocols for byzantine clock synchronization. Technical Report TR 87–859, Cornell University, Dept. of Computer Science, Upson Hall, Ithaca, NY 14853, August 1987.

[Sch94a] Ulrich Schmid. An annotated bibliography on clock synchronization in distributed systems. Technical Report 183/1-45, Technische Universität Wien, Department of Automation, Vienna, Austria, December 1994.

[Sch94b] Fabio A. Schreiber. Is Time a Real Time? - An Overview of Time Ontology in Informatics. In W. A. Halang and A. D. Stoyenko, editors, *Real Time Computing*, pages 283–307. Springer, Berlin, Heidelberg, 1994.

[Sch00] Klaus Schossmaier. *Interval-Based Clock State and Rate Synchronization*. PhD thesis, TU Vienna, Österreichischer Kunst- und Kulturverlag, Vienna, Austria, 2000.

[SK92] Mukesh Singhal and Ajay Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43(1):47–52, 1992.

[SM08] Renaud Sirdey and François Maurice. A linear programming approach to highly precise clock synchronization over a packet network. *4OR: A Quarterly Journal of Operations Research*, 6(4):393–401, December 2008.

[SPV07] Suresh Siddha, Venkatesh Pallipadi, and Arjan Van De Ven. Getting maximum mileage out of tickless. In *Proceedings of the Linux Symposium*, pages 201–208, Ottawa, Ontario, Canada, June 27–30 2007.

[SWL90] Barbara Simons, Jennifer Lundelius Welch, and Nancy Lynch. An overview of clock synchronization. In *Fault-tolerant distributed computing*, volume 448 of *Springer Lecture Notes In Computer Science*, pages 84–96. Springer-Verlag, London, UK, 1990.

[SY04] Fikret Sivrikaya and Bülent Yener. Time synchronization in sensor networks: A survey. *IEEE Network*, 18(4):45–50, July/August 2004.

[TEFK05] D. Tsafrir, Y. Etsion, D.G. Feitelson, and S. Kirkpatrick. System noise, OS clock ticks, and fine-grained parallel applications. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 303–312. ACM New York, NY, USA, 2005.

[Ten08] Allan Tengg. The I-SENSE Communication. Unpublished technical documentation, 10 pages, 2008.

[Tex00] Texas Instruments. *DSP/BIOS II Technical Overview*. SPRA646, March 2000.

[Tex04a] Texas Instruments. *TMS320C6000 Chip Support Library API Reference Guide*. SPRU401J, August 2004.

[Tex04b] Texas Instruments. *TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide*. SPRU403G, April 2004.

[Tex04c] Texas Instruments. *TMS320C6000 DSP/BIOS User's Guide*. SPRU423D, April 2004.

[Tex05] Texas Instruments. *TMS320C6000 DSP 32-Bit Timer Reference Guide*. SPRU582B, January 2005.

[TKR07] Allan Tengg, Andreas Klausner, and Bernhard Rinner. I-SENSE: A light-weight middleware for embedded multi-sensor data-fusion. In *Proc. of the 5th Workshop on Intelligent Solutions in Embedded Systems (WISES'07)*, pages 165–177, Madrid, Spain, June 2007.

[Vig07] John R. Vig. Quartz crystal resonators and oscillators for frequency control and timing applications (Rev. 8.5.3.6). Technical report, US Army Communications-Electronics Research, Development & Engineering Center, Fort Monmouth, NJ, USA, January 2007.

[WG92] F.L. Walls and J.-J. Gagnepain. Environmental sensitivities of quartz oscillators. *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, 39(2):241–249, March 1992.

[YVS07] Suyoung Yoon, Chanchai Verrarittiphan, and Mihail L. Sichitiu. Tiny-sync: Tight time synchronization for wireless sensor networks. *ACM Transactions on Sensor Networks*, 3(2):34p, June 2007.

[ZLX02] Li Zhang, Zhen Liu, and Cathy Honghui Xia. Clock synchronization algorithms for network measurements. In *Proceedings of the Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'02)*, volume 1, pages 160–169, 2002.