Precision Time Synchronization on COTS Devices: A Reliable Primitive or Work-in-progress?

Vincent Bode
Department of Computer Science
Technical University of Munich
Munich, Germany
vincent.bode@tum.de

William Shen
Department of Computer Science
University of British Columbia
Vancouver, Canada
wshen05@student.ubc.ca

Arpan Gujarati
Department of Computer Science
University of British Columbia
Vancouver, Canada
arpanbg@cs.ubc.ca

Martin Schulz
Department of Computer Science
Technical University of Munich
Munich, Germany
schulzm@in.tum.de

Page limit: 11 pages excl. references

***Abstract*— Timekeeping systems are pervasive in the modern world and most real-time control systems could not function without them. However, while our embedded systems are becoming more and more distributed and have to fulfill ever more stringent fault-tolerance requirements, the ability to maintain a common notion of time across commodity computer networks is not yet perfected despite recent advancements in hardware support.**

We use the capabilities provided by our novel PTP benchmarking tool, PTP-Perf, to conduct a systematic study of four time synchronization protocols (PTPd, LinuxPTP, SPTP, and Chrony) across four embedded hardware testbeds (based on two generations of Raspberry-Pis, Xilinx AVNet, and NVIDIA Jetson boards). Our study determines how these protocols perform under normal operating conditions, how many system resources they consume when scaling the embedded systems network, and under what stress conditions they break down.

Overall, we find that Chrony is the most well-rounded system, with competitive synchronization accuracy, resilience and resource efficiency, while both LinuxPTP and SPTP offer synchronization accuracy but exhibit stability problems or require more resources to run. PTPd, which despite its age is still being deployed, shows shortcomings in synchronization accuracy due to the lack of support for modern hardware features. The most capable hardware platform evaluated is the Raspberry-Pi 5, both due to hardware timestamping support and importantly because of the real-time clock, which helps with fault-tolerance.

Critical challenges for maintaining accurate clock synchronization include resource contention (specifically network and memory, CPU is less of an issue) and faults on the master node when the master node lacks a real-time clock. We find that these challenges still need to be addressed before high-precision network time can truly be relied upon.

***Index Terms*—Time Synchronization, Fault Tolerance**

I. Introduction

From last discussion: Diagram showing Testbed setup (including Database)

From last discussion: Short section explaining what PTP-Perf is

Find places to insert

Time is both ubiquitous and critical, no computer system today can run without it. With the availability of the internet, satellite communications and digital clocks, we take for granted that we can tell the time anywhere and anytime. However, in communications, real-time systems and circuitry, we operate on a scale where things are not so simple, with sub-nanosecond-level accuracies gaining significance in areas like networks and chip design [1, 2].

Global Navigation Satellite Systems (GNNS), a classical application enabled by precision time-synchronization, relies on signal propagation delays to determine relative positioning [3]. High-quality location estimates hinge on accurate timing and -compensation mechanisms, so GNSS is frequently leveraged as a nanoscale clock source [4, 5]. The precision incurs expense though, it requires dedicated hardware, strains limited power budgets, and inhibits e.g. indoor deployments.

Timing is also critical for fault-tolerant systems. Architectures such as double- or triple-modular lock-step redundancy [6, 7, 8], where algorithms are run independently on different machines to automatically detect and correct errors, rely on a common notion of time and deadlines to make progress even when a machine has failed. Such systems are found frequently in high-reliability and -availability applications such as aviation, where computer systems are relied on for controlling machinery with strict timing requirements. Without a common notion of time, it would not be possible for resilient systems to judge internally missed deadlines correctly, thus preventing the system from providing proper error-detection and correction, voiding the fault-tolerance it was designed for.

@Arpan I removed the example, it's too much for the intro.

However, time is not only critical in fault-tolerant systems, it pervades all distributed systems. High-performance and datacenter computing are areas large amounts of resources are being spent on maximizing hardware utilization by reducing I/O or communications bottlenecks [9, 10, 11], and a common notion of time is a prerequisite enabling technology to effective profiling and optimization on distributed systems. Google and Facebook have come up with time synchronization implementations based on the notion of windows of uncertainty that can be used to ensure transaction ordering across a distributed system without an application having to explicitly synchronize, a property useful not only to databases but also to real-time control, fault detection/mitigation and parallel data structures.

cite

Time synchronization techniques also transfer to edge computing, where scheduling computation and communication to efficiently manage load can conserve power and connectivity resources, allowing applications to achieve more with less hardware. Even large scale deployments like the internet cannot function without timing, as SSL/TLS cryptographic certificate validation, renewal of leases (DHCP) and caches (DNS, HTTP), and various other state machines built into systems and protocols (TCP, firewalls, etc.) all rely on clocks to function.

Clearly, clock synchronization is essential in almost all aspects of computing, and there is a broad range of accuracies that we need to keep time in. But to what degree we can truly rely on modern computer timekeeping in distributed embedded systems when failures affect correctness? We need to evaluate what our systems are capable of and what sort of reliability guarantees they can provide for our algorithms to work with. Especially in industrial and embedded computing/IoT, deployments are often characterized by dependability/redundancy requirements and a general necessity to conserve all types of resources to reduce costs, as well as their relative isolation regarding connectivity. This sets it apart from more traditional computer timekeeping scenarios, where an external source of truth can be relied upon, here internal consistency is the primary motivation. Before we can trust a synchronization algorithm, we need to understand how it performs and what causes it to break, which is what we will be investigating throughout the rest of this paper.

We evaluate the performance and resilience of four implementations of prevalent time synchronization protocols for packet-switched Ethernet networks, including the Precision Time Protocol (PTP) and the Network Time Protocol (NTP), across four hardware testbeds, showing strengths and weaknesses in a variety of configurations. We offer the following contributions:

- Development of a tool, PTP-Perf, for fair and comparable cross-vendor evaluation of multiple time synchronization protocols and implementations,
- Establishment of a baseline of observed time-synchronization performance on several hardware testbeds across four vendors,
- Evaluation of synchronization resilience against adverse conditions/faults and possibilities of mitigating risk of synchronization failure,
- Assessment of deployability/resource consumption on embedded platforms,
- Provide lessons-learned and best practices for configuring time-synchronization deployments for high reliability and availability.

To the best of our knowledge, with PTP-Perf we provide the first data collection and analysis tooling available open-source that supports the empirical evaluation of multiple Ethernet-based time synchronization protocols and implementations across several embedded hardware testbeds with an explicit focus on dependability and fault-tolerance. The rest of this paper is structured into the following sections: Section II covers time synchronization and PTP background, as well as related work, while Section III looks at potential failure scenarios. Results are presented in Sections IV Baseline, V Resource Contention, VI Fault Tolerance and VII Resource Consumption. Finally, we present some lessons learned and best practices in Section VIII.

anonymize

II. Time Synchronization, PTP & Related Work

Check out related paper suggestions by Arpan

Over the years, many different protocols, algorithms and solutions to time synchronization across computer networks have been explored [12, 13, 14, 15, 16, 17, 18, 19]. In today's most widely deployed network stack for general purpose computing, IP, two protocols have established themselves as standard for time synchronization: the Network Time Protocol (NTP) [12] for Wide Area Networks (WANs) and the Precision Time Protocol (PTP) [20] for Local Area Networks (LANs) that require a greater degree of time synchronization precision. Both protocols are standardized and each have multiple implementations available at the time of writing.

Because we are interested in evaluating time synchronization for dependable embedded systems, we will focus on PTP and derivatives as they are designed to be deployed in controlled environments such as fault-tolerant industrial control networks. PTP has wide-spread adoption in general computing and telecommunications, and also serves as a foundation for technologies such as IEEE Time Sensitive Networking (TSN).

Clock synchronization protocols have been proposed for many different applications including packet-switched networks [21, 17, 15, 19] and their strengths and weaknesses have been evaluated analytically [22]. Due to the need for dependability, multiple studies have compared how time protocols can fail [23], how fault tolerance and redundancy can be offered [24], and how timing protocols might be attacked by malicious actors [25, 26]. However, we find that there is a lack of literature collecting empirical findings across implementations and protocols, with an early study [27] from 2006 focusing on technologies that were available at the time and more recent studies [28] not incorporating fault-tolerance as a central aspect.

We surveyed available PTP implementations and found that while there are plenty of commercial products available, the number of viable freely available implementations is rather limited. We excluded three vendors from our evaluation: OpenPTP [29] due to the unmaintained state of the project (last activity $>$ 10 years ago, the solution has since been commercialized), Timebeat [30] due to the reliance on heavy-weight infrastructure (specifically the Elasticsearch/Logstash/Kibana stack) making it unsuitable for embedded applications, and PPSI [31] due to stability issues caused by buffer overruns (bug report[32] has been filed), respectively. White rabbit [17], an open extension to PTP for ultra-precise (sub-nanosecond) timing was also reviewed, but the highly specialized hardware required (White Rabbit-capable synchronous Ethernet switches and NICs with *synctonization* support) makes it less suitable for embedded systems.

@arpan: cited as personal communication: submitted via email because no public issue tracker is available

TABLE I
SURVEYED TIME-SYNCHRONIZATION PROTOCOLS

Vendor	Protocol	Included	Notes
PTPd	PTP	✓ 2.3.1	Established implementation with derivatives
LinuxPTP	PTP	✓ 3.1.1	Advanced capabilities but specific to Linux
SPTP*	Custom	✓ b27bdba	Meta's custom time protocol
Chrony	NTP	✓ 4.3	Widespread NTP server/client
OpenPTP	PTP	× r2	Unmaintained
Timebeat	PTP	× 2.0.7	Unsuitable for embedded
PPSI	PTP	× 6.1	Critical bug
White Rabbit	Custom	× -	Specialized hardware requirements

* Note that we had to patch 32-bit support into SPTP for our ARMv7 boards.

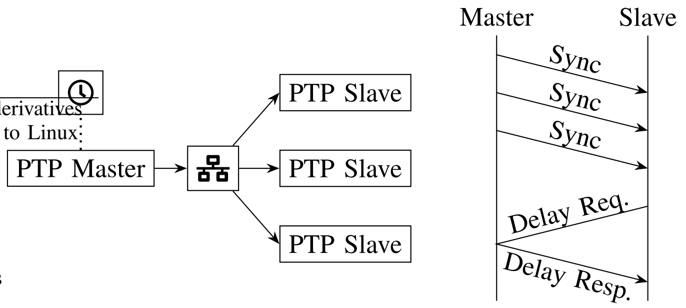


Fig. 1. A PTP master provides a synchronization signal to a number of slaves so that they can keep their local clock synchronized to the master's clock (left). The clock synchronization relies on two types of signals (right): a periodic synchronization signal to distribute the current master time and the delay request/response to estimate the propagation delay.

This left us with four vendors suitable for our analysis: PTPd, LinuxPTP, SPTP and Chrony (summarized in Table I). PTPd is a traditional implementation of the PTP protocol that, despite being less maintained in recent years [33] and lacking modern features such as hardware timestamping [34], spawned a variety of derivates (also commercial) and is still being deployed according to the Debian package tracker [35], perhaps due to its relative simplicity and wider support of non-Linux UNIX systems. LinuxPTP is the most deployed [35] open-source PTP solution on Debian, with the stated goals of providing robustness while integrating tightly with Linux and taking advantage of the kernel and hardware features provided to improve synchronization accuracy [36]. SPTP, a novel time-synchronization protocol consisting of the PTP4U server and the SPTP client described in a 2023 publication by Meta employees [37], was developed at Meta due to difficulties they encountered with deploying PTP. It claims to support a comparable level of time-synchronization performance while offering lower resource consumption and better resilience [37]. Finally, Chrony is included as a reference representing the state-of-the-art implementation of the Network Time Protocol (NTP). It is by far the most feature-complete time-synchronization implementation of the ones we evaluated and is far more widely deployed than PTP [35], likely due to its applicability to wide area networks. It serves as a baseline for the other implementations so that we can compare PTP implementation performance to the performance of general-purpose time synchronization.

cite

A. PTP – Background and Architecture

PTP operates across a network and has two types of endpoints: master nodes and slave nodes. A PTP master provides signals to the slaves (Figure 1), which each use the synchronization signal in combination with a path delay estimate to determine an estimate of the local clock offset to the master clock, which is subsequently used to discipline the local clock to keep it synchronized with the master's clock. While the actual protocol is slightly more complex and includes additional messages for synchronization leases and state management, these two core signals/message types are sufficient in principle to synchronize the system clocks. Note that obtaining a good clock signal on the master is considered out of scope for PTP, it is assumed that the master already has a high-quality clock source for the current time – preferably an external source like an atomic clock or a GPS signal, but the master clock can also be obtained from e.g. a different PTP domain.

B. PTP Profiles

PTP is built to be configurable, and settings include anything from the underlying transport (unicast/multicast packet switching), the delay mechanism to use (end-to-end or peer-to-peer), message frequencies and leases, as well as rules to discipline the clock. To reduce the complexity of configuring PTP, several so-called profiles are available that provide default settings for the specific use-case, such as general-purpose or ITU telecoms. To conduct our evaluation, we examine each vendor's default profile, as this profile is a widely-applicable general-purpose profile that does not require special configuration, and is thus likely to be deployed in many different contexts.

C. PTP Lifecycle

PTP clients traverse multiple stages in a lifecycle to synchronize their clock (Figure 2). At any point, unexpected conditions such as loss of connectivity can lead PTP to return to an earlier stage in the lifecycle, potentially changing the operating conditions.

S1. **Discovery** is the stage where remote clocks are identified, usually via periodic multicast announcements. Discovered clients are collected into a PTP domain.
S2. **Best Master Clock Algorithm (BMCA)** is the predefined algorithm used for each peer to determine whether it should become a master clock or a slave, which can be configured using priorities and clock properties [38]. By the end of this phase, each peer will become either a master clock or a slave. Slaves proceed to connect to, and negotiate with, the master clock.
S3. **Calibration** is a brief phase where no local clock modifications are made yet to allow the estimate of the offset to increase in precision through multiple synchronization intervals and path delay estimates.
S4. **Clock Step** Having arrived at a reasonably precise estimate of the offset, an attempt is made to immediately

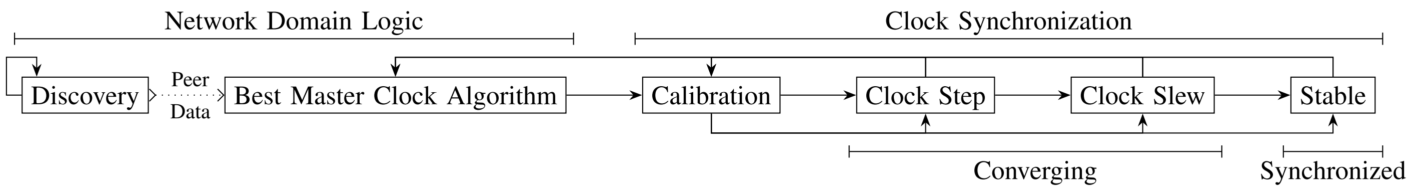


Fig. 2. Different stages in the PTP lifecycle that a PTP slave traverses while synchronizing its clock. For stability of the system clock, some state transitions are typically restricted.

rectify the offset by directly resetting the slave's local time. Since this is a direct violation of invariants I2 and potentially I1 (i.e. this is expected to break applications), it is only done when the offset is large (e.g. > 1 second) and many PTP profiles usually allow this to happen only once during the initial synchronization where only using clock-slew would take too long.

S5. **Clock Slew** is the second phase of convergence where time is made to pass slightly slower or slightly quicker via kernel parameters to fine-tune the clocks position, thus further reducing offset. The rate of change is usually limited via software and configuration parameters, e.g. the Linux kernel limits the rate of change to 500 parts per million, equivalent to 0.05% [39]. These limits ensure that the clock signal does not drift too rapidly, but can make convergence slow – the 0.05% limit implies that correcting 1 second of offset takes at least 33 minutes.

S6. **Stable** is the phase where the clock offset cannot be reduced further, and therefore the clocks are as synchronized as they can get. In this phase, further clock slew is happening to correct clock frequency differences and slight variations still occur due to lack of offset estimation precision, but at this point the clock signal is the most stable and most useful.

For calculating metrics on the quality of the clock synchronization, it is important to consider what should be measured. Often, comparisons are only meaningful when PTP is in comparable states (stable synchronization is usually of most interest), but ensuring this is the case is not always trivial, especially across vendors.

D. Synchronization Performance

It is generally understood that network clock synchronization accuracy is a function of the signal propagation delay and its variance [40]. A naive approach to clock synchronization that just sends a timestamp from the master to the slave would always be off by the propagation delay, but PTP uses path delay estimation compensate for the propagation delay thus increasing the accuracy. In an ideal world where there is no packet delay variation, the propagation delay could be mitigated entirely, but in reality we have multiple software and hardware components, like the kernel, network stack and network hardware, that each introduce latency variability, which in turn worsens the performance of the delay compensation. Effects, such as asymmetric latencies caused by uneven loads, that cannot easily be compensated for further reduce

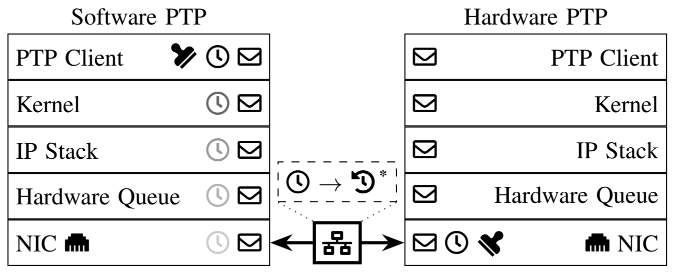


Fig. 3. Timestamping for PTP messages when using software timestamping (left) and hardware timestamping (right). For software timestamping, the timestamp is generated inside the PTP client and traverses many layers in both egress and ingress, causing additional path delay and delay variation. With hardware timestamping, a timestamp is only added to the message just before it is written to wire by the NIC, thus ensuring a more up-to-date timestamp is sent to the network.
*Network hardware (switches, routers, etc.) also introduces queuing delays. Special PTP-aware hardware can compensate for its own delays to further improve timestamping quality.

the synchronization accuracy. Thus, limiting the packet delay variation becomes a primary concern when tuning for precision and dependability.

E. Hardware Support

Since delay variation is a primary concern, techniques have been developed to reduce the variability. While PTP can run entirely in software, the path that packets need to traverse between two PTP clients not only includes several hardware components but also some software layers (Figure 3 left). Each component along the path introduces latency and packet delay variation, deteriorating the signal quality. With the appropriate hardware support, message timestamps can instead be generated directly by the NIC driver/hardware (Figure 3 right), which ensures that the timestamp is not affected by the layers above it. This increases the clock synchronization's resilience against interference from adverse conditions, such as high network-, CPU- or kernel load.

However, hardware timestamping alone does not guarantee a high level of dependability. In order to determine what can go wrong, we need to take a look at what constraints the timing system needs to observe and what might lead to potential failures.

III. Failure Scenarios

Should this be a subsection

Having an accurate clock synchronization across a network of machines can be just a nice-to-have for some applications, but applications that need to rely on the accuracy of the clock to perform their work need better guarantees than just an estimate of the accuracy. Resilience is a top priority when building dependable systems, and to ensure that a system can operate reliably engineers will often add safety margins to their systems so that deviations in components of the system will not cause complete system failure. In the context of time synchronization, since estimating the offset between clocks in software is imprecise even after applying denoising techniques, and the actual offset is influenced by a range of environmental conditions, the margin needs to be sufficiently large. The high level of uncertainty combined with high risk, major consequences of failure, and/or system criticality for certain application scenarios can merit safety factors of 5×, 10× or beyond depending on the application scenario. Naturally, the exact factor varies between use-cases, but for the purpose of our analysis of determining what conditions can lead to failures, we show that even highly conservative safety margins (e.g. three orders of magnitude, 1000×) over the baseline can be broken, reinforcing the necessity of careful consideration of possible causes of failure.

Would be nice to have something to cite here

A. *Sources of Error*

Error can originate from 2 principal sources: either the clock fails to perform according to specifications, or PTP's ability to measure and compensate the difference between clock sources is degraded or disrupted entirely. To understand what constitutes a failure in a timing system, we need to consider the guarantees that a timing system is required to provide to be useful and how they might be broken.

Invariants of Timing Sources: Many of the assumptions made when using a clock are made implicitly because they seem natural in the real world, and a surprising number of software systems rely on them perhaps without being specifically aware of it. Timing systems need to fulfill all the following invariants, ranked in decreasing order of importance:

@Arpan wants citations but as of now: Based on my understanding. Find if somebody has formalized this.

I1. **Time flows forward.** This is perhaps the most trivial assumption, and the one with the worst consequences when it is violated. Generally, software systems do not account for the possibility of time flowing backwards, and it can cause any number of cascading failures, with common ones including missed deadlines (in cyclic events) as well as data inconsistencies (through tasks being executed multiple times) and data loss (accidental overwriting of previously collected data, e.g. backups, based on timestamps). Due to the these risks, even minor backward time shifts can result in system stability issues.

I2. **Time passes continuously, it does not jump.** While systems expect time to pass, jumping too far into the future can result in load spikes through e.g. too many scheduled tasks arriving simultaneously or scheduled tasks being skipped entirely. For stability, direct modifications of a clock at system runtime should be avoided in order to avoid breaking any real-time constraints the system is trying to observe.

I3. **Time passes at a constant rate.** While this might seem like the most important constraint, it is actually the easiest one to bend without compromising the usefulness of the clock. No process is perfect, thus tolerance is always expected, and for clocks this means that a certain level of clock drift is specified in the data sheet. In fact, software is designed to cope with some amount of variance in the flow of time, as modern day systems are often too complex to schedule anything at a precise point in time. Making clocks tick slightly faster or slower can be used for synchronization with fewer stability implications (this is called a clock slew).

Even during our experiments, which were carefully designed to account for the unreliable telling of time, we ran into issues with all of the above since PTP inevitably needs to violate (potentially all of) these assumptions to synchronize the clocks when there is a difference. Correcting for this error is difficult when there is no second source of time to cross-validate with, which is why we eventually used an external time source for record-keeping. Generally, PTP implementations try to minimize disruptions by breaking the lowest priority invariant possible to keep clocks in sync, exploiting the fact that I3 is more of a soft requirement.

Multiple Sources of Time: When two or more clocks are at play, then another invariant arises:

I4. **Coherence between ordered readings of different clocks.** A reading of one clock that comes before another should always yield a timestamp smaller than or equal to the latter to allow comparing them to be useful. This is theoretically equivalent with obtaining the same arbitrary-precision timestamp when reading both clocks simultaneously, but this is not possible in practice. Since this condition cannot be perfectly fulfilled i.e. there is always a margin of error, the coherency is only considered to be violated when the order is not preserved even though the readings were further apart than the tolerance bound.

Establishing and maintaining coherency is PTP's primary purpose, and maintaining invariant I4 does not have a fixed priority, it sometimes merits breaking I3, I2 or even I1. However, for synchronization to even become possible two preconditions need to be fulfilled. These are the two principal challenges that PTP has to solve:

Find if somebody has formalized this

P1. **Clock difference can be measured.** The prerequisite to synchronizing two clocks is to be able to quantify the offset, which is difficult to do reliably across packet-switched networks with best-effort quality of service due to the inherent packet-delay variation present.

P2. **Clock difference can be corrected.** Even when a difference can be measured it does not automatically imply that it can be corrected. Modifying current time has system stability implications because it directly violates conditions I1-I3, so time synchronization programs frequently have rules in place limiting under what conditions which invariants may be violated.

In order for two clocks to stay synchronized, both conditions need to be continuously fulfilled to prevent divergence from occurring, which would cause invariant I4 to break down.

B. Consequences of Failing to Maintain Invariants

A violation of any of these conditions is an error in the timing system that can propagate throughout the entire system. In the case of the first three clock invariants, failures will cause local incoherence, potentially triggering cascading failure even in system components that do not interact across a network. On the other hand, problems with time synchronization prerequisites causes clocks to diverge (I4), which only becomes noticeable upon communication when the coherency breaks down. While this might seem less troublesome, the entire point of deploying PTP is to provide reliable and high accuracy time synchronization so that algorithms can use time for distributed scheduling, so a failure to provide this invalidates the usefulness of PTP itself.

The importance of maintaining the clock invariants can be easy to underestimate. While intuitively a small clock error might seem negligible for most timekeeping use cases, multiple production outages have been linked to timekeeping failures. One common cause is the existence of the leap-second, an extra second that occurs once every few years to keep the calendar in sync with the earth's rotation. Despite its relatively small magnitude of just one second, it has been known to cause problems with navigation/collision avoidance systems, deadlocks and livelocks occurring in the Linux kernel causing computer systems to freeze (both transient and persistent, meaning a server restart could not fix the problem), and multi-hour freezes in financial transaction processing systems [41]. Significant engineering work has gone into trying to mitigate the leap-second, with some solutions employing a leap-smear [42, 43] to avoid a time jump (thus breaking lower-priority condition I3 rather than I1/I2), but imperfections in the mitigation systems have in turn caused synchronization failures leading to accidental time jumps of 10 years magnitude [41], large enough to break even "simple" systems that we take for granted such as SSL. The leap-second is such a significant reliability problem that recently calls are being made to abolish it altogether rather than trying to deal with its consequences [44], a resolution which has since found its way into the definition of UTC [45]. When even well-vetted software such as the Linux kernel has issues with resilience against timing invariants breaking on a magnitude as small as just one second, it is easy to guess that the average user application will be much less resilient still, and no amount of replication can fix the problem.

Throughout this study, we will determine what can be expected of PTP, when and under what conditions the invariants and prerequisites inevitably break down under adverse conditions, and what this means for the ability of applications to reliably leverage Ethernet-based time synchronization.

IV. Baseline Results

A. Testbed – Hardware and Software

To conduct our evaluation we employ a total of four hardware testbeds. These include a cluster consisting of three Raspberry Pi 4 attached to an isolated wired Ethernet network via a single Gigabit Ethernet switch and a second cluster consisting of three Raspberry Pi 5 in the same hardware layout (both on Debian 12 and Linux 6.6.20). There are two key differences relevant to timekeeping between the Raspberry-Pi 4 and the Raspberry-Pi 5: The Raspberry-Pi 5 has support for PTP hardware timestamping on the network interface, and it has an integrated real-time clock (RTC) that can be powered by an external battery, both of which the Raspberry-Pi 4 lacks [46]. Furthermore, we utilize a cluster of four Xilinx ZUBoard 1CGs, which feature a combination of ARM Cortex A53 and ARM Cortex R5F cores. These boards were adapted to run Debian with the standard Xilinx Kernels for access to the larger package repository. The base kernel was 5.15, which we patched for R8152/R8153 support for our secondary Ethernet adapters. However, we observed that under Kernel 5.15 hardware timestamping only worked on the transmission path for this board but not on the receiving path, which degraded synchronization accuracy and prevented SPTP from running at all, so we replaced it with a Xilinx 6.6.10 kernel. Finally, we have NVIDIA Jetson TK-1 boards on 32-bit ARMv7, painstakingly updated to Ubuntu 22.04 LTS for software compability but still running the comparatively ancient NVIDIA-customized 3.10.40 kernel. This selection of hardware is representative of a range of embedded systems/edge devices available between 2014 and 2024, both 32-bit and 64-bit and with/without real-time clocks and hardware timestamping across a range of network interface manufacturers. All clusters are controlled by programmable power delivery units so that we can simulate hardware failures in each component individually.

B. Baseline: The 1-to-1 benchmark

We start by establishing a baseline for each tested system and vendor which we can later use as a comparison for different environment configurations. We collect data in 20 minute runs (corresponding to just above 1000 samples per run), while retaining the default setting of one synchronization and one measurement of the path delay per second, as our results suggest that a higher frequency does not significantly improve clock synchronization (we observe the best synchronization quality results for LinuxPTP on the Raspberry-Pi 4 at 2 samples/second or slower) while it does negatively impact the stability of the measurement (for the highest rate setting -7, corresponding to 128 samples/s, we observe a 7× worse median accuracy and 105× at the 95th percentile).

Figure 4 shows a sample run of the baseline for the PTPd vendor. In order to collect meaningful statistics, we apply some preprocessing to the collected profiles. The most important step is to remove the convergence phase (Figure 4 left) from the profile to avoid it skewing the summary statistics. slightly duplicated

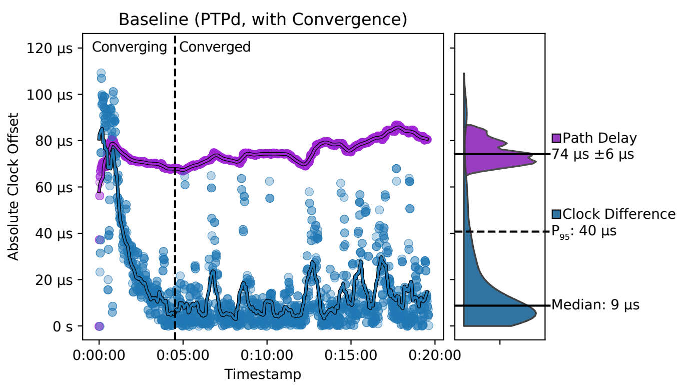


Fig. 4. A sample run of PTPd in its default configuration (left: scattered raw signal and denoised moving average, right: kernel density estimates). Clock synchronization can follow different convergence trajectories, which needs to be accounted for when calculating statistics. Because PTP uses path delay compensation, the final clock synchronization accuracy is much better than the one-way path delay.

However, the clock travels on a different trajectory during each convergence, and each trajectory may include discontinuities, rebounds, and clock steps. To eliminate this unwanted data, we apply a heuristic to the raw clock offset (which can have positive or negative values depending on the offset direction): If the sign of the offset flips repeatedly over a specified window of time, this implies that there is no clear pattern to the measured offset that PTP can compensate for, and thus the clock offset is close to stable at its minimum value. Intuitively, this is somewhat equivalent to the average signed clock offset being zero, but note that the two are distinct for certain corner cases, as there are ways to produce an average zero clock offset that still exhibit a signal trend (which we want to avoid) and there can be large magnitude measurement outliers that cause a non-zero average clock offset while showing no trend. Empirically, we find that using the frequency of sign switches in the clock offset produces much more accurate predictions of whether a clock is synchronized than relying on average (signed) offset.

From Figure 4 we observe that the clock offset is much lower than the corresponding path delay, due to the path delay compensation used by PTP clients. Across the tested vendors, we observe that the median absolute clock offset is between 11× (PTPd) and 93× (Chrony) smaller than the magnitude of the path delay on the Raspberry-Pi 4 cluster, and 100× (PTPd) to 430× (Chrony) smaller on the Raspberry-Pi 5 cluster, where hardware support is available. Logically, the absolute clock difference exhibits some amount of skew due to the fact that the value cannot be negative, and the signal can experience bursts of noise which makes determining the true clock offset more difficult. Across the profile however, the median clock offset represents the best estimate of the real clock offset, while the 95th percentile can be used as a bound where one can be reasonably sure that the true offset is lower.

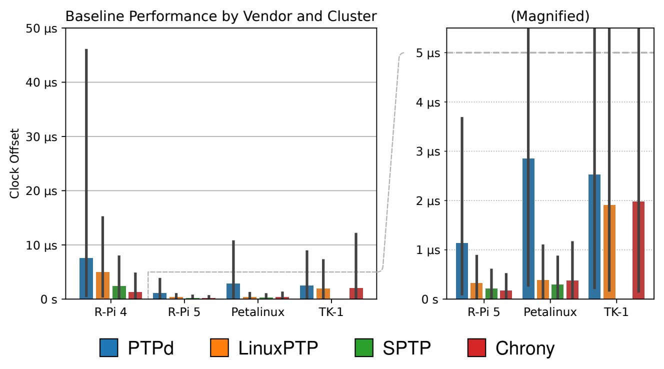


Fig. 5. Median baseline performance for all vendors, across our two hardware testbeds (left) and magnified for only the Raspberry-Pi 5 cluster (right). Error bands represent the 5th and 95th percentiles, respectively.

TABLE II
BASELINE VALUES BY VENDOR AND SYSTEM

		Clock Offset			
System	Vendor	Median	P_{95}	P_{99}	Max
Raspberry-Pi 4	PTPd	7.5 µs	45.9 µs	67.2 µs	154 µs
	LinuxPTP	5 µs	15 µs	20.1 µs	71 µs
	SPTP	2.4 µs	7.8 µs	11.4 µs	22 µs
	Chrony	1.3 µs	4.7 µs	7 µs	158 µs
Raspberry-Pi 5	PTPd	1.1 µs	3.7 µs	4.9 µs	13 µs
	LinuxPTP	0.3 µs	0.9 µs	1.1 µs	28 µs
	SPTP	0.2 µs	0.6 µs	0.8 µs	9 µs
	Chrony	0.2 µs	0.5 µs	6.1 µs	9 µs
Petalinux	PTPd	2.8 µs	10.6 µs	15.2 µs	28 µs
	LinuxPTP	0.4 µs	1.1 µs	1.4 µs	18 µs
	SPTP	0.3 µs	0.9 µs	1.1 µs	2 µs
	Chrony	0.4 µs	1.1 µs	1.6 µs	11 µs
Jetson TK1	PTPd	2.5 µs	8.8 µs	13.9 µs	75 µs
	LinuxPTP	1.9 µs	7.1 µs	11.8 µs	142 µs
	SPTP	– µs	– µs	– µs	– µs
	Chrony	2 µs	12 µs	21.6 µs	124 µs

C. *Vendors & Systems*

A logical first step is comparing each vendor's baseline across the systems. Figure 5 shows the four vendors on all four clusters, with the precise values available in Table II. Unfortunately, we had to exclude SPTP on the TK-1 cluster as it was not able to run even with our 32-bit patches, since the 3.10 kernel is missing support for some socket options required by SPTP. We observe that PTPd has the worst synchronization offset on all systems, with a median clock offset between 32% worse than the best performing vendor on Jetson TK1 and 855% on Petalinux. Chrony, on the other hand, has the best performance on the Raspberry-Pi systems, with the clock offset estimates at 68% and 62% lower than the mean performance, respectively. On the Petalinux and TK-1 systems,

INSERT NEW BASELINE DATA ONCE AVAILABLE.

That a regular NTP client can match or outperform all the tested PTP clients might come as a surprise, with PTP being engineered specifically for precision, but Chrony is very much state-of-the-art and can take advantage of the same hardware acceleration that our PTP clients can while providing a lot

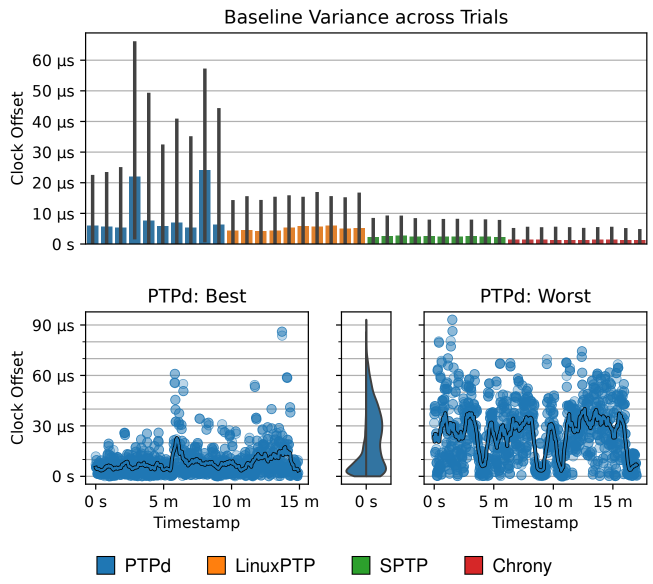


Fig. 6. Validating the baseline results by repeatedly measuring the baseline for both vendors on the Raspberry Pi 4 system. Top: The median absolute clock offset for each run, with error bars reaching from quantiles 0.05 to 0.95. Bottom: Timeseries profiles for the best run and the worst run under identical conditions, showing significant differences in noise levels.

this claim might need to be backed up

more features.

The most noticeable effect on the synchronization quality is the choice of hardware, which is expected since the Raspberry-Pi 5 offers hardware timestamping while the Raspberry-Pi 4 does not. The newer hardware's performance advantage ranges from 7× for PTPd to 15× for LinuxPTP. Curiously, although PTPd does not support hardware timestamping and thus cannot use hardware capabilities, it still offers around 15% of the performance of the top contender Chrony. While the difference is significant, it is not orders of magnitude better than the software-only implementation, showing that hardware timestamps alone are not a cure-all solution to clock-synchronization and cannot eliminate timing variations entirely.

Another aspect to notice is the difference between the median clock offset and the 95th percentile. Without hardware support, this difference can be rather large and has a high spread (between 3× for the most stable vendor LinuxPTP and 6× for the least stable vendor PTPd), whereas the magnitude is smaller when hardware support is offered on the Raspberry-Pi (2.7× LinuxPTP – 3.2× PTPd). This means that not only is the average performance improved, but the magnitude of outliers is reduced, which is especially important when considering the dependability aspect. For applications that need to rely on timing sources, this shows that hardware acceleration can make a significant impact, but of course this comes with the price tag associated with a more capable network interface.

D. *Reproducibility*

A key challenge with conducting high-quality performance studies is figuring out how to aggregate data to increase information density while avoiding the aggregation hiding important events such as outliers. While comparing clock-synchronization quantiles between vendors is certainly useful, in reality this does not tell the entire story because each baseline consists of a number of independent measurements, which in turn consists of around 1200 samples, all of which are affected by multiple sources of noise. As it turns out (Figure 6), not only is there a significant amount of noise during each run, but there are also discrepancies between measurement runs. This leads to completely different results for identical measurement runs despite decent measurement lengths and number of samples.

Find a good metric to use here

We observe that in absolute terms, PTPd is the main source of variance for both median/95-th percentile observed clock offsets and path delays across independent measurements. A simple restart of the PTPd client can suddenly cause the median latency to jump from 5 µs up to 24 µs, which corresponds to an increase of 350% not only momentarily, but throughout an entire run of 20 minutes. Fortunately, LinuxPTP produces a lot more stable results, with a smaller range of 4 µs and 6 µs between the best observed run and the worst observed run, corresponding to just 42% difference. The best vendor in this regard is Chrony, with both a relative range of just 12% and an absolute range of 0.2 µs.

To deal with the amount of noise, we apply a variety of noise reduction techniques. All measurements are interleaved for better compensation of noise that may be introduced through mechanisms outside our control, we repeat each observation at least 10 times for each vendor on each hardware platform (totaling in around 27 hours of runtime and 96000 samples collected) and between each measurement run, the entire cluster is restarted to ensure that no state is carried over, which would harm the independence of observations. We further filter data that contains too many holes, undercuts a threshold of the actual number of samples collected, or fails basic consistency checks (all of which are likely to lead to bad quality data). Otherwise, the setup is left untouched, so that the only differences that occur are in the internal state of PTP.

Needless to say, a vendor that cannot deliver stable timing guarantees is risky to deploy in a production environment that needs to rely upon the time source functioning. In the upcoming sections, we will examine whether PTP clients can be made resilient against potential external and internal influences.

V. Resource Contention

One aspect that influences how reliable PTP can operate is the outside interference originating from resource sharing. Through our research, we examined a range of shared resources that might impact synchronization accuracy and therefore need to be carefully managed so that PTP can provide synchronization even in the presence of stress. Unsurprisingly, the key resource tends to be the network.

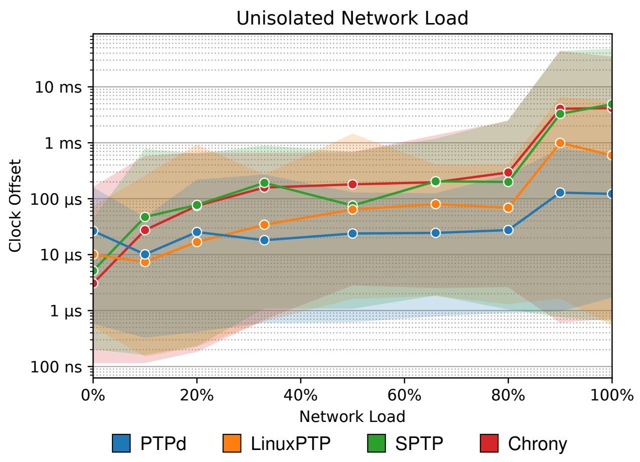


Fig. 7. Clock synchronization accuracy at different levels of network load. All vendors experience degradation at higher network loads, though the degree to which they are affected differs. Apart from a change in median clock offset, the 95th percentile is heavily affected, signaling larger and more frequent outliers.

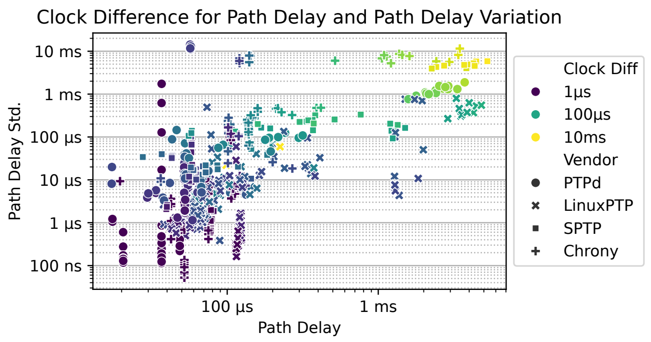


Fig. 8. Clock differences for all PTP implementations under various tested circumstances and across all systems depending on how high the path delay is and how frequently it varies, with each example representing a 20-minute run.

A. Network Contention

Like stated previously, the synchronization accuracy principally depends on the magnitude and the variation of the path delay. Intuition therefore suggests that the more network load is present, the more PTP will struggle to measure the clock offset consistently due to increased queuing in software and hardware causing delay and delay variation in delivery. We simulate network load artificially using iPerf, a traffic generator that allows us to target predefined data rates. By default, when a PTP vendor is installed on a target system and run, there are no special precautions in place to guard against contention, so a decrease in accuracy as the network load increases is exactly what we observe (Figure 7). Interestingly, the vendor with the smallest increase in clock offset is LinuxPTP, with an increase of just 1.3× (5 µs – 7 µs) from no network load to 100% network load. On the other hand, SPTP has a much higher ratio of 15× (2 µs – 35 µs), which can only be partially attributed to the fact that SPTP has a smaller baseline value. SPTP is the only vendor that relies solely on unicast message exchange (a design choice by Meta), whether this is why SPTP appears to be more susceptible to contention is subject to further investigation. The 95th percentiles reflect the behavior of the outliers, and here we can observe even larger ranges, with Chrony now showing a ratio of 1780× (5 µs – 8320 µs). This is clearly above and beyond even very generous safety margins, so we need to examine how the effect of network load can be mitigated.

Add vertical lines so its easier to judge

To show that the path delay is a good predictor of the synchronization quality, we can aggregate all 20-minute profiles across all our measurements and plot them by path delay and path delay variation (Figure 8). This shows us that generally, a high path delay and path delay variation is strongly correlated with a bad synchronization quality. However, each metric alone is not a good indicator as examples of high-quality synchronization are present wither either only high path delay or only high path delay variation. Notice also that there are multiple isolated clusters very far away from the norm (sometimes more than 10×), which, while small, still have multiple examples often originating from the same vendor. This suggests that the internal path delay compensation mechanisms of these vendors can be tripped off balance more or less consistently.

Make caption more clear

Note that under heavier network load PTP sometimes fails to even synchronize at all, when it gets stuck in an infinite loop of transmission timeouts indefinitely cycling between “state init” and “state faulty”. In this case the timing system breaks entirely, and no synchronization can be established. These measurements have been excluded from the shown data, but we encountered them rather frequently at high load levels (∼26% of 20 minute runs for 100% network load on the Raspberry-Pi 5 did not synchronize at all, with LinuxPTP accounting for most failures while Chrony produced no errors). Preventing this is paramount, as having no synchronization and potentially not even being aware of it is a lot worse than having a bad quality signal.

Depending on the possibility of dedicating exclusive resources to PTP, two principle solutions are viable: a software/hardware prioritization of traffic to reduce interference of lower priority traffic and providing a dedicated network interface for physical isolation. While the latter might seem like an expensive proposition and could therefore be less suited for embedded solutions, other use cases like industrial or datacenter settings will often already provide a second management interface separate from the application’s network. On our Raspberry-Pis, we emulate this by configuring the routing table so only PTP traffic can use the wired interface, with other traffic being relegated to the wireless interface. Figure 9 shows these possibilities, with the default unisolated case on the left and the baseline on the very right. Conforming to expectations, a physical isolation of PTP traffic can entirely mitigate the adverse effects of network load, with results sometimes even outperforming the baseline slightly (PTPd: -28%, LinuxPTP: 18%). Theoretically, the only interference

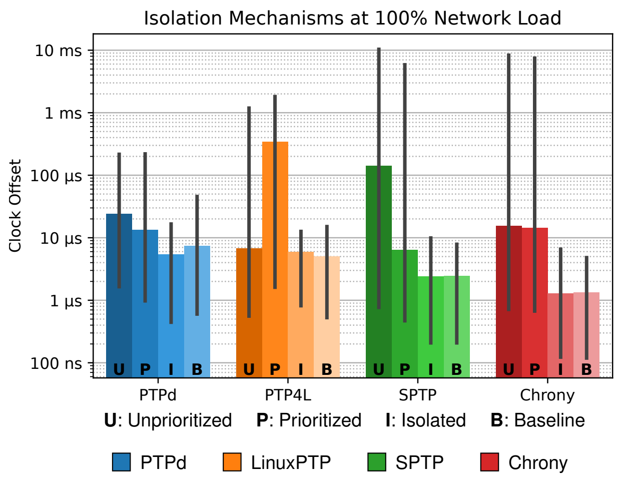


Fig. 9. Different possibilities of isolating the network load, compared to the baseline with no load. Both prioritization and physical isolation can often improve performance over the unprioritized default, however only isolation can match the performance of the baseline without resource contention.

that could traverse the isolation barrier is latency caused by cross-talk on the software network stack, which does not appear to be an issue here.

The prioritization-based solution, however, cannot quite achieve the same level of performance on our testbed. Prioritization is achieved through differentiated services (via the Differentiated Services Code Point, DSCP), and requires prioritization support on every networking software and hardware component on the critical path to perform optimally. While our switch and operating system claim to support the technology, this does not appear to be sufficient for complete traffic segregation. Compared to no traffic prioritization, activating DSCP results in performance improvements up to 6× for SPTP. However, one anomaly exists, which is LinuxPTP, where the median vendor performance actually decreases by 51×, while the 5th and 95th percentiles remain the within 3× the baseline. We have examined the affected profiles in more detail and can confirm that this effect is visible on all 9 trials encompassing approximately 10800 samples, so while we are confident of the validity of the observation, the reason why it occurs is currently unknown.

Perhaps recommendations

B. Other shared resources

We have further examined the effect of contention on other shared resources, and fortunately none of them cause synchronization quality degradation at the level that network congestion does. An educated guess for the next most important shared resource could be CPU, as difficulties to get scheduled timely intuitively could cause delays in packet processing. However, our data shows this is not the case on the Raspberry-Pi 4, as CPU contention only causes a maximum observed median degradation of just 15% (Chrony) and 28% at the 95th percentile on the Raspberry-Pi 4, which is small enough not to make a practical difference. This is likely due to the fact that PTP clients do not consume much processing time, which means they can easily get scheduled on Linux's Completely Fair Scheduler (CFS) even when idle computing time is scarce. Interestingly, CPU load can actually boost performance relative to the baseline for PTPd by around 39% for the median value and 60% for the 95th percentile (5 trials, ∼6000 samples). We assume that this benefit originates from less scheduling variance when the processor is kept busy due to less power saving. On the Raspberry-Pi 5, things look a little different. While Chrony, LinuxPTP and SPTP are mostly unaffected (up to just 17% overhead), PTPd has a more noticeable degradation in synchronization performance of up to 71%, which is however not nearly as critical as the previous degradation through network congestion.

The third principal hardware component where contention can occur is the memory hierarchy. Generally, PTP does not require a large amount of memory to function, but we do observe a moderate amount of sensitivity against cache contention on some systems (up to 8.9× with PTPd on Raspberry-Pi 5) while others are relatively unaffected (1× with SPTP on Petalinux), and memory bandwidth contention (between 0× with Chrony on Jetson TK-1 and 5× with Chrony on Raspberry-Pi 4), though cache contention appears to be slightly more important. In the end, both memory bandwidth contention and cache contention turn out to be more important than CPU contention.

TODO: RE-ENABLE ASSERTION WHEN DATA AVAILABLE

We have put the PTP clients through further stress tests (leveraging the capabilities provided by Stress-NG) and observe the following:

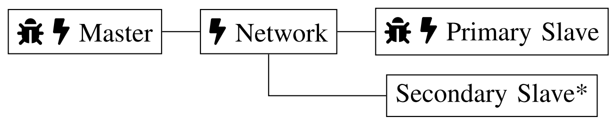
- Stressing time-related kernel features such as timers or alarms do not impact PTP performance significantly (with a maximum deviation from the baseline of under 50%).
- Installing cyclic tasks with a deadline in the schedule (`SCHED_DEADLINE`), which might cause PTP to get deferred when a real-time task needs to be scheduled, also do not cause significant adverse effects. This is good news as applications that require precise notions of time will often also be time-critical, and thus they usually use a real-time scheduling policy. However, note that things will likely start to fall apart when real-time tasks hog almost all available compute, as this will make it difficult for PTP to get scheduled at all unless it is promoted to a corresponding scheduling policy itself (which is not the case for the default profile).
- Finally, all PTP implementations also show good resilience against excessive context switching, which places stress on multiple software and hardware components simultaneously.

While resource contention, especially network congestion, can lead to degradation of clock synchronization performance, it is not the sole reason that the time synchronization system may fail.

VI. Fault Tolerance

Apart from resource contention, faults are another key threat to the functioning of the timing system. We examine faults

Review this: The information density of this section needs to be increased



*: For certain outlined scenarios, the secondary slave gets promoted to a failover master when the master fails.

Fig. 10. Simple fault-setup consisting of three clients, where potential software (🐞) or hardware (⚡) faults may occur. Since we exclude compound failures, only one fault can occur simultaneously, so no faults are triggered in the secondary slave.

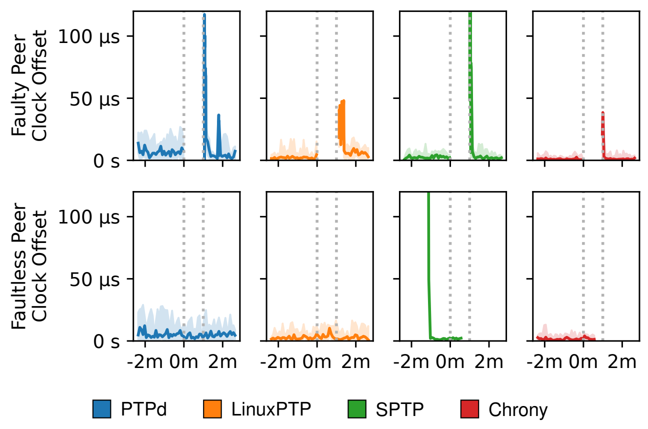


Fig. 11. Faults induced by a software crash on Raspberry-Pi 4, with a faulty client (shown on the left) and a second client as a control (right). Just after the fault, an increased offset can be observed as PTP tries to resynchronize the clock. Since these offsets occur randomly, we superimpose 2 trials for each vendor so that the worst case can be seen.

originating from three distinct causes: a fault in the network, a fault in the software, and a fault in the hardware. The software/hardware faults can either occur on a PTP slave, in which case they are less critical as the expectation is that the failure will be relatively contained within a single node, or on the PTP master, in which case they are more critical as the failure could propagate across the PTP domain. In high-reliability deployments, there will typically be a failover master to take over upon a fault in the master, so we examine that scenario too. Figure 10 shows an overview of the locations and types of faults that may occur. We emulate software faults by hard terminating the PTP client in software – in reality such a fault might come from within the client due to e.g. a bug, or from an external source (e.g. a process kill during an out-of-memory situation). We produce simulated hardware faults using our programmable power delivery unit, which is equivalent to a loss-of-power scenario, but real-life production hardware faults and system hang-ups could also come from other component failures.

A. *Software Fault – Slave*

The simplest possible fault is a PTP client crashing, because the only state that is lost is the state within the PTP slave client application (such as the state of the servo disciplining the clock). While the PTP application is down, the clock will continue to drift at the real clock drift rate minus the compensation rate that was last set by the PTP application. If the synchronization before the crash was stable, this observed drift rate is expected to be comparatively low. However, in the worst case when the clock slew is at the maximum rate (as will often be the case in the early stages of convergence), the software-driven drift can be up to 0.05% [39], which can cause a software-induced drift of 500 µs per second of downtime, or 30 ms per minute. We wish to determine empirically how big the offset is from a software fault when the clock is stable.

From the results (Figure 11 left), we can determine that the maximum observed offset after a 1-minute software fault among 2 trials is 247 µs for SPTP, which is around 101× worse than the median baseline performance. While this drift is definitely noticeable, it is fortunately not even close to the theoretical worst-case window of uncertainty of 30 ms, representing just 0.82% of the maximum software induced drift. We also observe that it is possible to get lucky: in some trials there is very little observed offset despite a full minute of downtime (just 38 µs with Chrony). In any case, all PTP implementations can quickly reconverge on the clock signal within a few seconds of restarting. Due to a quirk in how the PTP protocol works (slaves request synchronization signal leases from the master with a predetermined expiry), the master node will keep sending synchronization messages to the dead peer, which combined with the second lease that is issued when the PTP slave is restarted can help the peer synchronize quicker due to the higher influx of synchronization messages (this is sometimes referred to as an abandoned sync [21]).

import more trials

In high reliability deployments, replication is frequently applied to increase dependability. However, in order for replication to be useful, we need to ensure that a failure can be contained to the node of origin. In terms of PTP, this means that the fault on one slave should not affect the quality of synchronization of the other slave. To verify that this is the case, we attached a second control node to the same master (Figure 11 right), which shows us that there is no meaningful degradation of the performance of other nodes.

improve graphic

One important caveat is to be noted for PTPd: when multiple software faults start occurring (not necessarily in close proximity to each other), problems can start propagating. On our test systems, we observe that upon exactly the third software fault, PTPd will cause the network interface to fail. In fact, no amount of manually bringing the network down and back up or reloading the network interface driver can solve this problem, up until the system is restarted by hand. This is especially critical as it implies that not only will the timing system fail, but it will simultaneously bring down any applications that need to communicate over the interface as well. Ironically, in this case, rather than support the deployment high-reliability distributed systems, PTPd will actually harm said reliability. We thus advise caution when deploying PTPd or any of its derivatives that may contain the same bug.

B. *Hardware Fault – Slave*

A slightly more difficult scenario is a hardware fault on the slave, as not only will the internal PTP state be lost,

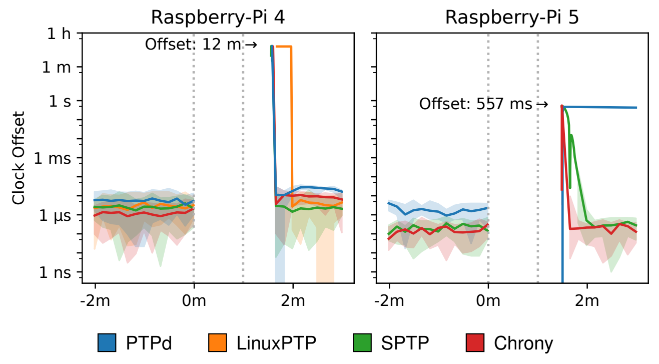


Fig. 12. A hardware fault on the slave of the Raspberry-Pi 4 cluster (left) and the Raspberry-Pi 5 cluster (right). Both slaves need to fully resynchronize after rebooting, but the Raspberry-Pi 5 has an advantage due to its hardware clock.

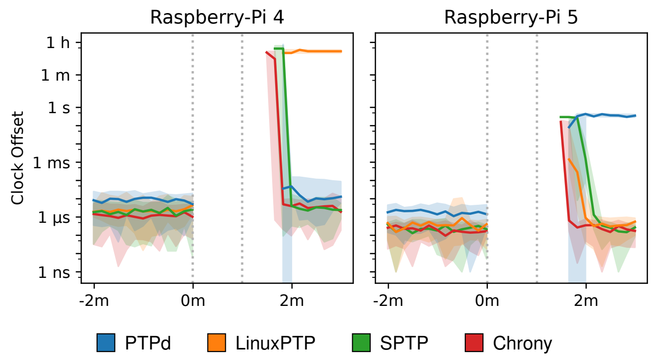


Fig. 14. A hardware fault on the master of the Raspberry-Pi 4 cluster (left) and the Raspberry-Pi 5 cluster (right). Slaves cannot compensate for unexpected changes in the announced time without correct configuration, so large clock offsets may be present practically indefinitely.

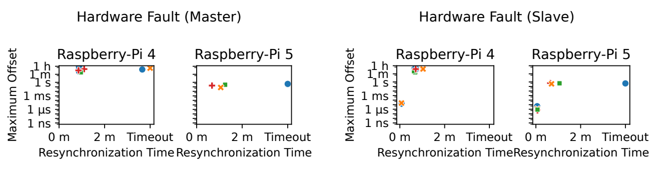


Fig. 13. WIP: Fault offset and time to resynchronize

but also the kernel state containing the current system time and the clock drift. This implies that PTP will need to fully reconverge, using the standard step-and-slew approach seen in the baseline. Because the Raspberry-Pi 4 (Figure 12 left) does not contain a real-time clock to preserve the system time across reboots, after a reboot a node will have a system time equal the last time that the time was persisted to disk if fake-hwclock [47] is active or a default value (e.g. the epoch in 1970) if fake-hwclock is not active. On our system (Raspbian, which bases on Debian), fake-hwclock is configured, so we observe a temporary deviation of just 12 min (on PTPd). To fix the large offset, PTP needs to step the clock (breaking invariant I2) and then takes some minutes to reconverge on the previous level of accuracy (in the case of PTPd, the other vendors are generally faster). On the other hand, the Raspberry-Pi 5 can maintain the current time using the hardware clock while it is not running. While the RTC generally has lower resolution than the internal oscillator (a common resolution is 1/32768), it generally has good long-term stability. This advantage shows itself in the observed maximum offset, which is a comparatively tiny 556700 µs (Chrony), a full 3 orders of magnitude smaller that can easily be compensated by clock slew only. A fault on a system without a hardware clock will thus cause an application to encounter large inconsistencies in timing in a place where it might not expect it (right in the middle of a run), while a simple RTC can mitigate this problem entirely. For deployments that are stuck on systems where an RTC is not viable (e.g. due to cost reasons), it is recommended to configure a delay for the application relaunch to allow PTP to resynchronize first.

Fault not visible on RPI-5 chart

cite

C. *Hardware Fault – Master*

PTP slaves are not the only type of node that can fail, the PTP master can just as well be affected by a disruption. Of all the scenarios, this is the most difficult one, as a failure will cause all slave nodes to become desynchronized. A failure on the master will inevitably also lead to hiccups on the announced time, especially in the embedded scenario where no external clock source for ground truth is available. On the Raspberry-Pi 4 (Figure 14 left), we observe that when the master restarts and now has a different reference time, there is once again a large offset between the master time and the slave time. However, unlike previously, the slave nodes are now no longer able to compensate for the offset by creating a clock step and instead are stuck at an (almost) constant offset to the master node. The reason for this lies in the default configuration of the PTP clients: It is assumed that stable time needs to be served to the application, so invariants I1 and I2 may not be broken, except once at start-up time when a single clock step is permissible to avoid excessive convergence time through clock slew. Because the slaves did not restart, they are not allowed to perform a clock-step, and while they can easily observe the difference they are stuck converging on it with a regular clock slew, which for the observed offset of 12 min is projected to require at least 24000 min – clearly impractical. Moreover, while the slaves will be converging on the new master clock at close to the maximum software drift rate simultaneously, we observe that they do not stay particularly well in sync between each other while this happens because the hardware clock drift continues to differ between nodes, which means that the remaining peers can also not rely on their clocks being close to each other either. This problem of indefinite clock-slew can be avoided by reconfiguring the PTP slave to also allow subsequent clock steps (which is disabled by default for safety), but the system should be well tested for resilience as the application will have to cope with a large magnitude rewinding of the clock (this is the worst case, breaking both I1 and I2). Experimentation on the stress-test tools and other applications show that many misbehave during a clock step,

Fix this chart too

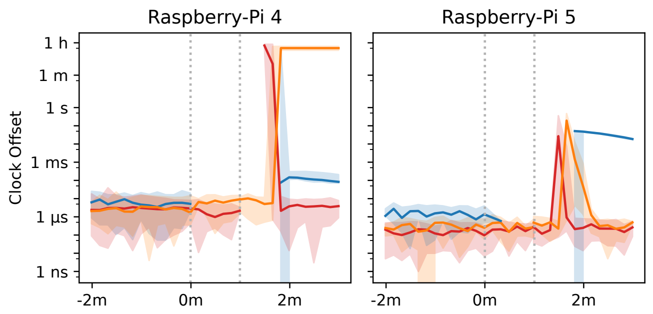


Fig. 15. Fault Tolerance with a Failover Master

so it cannot be assumed that an application is resilient simply because it uses the monotonic clock.

As already observed previously, the issue is easily mitigated by using an external clock that can maintain system time during downtime (e.g. Raspberry-Pi 5). The clock source does not need to be of particularly high quality if the goal is to only maintain sub-second synchronization accuracy. Should constraints only allow an external clock to be installed on a single node, intuition and our observations suggest that the primary target for installation should be the master node – it can propagate its stable time source to the slaves.

D. Hardware Fault – Failover Master

To increase service availability it is common to install a failover for any single point of failure in a distributed system so that the backup system can take over when the primary system fails. This is not only applicable for data-centers, but also applies to our minimal three node embedded setup, as a failover master will allow the two remaining nodes to remain synchronized with each other when the primary master fails. Fortunately, PTP was designed with this explicitly in mind so that the failover master does not necessarily require its own clock source, the failover node can automatically assume a slave state while the master is healthy (thus acquiring a clock signal consistent with the master's) and subsequently promote itself to a master when the original master is no longer healthy. Note that this setup is not supported in Meta's SPTP, as masters and slaves run different binaries and thus cannot switch roles (an external clock source would be necessary for two synchronized masters, we thus exclude SPTP from this evaluation).

We observe that the failover master can very rapidly take over the master role in the event of a failure, causing virtually no disruption in the timing service (Figure 15). However, the presence of a failover master cannot mitigate the problem of the master eventually restarting and disseminating a different time, which will be picked up by the failover master and the slaves, who will proceed to again get stuck in an indefinite clock slew. Due to the way the best master clock algorithm works (the same master clock will always be selected unless a configuration is changed thus shifting the prioritization), the failover master cannot re-teach the current network time to the master that has just restarted because it does not take priority over the actual master unless the master is demoted through configuration (e.g. using data from an external clock source).

data import failed

E. Network Fault

Another possible root cause for a fault is the network. While a network outage is generally bad news for reliable distributed systems that need to communicate, a total network outage is ironically easier for PTP to handle than a functioning network with high levels of congestion or a contained failure within any of the nodes. Since no state is lost, the servos are kept in holdover and PTP just has to detect when the network is available again to reconverge on the common network time. The effects of this scenario are not particularly surprising, so we do not show it separately.

include perhaps 1 number

VII. Resource Consumption

Find where to place this section.

Finally, embedded systems need to take particular care of the resource consumption of their applications. Since time synchronization fulfills only a supporting role in the overall deployment, it is important that it does not detract significantly from the resources available for the actual application. This is not only true for resources with a hard limit (compute, memory, etc.) but also for "soft" resources like power draw and heat dissipation. Especially SPTP was created with the intention of less resource consumption in mind [21], so we will verify whether reality holds up to these claims.

While some PTP systems are specifically tied to Linux and thus come with the associated resource requirements, others can also be deployed in more lightweight environments. On low capability devices, ROM/flash space is generally restricted, so the footprint becomes a concern. PTPd has the comparatively lightest footprint, with executables, data and dependencies totaling at around 840 KB after removal of pre-packaged documentation and dependencies that can reasonably be expected to already be available (e.g. lib-c). The LinuxPTP package is larger but requires fewer dependencies, thus the resulting footprint of 970 KB is around the same. SPTP (21 MB) and Chrony (12 MB) are both more than an order of magnitude larger due to the integrated Go runtime and the comparatively large number of dependencies, respectively. Note that the size of SPTP could be reduced by around 40% by only deploying the master or client executable and the Chrony footprint is significantly smaller if the larger dependencies (iproute2 + libgnutls30 + tzdata = 10 MB) are already available. In combination with an embedded Linux distribution, the more lightweight systems could reasonably be deployed on 16 MB flash devices, although this would likely not leave sufficient headroom for the actual application – thus 32 MB is the more realistic lower bound for PTP deployments.

Closely tied to the storage footprint is the memory footprint. LinuxPTP (250 KB-400 KB), Chrony (800K̃B-1 MB) and PTPD (∼1 MB) all have reasonably small unique set sizes, with resident set sizes around 2-4 MB. SPTP requires significantly more, with unique and resident set ranging between 8-10 MB for the master node and 15-16 MB on the slaves.

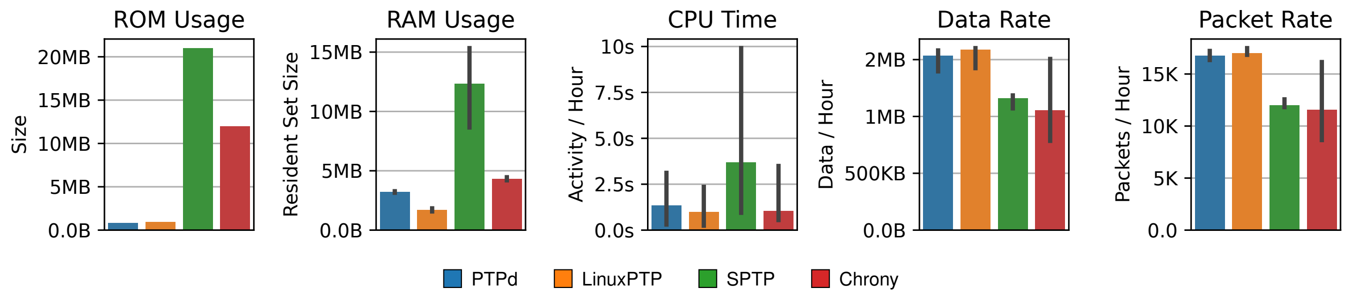


Fig. 16. Resource consumption statistics across vendors and systems. In general, PTP systems use few onboard resources but generate more network traffic, while the evaluated NTP and SPTP implementations reduce network utilization at the cost of additional ROM and other on-board resources.

Also notable is the virtual memory allocated by SPTP, 3 GB on the master and 1 GB on the client: over 100x more space than the next most hungry implementation (Chrony at 11 MB) and enough to cause potential issues on 16-bit deployments and platforms that don't have large amounts of virtual memory available.

Aside from deployability, power consumption is another big concern especially for mobile deployments. Using consumed CPU time as a heuristic for power consumption, we observe that LinuxPTP and PTPd consume the fewest compute, while Chrony requires around 30%−150% more. SPTP consumes the most cycles, with around 300% more consumption than LinuxPTP and PTPd, causing the system to run 0.3°C−0.6°C hotter on both the Raspberry-Pi 4 cluster and the Raspberry-Pi 5 cluster (indicative of the additional energy expended). Overall, the consumption of compute power is however relatively small even for SPTP, with around 10 s of compute consumed in 1 hour of runtime on our comparatively powerful ARM Cortex-A72.

cite

Network traffic can also be a concern for low-throughput networks. Previous studies have found that PTP tends to consume a negligible amount of bandwidth. Our results suggest that SPTP consumes the fewest network resources overall (around 1.1 MB per hour for one client), only surpassed by the Chrony master endpoint, which uses even less. The Chrony slaves consume more traffic, up to double the data rate. So while Chrony might have a better total network traffic score, there is a skew between the master and the slaves. PTPd and LinuxPTP are roughly equal on both data and packet rates since they implement the same protocol, totaling around 80% above SPTP's resource consumption. Thus, the claim that traffic can be reduced by simplifying the PTP protocol [21] using SPTP appears valid, however the NTP client Chrony shows roughly comparable efficiency using a well established protocol.

check whether this can be converted into a sneaky scalability analysis for resource consumption between 2-6 peers.

Overall, despite SPTP's promise of better resource efficiency, we find that SPTP surpasses the other vendors only in network traffic efficiency (likely due to the simplified protocol), but uses considerably more ROM, RAM, and compute (Figure 16). Thus, it is actually less well suited to embedded deployments than the more widespread PTP vendors. However, this does not necessarily contradict the claims made in the SPTP article [21], since the article targeted mass-scale deployments of 100K clients, a scale that is highly uncommon in the embedded world. For minimizing resource consumption on embedded systems, we recommend LinuxPTP foremost: it has the highest efficiency in on-board resource consumption at the cost of comparatively high network traffic, but it outperforms the other suitable vendor PTPd in synchronization quality.

VIII. Learnings and Conclusion

From our experiments we derive the following learnings and best practices:

a) Choice of Vendor: We have found there to be substantial synchronization performance and reliability differences between the vendors. While deploying PTPd may be tempting due to its simplicity and maturity, the fact that it can soft-brick the network driver along with its often order of magnitudes slower clock convergence should be enough of a red flag for projects to steer clear of it (and perhaps its derivatives), even if it can offer comparable accuracy with less effort required for setup (of the tested vendors, it was nevertheless the least accurate). Meta's SPTP was developed specifically with datacenter applications in mind and claims resource consumption advantages, however at the time of writing it comes with some limitations (the reduction in data and packet rates appears to be a tradeoff with higher ROM, RAM and compute usage), and most importantly a lack of maturity due to the recent release (this can be easily confirmed by a quick inspection of the documentation). This leaves LinuxPTP for PTP and Chrony as the NTP alternative – perhaps surprisingly, Chrony appears to outperform LinuxPTP in terms of synchronization quality on our system. However, Chrony requires several times more ROM space, thus making it more difficult to deploy in resource-constrained environments. In the end, both are quite mature and widely adopted, the better fit ultimately depends on the type of deployment.

This should be mentioned explicitly somewhere in the baseline

b) Guarding against Resource Contention: When dealing with resource contention, we observe that network congestion is the primary cause of degradation and that the ideal setup physically separates PTP traffic from application traffic (using e.g. a secondary management network). While certain deployments (e.g. datacenter or industrial IoT) might already feature secondary networks, this solution is often an expensive proposition and embedded deployments might opt for the software solution of traffic prioritization instead. However, setting a DSCP priority is not a cure-all for this case: we have shown that in some of our setups applying prioritization can lead to a considerable decrease in synchronization quality (this anomaly likely depends on the combination of priorities, NIC hardware queues and queuing disciplines, as well as network hardware). Thus, using a software prioritization solution requires proper testing before it can be relied upon. Using the default configuration of no prioritization at all is not recommended, as the 95th percentile clock offset under network load can reach as high as 1800× the baseline's 95th percentile. Other resources can also play minor roles in the clock synchronization accuracy (with cache contention and memory bandwidth bottlenecks being the most notable), however they do not get anywhere near the magnitude of network contention.

c) Resilience against Faults: While the most beneficial piece of hardware support for PTP might seem to be a NIC's hardware timestamping capability to protect against queuing and network delays, it turns out that it is actually more important to have an external clock source available to protect against unexpected conditions when failures occur. We strongly recommend using at least a real-time clock (or a better quality clock source such as GNNS) on the master node (and preferably on slave nodes too) to mitigate inconsistencies that inevitably happen when a node loses its state and therefore its system time. Alternatively, if internet access is available, then the master can import its time from a reliable network source, which achieves the same purpose. In lieu of these solutions (e.g. in isolated embedded settings), careful PTP configuration is necessary, as with the defaults edge cases may occur where clocks differ on the order of magnitude of minutes or years indefinitely, something that cannot be fixed solely by e.g. deploying a failover.

While time synchronization capabilities across packet-switched networks are evolving especially with the advent of TSN, it will probably be some time before we see out-of-the-box synchronization capabilities that are dependable and resilient enough to allow the deployment of synchronization-free distributed algorithms that operate at a timescale of interest to real-time systems while maintaining strict correctness requirements. There are simply too many ways yet to break any bound one might place on a clock signal difference through either bad configuration or hardware limitations. Thus, for the time being, the most reliable way to enforce ordering continues to be manual synchronization as part of a distributed algorithm. However, we invite readers to use the experimental capabilities offered by PTP-Perf to evaluate upcoming hardware and software capabilities to determine whether this might change in the future, and we hope that our best practices prove useful for real-world deployments.

Page limit: 11 pages

References

[1] S. Ibanez *et al.*, "The nanopu: A nanosecond network stack for datacenters," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, USENIX Association, Jul. 2021, pp. 239–256, ISBN: 978-1-939133-22-9. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/ibanez.

[2] Y. Rong *et al.*, "High-performance distributed power electronics communication network design with 5 gbps data rate and sub-nanosecond synchronization accuracy," in *2021 IEEE Energy Conversion Congress and Exposition (ECCE)*, 2021, pp. 5907–5911.

[3] R. B. Langley, P. J. Teunissen, and O. Montenbruck, "Introduction to gnss," in *Springer Handbook of Global Navigation Satellite Systems*, P. J. Teunissen and O. Montenbruck, Eds. Cham: Springer International Publishing, 2017, pp. 3–23, ISBN: 978-3-319-42928-1. [Online]. Available: https://doi.org/10.1007/978-3-319-42928-1_1.

[4] P. Tavella and G. Petit, "Precise time scales and navigation systems: Mutual benefits of timekeeping and positioning," *Satellite Navigation*, vol. 1, pp. 1–12, 2020.

[5] W. Guo *et al.*, "Foundation and performance evaluation of real-time gnss high-precision one-way timing system," *GPS Solutions*, vol. 23, pp. 1–11, 2019.

[6] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM journal of research and development*, vol. 6, no. 2, pp. 200–209, 1962.

[7] J. Abraham and D. Siewiorek, "An algorithm for the accurate reliability evaluation of triple modular redundancy networks," *IEEE Transactions on Computers*, vol. C-23, no. 7, pp. 682–692, 1974.

[8] X. Iturbe, B. Venu, E. Ozer, J.-L. Poupat, G. Gimenez, and H.-U. Zurek, "The arm triple core lock-step (tcls) processor," *ACM Trans. Comput. Syst.*, vol. 36, no. 3, Jun. 2019, ISSN: 0734-2071. [Online]. Available: https://doi.org/10.1145/3323917.

[9] M. Shantharam, M. Tatineni, D. Choi, and A. Majumdar, "Understanding i/o bottlenecks and tuning for high performance i/o on large hpc systems: A case study," in *Proceedings of the Practice and Experience on Advanced Research Computing*, 2018, pp. 1–6.

[10] A. Vasudeva, "Solving the io bottleneck in nextgen datacenters & cloud computing," *IMEX Research-Cloud Infrastructure Report (2009–11)*, pp. 1–20,

[11] P. Harrington, W. Yoo, A. Sim, and K. Wu, "Diagnosing parallel i/o bottlenecks in hpc applications," in *International Conference for High Performance Computing Networking Storage and Analysis (SCI7) ACM Student Research Competition (SRC)*, 2017, p. 4.

[12] D. Mills, J. Martin, J. Burbank, and W. Kasch, *Rfc 5905: Network time protocol version 4: Protocol and algorithms specification*, USA, 2010.

[13] D. Mills, *Rfc 4330: Simple network time protocol (sntp) version 4 for ipv4, ipv6 and osi*, USA, 2006.

[14] R. Cochran and C. Marinescu, "Design and implementation of a ptp clock infrastructure for the linux kernel,"

in *2010 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, 2010, pp. 116–121.

[15] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi, "The flooding time synchronization protocol," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, 2004, pp. 39–49.

[16] C. Lenzen, P. Sommer, and R. Wattenhofer, "Pulsesync: An efficient and scalable clock synchronization protocol," *IEEE/ACM Transactions on Networking*, vol. 23, no. 3, pp. 717–727, 2014.

[17] M. Lipiński, T. Włostowski, J. Serrano, and P. Alvarez, "White rabbit: A ptp application for robust subnanosecond synchronization," in *2011 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, 2011, pp. 25–30.

[18] M. Akhlaq and T. R. Sheltami, "Rtsp: An accurate and energy-efficient protocol for clock synchronization in wsns," *IEEE Transactions on Instrumentation and Measurement*, vol. 62, no. 3, pp. 578–589, 2013.

[19] F. Gong and M. L. Sichitiu, "Cesp: A low-power high-accuracy time synchronization protocol," *IEEE Transactions on Vehicular Technology*, vol. 65, no. 4, pp. 2387–2396, 2015.

[20] "Ieee standard for a precision clock synchronization protocol for networked measurement and control systems," *IEEE Std 1588-2019 (Revision ofIEEE Std 1588-2008)*, pp. 1–499, 2020.

[21] O. Obleukhov, A. Bulimov, and A. Byagowi, "Simple precision time protocol (sptp)," in *2023 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS)*, 2023, pp. 1–6.

[22] M. Lévesque and D. Tipper, "A survey of clock synchronization over packet-switched networks," *IEEE Communications Surveys and Tutorials*, vol. 18, no. 4, pp. 2926–2947, 2016.

[23] B. Ferencz and T. Kovácsházy, "Effects of runtime failures in ieee 1588 clock networks," in *2017 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*, 2017, pp. 1–6.

[24] P. Ramanathan, K. Shin, and R. Butler, "Fault-tolerant clock synchronization in distributed systems," *Computer*, vol. 23, no. 10, pp. 33–42, 1990.

[25] W. Alghamdi and M. Schukat, "An analysis of internal attacks on ptp-based time synchronization networks," *NUI Galway*, 2022.

[26] S. Shi *et al.*, "Ms-ptp: Protecting network timing from byzantine attacks," in *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '23, Guildford, United Kingdom: Association for Computing Machinery, 2023, pp. 61–71, ISBN: 9781450398596. [Online]. Available: https://doi.org/10.1145/3558482.3590184.

[27] T. Neagoe, V. Cristea, and L. Banica, "Ntp versus ptp in computer networks clock synchronization," in *2006 IEEE International Symposium on Industrial Electronics*, vol. 1, 2006, pp. 317–362.

[28] S. J. Wissow, "Time enough: Synchronization for latency measurement," Ph.D. dissertation, University of New Hampshire, 2020.

[29] Stefan Tauner. "OpenPTP - Precision Time Protocol Implementation." (2012), [Online]. Available: https://github.com/stefanct/openptp (visited on 04/23/2024).

[30] "Timebeat Installation Overview," Timebeat.app. (2022), [Online]. Available: https://support.timebeat.app/hc/en-gb/articles/360021334279-Timebeat-Installation-Before-you-begin (visited on 04/23/2024).

[31] P. Fezzardi, M. Lipiński, A. Rubini, and A. Colosimo, "Ppsi - a free software ptp implementation," in *2014 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS)*, 2014, pp. 71–76.

[32] V. Bode, *Potential Bug in PPSi*, private communication, E-Mail to Maintainer A. Wujek, Jan. 2024.

[33] J. Breuer. "PTPd Issue Tracker." (Apr. 2023), [Online]. Available: https://github.com/ptpd/ptpd/issues (visited on 04/10/2024).

[34] W. Owczarek, *ptpd(8) Precision Time Protocol Daemon User's Manual*, 2.3.1, Jun. 2015.

[35] I. Genibel and e. a. Berg Christoph. "Popularity Contest Statistics – Debian Quality Assurance." Packages: PTPd, LinuxPTP, Chrony. (2015), [Online]. Available: https://qa.debian.org/popcon.php (visited on 04/10/2024).

[36] "Welcome to The Linux PTP Project," Network Time Foundation. (Apr. 2024), [Online]. Available: https://www.linuxptp.org/ (visited on 04/10/2024).

[37] O. Obleukhov and A. Byagowi. "Simple Precision Time Protocol at Meta," Meta Platforms, Inc. (Feb. 2024), [Online]. Available: https://engineering.fb.com/2024/02/07/production-engineering/simple-precision-time-protocol-sptp-meta/ (visited on 04/05/2024).

[38] D. Arnold. "BMCA Deep Dive: Part 1," Meinberg Global. (Feb. 2022), [Online]. Available: https://blog.meinbergglobal.com/2022/02/01/bmca-deep-dive-part-1/ (visited on 04/03/2024).

[39] *Adjtimex(2) system calls manual*, Linux man-pages 6.03, Feb. 10, 2023. [Online]. Available: https://www.man7.org/linux/man-pages/man2/adjtimex.2.html (visited on 03/22/2024).

[40] D. T. Bui, A. Dupas, and M. Le Pallec, "Packet delay variation management for a better ieee1588v2 performance," in *2009 International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, 2009, pp. 1–6.

[41] S. Allan, "Planes will crash! things that leap seconds didn't, and did, cause.'," *Requirements for UTC and Civil Timekeeping on Earth. American Astronautical Society, Science and Technology Series*, vol. 115, 2013.

[42] C. Pascoe. "Time, technology and leaping seconds," Google Inc. (Sep. 2011), [Online]. Available: https://googleblog.blogspot.com/2011/09/time-technology-and-leaping-seconds.html (visited on 04/04/2024).

[43] M. Burnicki, "Technical aspects of leap second propagation and evaluation," in *Requirements for UTC and Civil Timekeeping on Earth Colloquium. Science and Technology Series*, vol. 115, 2015.

[44] O. Obleukhov and A. Byagowi. "It's time to leave the leap second in the past," Meta Platforms, Inc. (Jul. 2022), [Online]. Available: https://engineering.fb.com/2022/07/25/production-engineering/its-time-to-leave-the-leap-second-in-the-past/ (visited on 04/05/2024).

[45] "Resolution 4 of the 27th CGPM (2022) – On the use and future development of UTC," General Conference on Weights and Measures (CGPM). (2022), [Online]. Available: https://www.bipm.org/en/cgpm-2022/resolution-4 (visited on 04/05/2024).

[46] Raspberry Pi Foundation. "Raspberry Pi 4 and Raspberry Pi 5 Data Sheets." (2024), [Online]. Available: https://datasheets.raspberrypi.com/ (visited on 04/23/2024).

[47] *fake-hwclock(8) Fake Hardware Clock System Manager's Manual*, 0.11, Debian Distribution, Oct. 2014.