EtherTime: Cross-vendor Evaluation of PTP/NTP on Ethernet-based COTS Embedded Platforms

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

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

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

***Abstract*—We design and develop EtherTime, a tool for empirical evaluation of open-source implementations of the widely adopted Precision Time Protocol and Network Time Protocol standards. EtherTime is designed for distributed embedded systems networked over Ethernet. To demonstrate its benefits, we assemble a testbed of Raspberry Pi 4/5, Xilinx ZUBoard 1CG, and NVIDIA Jetson TK-1 boards, and carry out a measurement-based evaluation of four popular open-source implementations – PTPd, LinuxPTP, SPTP, and Chrony. EtherTime is successful in highlighting their limitations with regards to resource contention (specifically network and memory) and faults of various kinds (with/without hardware clock support). We open-source EtherTime and all datasets derived from this empirical study.**

I. Introduction

High-precision clock synchronization is a foundational requirement for a wide range of distributed and real-time systems. It enables applications such as accurate location estimation via Global Navigation Satellite Systems (GNNSs) [1], cryptographic certificate validation [2], seamless audio/video playback [3], and efficient profiling and coordination in distributed computing [4]. It also supports fault-tolerant architectures, such as those using double- or triple-modular lockstep redundancy, where tightly synchronized clocks ensure that all replicas execute in unison [5]. Without this coordination, safety-critical systems like those in avionics cannot reliably meet deadlines or maintain consistent behavior in the presence of faults, undermining their designed resilience.

Our goal is to examine how *software-based* clock synchronization performs in next-generation cyber-physical systems (CPS) built from commodity off-the-shelf (COTS) components. Unlike traditional high-assurance systems that rely on specialized hardware for synchronization [5], consumer-grade robots and unmanned vehicles favor low-cost, flexible solutions built on Ethernet and general-purpose hardware. To improve their reliability, designers may adopt fault-tolerant architectures inspired by high-assurance systems [6–9]. However, lacking access to custom synchronization hardware, such designs must depend on software clock synchronization protocols, such as Precision Time Protocol (PTP) [10] and Network Time Protocol (NTP) [11]. The effectiveness of fault tolerance in these contexts thus hinges on the reliability of the underlying synchronization. Yet little is known about how different implementations compare or how developers should choose among them, especially under adverse conditions common in embedded and industrial environments.

To address this gap, we present EtherTime, a tool for fair, automated, cross-vendor benchmarking of clock synchronization protocol implementations on Ethernet-based COTS platforms. Instead of introducing new high-accuracy measurement methods [12], EtherTime relies on established software-based offset measurements that trade some precision for easier deployment and greater portability. Its main focus is on testing the resilience of existing synchronization implementations; by injecting diverse failure scenarios, EtherTime enables users to automatically assess protocol behavior under CPU contention, network faults, and other challenging conditions.

We deploy EtherTime on a heterogeneous testbed consisting of Raspberry Pi 4/5, Xilinx ZUBoard 1CG, and NVIDIA Jetson TK-1 boards, evaluating four widely used open-source implementations of the NTP and PTP standards. Our study covers 265 distinct configurations and over 1000 profiling runs, uncovering practical limitations under stress and providing insights into robustness and scalability. We distill these findings into best-practice recommendations for configuring software-based synchronization, and open-source both EtherTime[1] and the complete dataset from our experiments[2].

II. Background

Among the many clock synchronization protocols explored over the years [10, 11, 14–19], PTP and NTP have emerged as widely adopted standards for local- and wide-area networks, respectively. Our focus is on embedded systems, where PTP is more relevant due to its higher accuracy and suitability for local synchronization. PTP operates with two roles: *master* and *slave*. The master periodically broadcasts its local time, and each slave estimates the clock offset by combining this with a propagation delay estimate obtained through a separate message exchange [20]. This offset is then used to synchronize the slave's local clock with the master. A key design consideration in PTP is how timestamps are captured. Software-based timestamping introduces variability due to delays in the OS and networking stack, whereas hardware timestamping enables more accurate synchronization by recording timestamps at the

[1]https://github.com/caps-tum/EtherTime
[2]https://github.com/caps-tum/EtherTime-Dataset

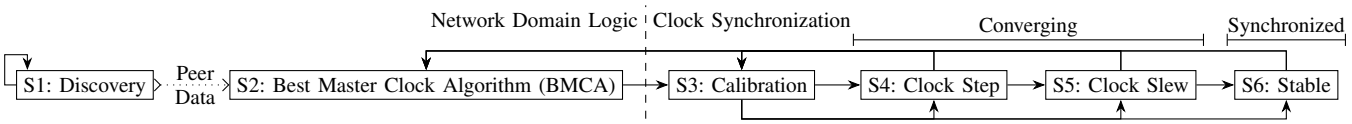


Fig. 1. Lifecycle of a PTP slave. S1 discovers remote clocks, typically via multicast announcements. S2 determines master–slave roles among peers using a standardized algorithm [13], based on, e.g., configured priorities and clock quality. S3 obtains coarse estimates of the clock offset. S4 attempts to directly overwrite the slave's local clock; this is generally restricted to initial synchronization, as it can violate the assumption that time flows forward continuously. S5 gradually slews the clock to reduce the remaining offset (e.g., Linux limits the rate to 500 parts per million. In S6, the local clock is made available to applications, although the slave may still transition back and resynchronize to earlier stages if conditions change.

network interface, just before transmission or upon reception [21]. Obtaining a global time reference at the master (e.g., UTC) is beyond PTP's scope and often unnecessary for many applications. When needed, however, such references can be sourced from a GNNS receiver, an atomic clock, or another PTP-synchronized domain.

Our goal is to understand what can reasonably be expected from current implementations of PTP, and under what conditions its assumptions and guarantees begin to break down. Out of the six stages in the lifecycle of a PTP slave (see Fig. 1), we focus on Stage S6, where the local clock signal is ready to be consumed by applications. While synchronization continues, this stage reflects the system's normal operating mode. However, since a slave can revert to earlier stages due to changes in network conditions, such as connectivity loss or large offset excursions, we also examine what causes such regressions, especially when behavior differs across implementations. To enable meaningful comparisons, we ensure that all PTP instances across vendors and platforms are evaluated in comparable synchronization states.

III. Related Work

Dedicated hardware solutions can provide high-precision synchronization with strong timing guarantees, but they often entail significant deployment and maintenance costs [22]. By contrast, protocols like PTP allow modern real-time systems to use commodity Ethernet or Wi-Fi for clock synchronization, lowering costs but also weakening timing guarantees. Hence, a systematic evaluation of these reliability–cost trade-offs is essential to inform production-scale deployment.

Several works have compared NTP and PTP theoretically and empirically, identifying key sources of uncertainty such as jitter and asymmetry [23, 24]. Other efforts have focused on fault tolerance, investigating failure modes that arise from aging hardware in long-lived deployments [25], discrepancies between hardware and software assumptions [26], and vulnerabilities in adversarial settings [27, 28]. Approaches to mitigate such risks include tracking multiple time sources to eliminate single points of failure [29, 30].

Researchers have also examined protocol limitations in specific domains. For instance, Mani et al. [31] demonstrated that existing synchronization protocols fail to meet the constraints of IoT platforms, and proposed algorithmic simplifications with hardware offloading. Others have studied PTP's deficiencies in industrial WANs [32], recommended unicast configurations for datacenter use [33], or attempted to emulate hardware timestamping over Wi-Fi [34]. However, many of these evaluations are restricted in scope, limited to simulations or a narrow range of implementations and configurations.

A broader issue across much of the literature is a lack of reproducibility. Studies are often limited to one or two implementations and rarely include open datasets or benchmarking frameworks [35–39]. For example, Schriegel et al. [12] note that most evaluations require manual setup, external hardware, or simulation-only models, none of which scale to cross-platform comparison. While cross-platform benchmarking tools exist for domains like MPI [40] and DDS [41] standards, no comparable infrastructure currently exists for clock synchronization protocols. The OpenClock framework [42] makes progress in this direction but requires adoption of its custom clock abstraction layer, limiting its general applicability to real-world systems.

Complementary efforts such as TrueTime [4] and Timeline [43] integrate clock uncertainty into application logic through simulation and API abstractions but do not support real-time synchronization via mainstream protocols like PTP. Likewise, TSN-based approaches can improve synchronization accuracy under network load [44], but remain non-trivial to deploy. Moreover, TSN only addresses timing accuracy across the network; much of the residual error originates from onboard effects, such as processing delays between the network interface and application software [45].

Security remains a critical but under-addressed aspect of clock synchronization. Many PTP implementations lack protections against clock manipulation, whether via denial-of-service, forced resets, or malicious skew [28]. While extensions have been proposed to enhance PTP's resilience [29, 46–49], standardization remains incomplete, and none of the implementations we evaluate offer native support. A systematic security evaluation lies outside the scope of this study.

Taken together, these studies highlight key limitations of existing synchronization protocols across domains and under various constraints. Yet, there remains a lack of reproducible, implementation-diverse evaluations that span platforms and configurations—especially in the embedded systems domain. Our work aims to address this gap through EtherTime.

IV. Motivation

We aim to design a distributed system for collecting and measuring data from various clock synchronization protocol implementations, with two primary goals: first, to enable the *automated* execution of complex, carefully orchestrated

experiments, such as fault-tolerance evaluations; and second, to support multiple protocol implementations concurrently in a *fair* and *comparable* manner, ensuring consistent conditions across experiments. Existing studies typically adopt one of two approaches. The first relies on ad hoc scripting, often combining implementation-reported accuracy metrics from PTP/NTP with hardware-based measurements. While common, this method demands substantial manual effort, is error-prone, and typically supports only a single implementation. The second approach employs commercial tools, e.g., Meinberg's Track Hound[3], Spirent Sentinel[4], Timebeat[5], and Microchip TimeMonitor[6], which, though powerful, are often tied to proprietary ecosystems and lack broader interoperability. A few open-source alternatives exist, such as NTPmon[7] and ntpperf[8], but these are largely focused on NTP and support only a narrow range of implementations with limited measurement capabilities. To enable more rigorous and comparable empirical analysis in future work, there is a clear need for more accessible, reliable, and flexible tools for collecting and analyzing synchronization data across diverse clock synchronization protocols.

Automation enables far more reliable data acquisition than manual experimentation. It can also manage the installation and configuration of different implementations, along with built-in data analysis, enabling end users to perform cross-vendor benchmarking with minimal effort. To ensure compatibility with a wide range of clock synchronization hardware and software, and to support seamless deployment across environments, the design avoids physical instrumentation methods. Techniques such as using oscilloscopes to measure circuit-level phase differences for estimating precise clock offsets [44, 50–53] are excluded, as they are invasive, require specialized equipment, and do not scale well. We also exclude modeling, simulation, and formal analytical techniques [12], which depend on tightly controlled environments, are restricted to specific hardware platforms, and often fail to capture differences between protocol implementations.

PTP already exposes internal accuracy metrics at runtime through the so-called *integrated measurement* approach [12], which offers a means to gauge synchronization performance. Hence, we can read out the offset estimate from the PTP implementation for every synchronization packet that PTP sends. However, simply installing an implementation and observing these reported metrics is insufficient. It is equally important to assess how reliable these protocols remain under conditions that degrade accuracy, such as resource contention or faults, including software or hardware resets. Our goal is to systematically explore the conditions under which clock synchronization protocols may fail, and to stress-test their robustness under adverse operational conditions.

[3]https://www.ptptrackhound.com
[4]https://www.spirent.com/assets/u/spirent-sentinel
[5]https://www.timebeat.app/
[6]https://www.microchip.com/en-us/product/TimeMonitor
[7]https://docs.ntpsec.org/latest/ntpmon.html
[8]https://github.com/mlichvar/ntpperf

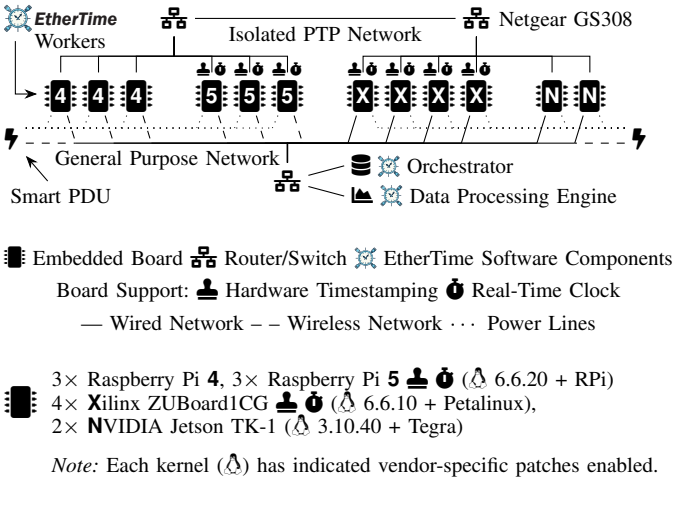


Fig. 2. Overview of the EtherTime system deployed on our testbed.

V. EtherTime

To realize our objectives, we develop EtherTime, implemented in Python and deployed on a distributed cluster of embedded devices that are connected over Ethernet (Fig. 2). Clock synchronization and orchestration/data collection operate over separate, isolated network links to avoid interference. The synchronization network uses Ethernet and connects devices through two cascaded Gigabit Ethernet switches.[9] The general-purpose network is for orchestration and data collection, and combines wired and wireless links,[10] though no time-critical packets are transmitted over WiFi. EtherTime comprises three main components: a central *orchestrator*, distributed *workers*, and a central *data processing* engine.

The *orchestrator* bootstraps the experimental environment, manages the cluster nodes and smart Power Delivery Units (PDUs), injects failures, and hosts the database that collects logs streamed from the boards. Its primary responsibility is to ensure that all experiments, including complex ones involving fault tolerance, can be conducted fully automatically.

Each node runs a *worker* process that installs and manages the corresponding clock synchronization clients, reconfiguring them as needed using predefined EtherTime configuration templates. This guarantees equivalent configurations across nodes, which is critical for fair comparisons. To ensure reproducible starting conditions, the worker performs an initial synchronization to the master clock with coarse precision (within 1 ms), then introduces a large, fixed offset (60 s) to the local clocks of the slave nodes. This procedure constrains the initial offset variance across trials and nodes to within $60\,\text{s} \pm 1\,\text{ms}$ (i.e., spread $< 0.002\%$), which leads to consistent synchronization behavior and convergence times

[9]Each Ethernet cable is no longer than 5 m, with signal propagation in copper at approximately 5 ns/m [54], resulting in a maximum of 25 ns delay per link and up to 75 ns total end-to-end link-layer delay.

[10]Due to physical constraints: for example, our smart power units support only WiFi, and Raspberry Pi boards have a single Ethernet port.

across experiments. Notably, the resulting offset is still large, representing a worst-case scenario that forces the PTP clients to perform full synchronization from a highly divergent state. In cases where clocks start closer together, synchronization would be correspondingly easier and faster. An added benefit of the initial sync phase is that it aligns software clock drift across nodes: Linux's drift compensation is calibrated during this phase and remains in effect for the main experiment, improving the reproducibility of initial conditions. The worker also collects system metrics and log outputs from various sources, which are transmitted to the orchestrator's database over the general-purpose network.

Finally, the *data processing engine* runs offline to convert raw logs into time series, generate aggregated profiles and summaries, and prepare the data for tabular and graphical display in the EtherTime web interface[11]. The dataset can be interactively queried through Django-based filters.

System clocks on the worker nodes cannot be trusted for globally timestamping logs, as they may be arbitrarily adjusted during synchronization and are subject to resets during fault-tolerance experiments. To maintain consistent chronological ordering across nodes, EtherTime must be resilient to such disruptions. Automatically reconstructing the global order of events after clock inconsistencies is notoriously difficult. EtherTime avoids this problem by streaming logs over the general-purpose network (separate from the Ethernet links used for clock synchronization) to a central database that timestamps each message upon arrival using a single, consistent clock source. This approach ensures a globally ordered timeline of events across the testbed. Importantly, the database clock is intentionally left unsynchronized with external time sources to guarantee monotonicity. Although this logging mechanism introduces a small latency, on the order of milliseconds, it is negligible relative to the duration of each experiment (typically 20+ minutes), especially since each worker commits logs at one-second intervals. An additional benefit is robustness: logging via the central database preserves consistency even during simulated node failures, avoiding the complexities of recovering from partially written logs had they been stored locally on each node.

Using EtherTime, we have collected, processed, and analyzed over 13 million log records to date. EtherTime is designed to be easily extensible to new clock synchronization protocols and implementations beyond those evaluated in this study. Adding support for a new vendor typically requires only three tasks: scripting an installer, defining a configuration template, and writing a log parser. We have streamlined deployment of EtherTime on new clusters by minimizing dependencies on specialized hardware and software. This allows users to focus on evaluating synchronization accuracy and reliability on their own platforms, rather than spending time building experimental infrastructure from scratch. Our aim is to enable users to assess baseline synchronization behavior with minimal manual effort, empowering system designers to generate their own measurements, tailored to their hardware and use case, rather than relying solely on prior results, which may be outdated, incomplete, or incompatible with their setups. By leveraging EtherTime, users can avoid ad hoc evaluation efforts and benefit from a standardized, reproducible methodology. The tool also helps reduce the risk of methodological or statistical errors, promotes integration of best practices, and lowers the barrier to entry for rigorous experimentation. Our entire study, including advanced experiments involving resource contention and fault tolerance, can be reproduced on different hardware or implementations with just a few EtherTime commands.

[11]See the Data Analysis section in https://github.com/caps-tum/EtherTime

VI. Protocols & Testbed

As a first step, we surveyed available PTP implementations and found that the number of viable, freely available options is limited. We excluded the following implementations from our evaluation: **(i)** *OpenPTP* [55] is currently unmaintained, with its last activity dating back more than a decade; it has since been commercialized. **(ii)** *Timebeat* [56] depends on heavyweight infrastructure (notably the Elasticsearch/Logstash/Kibana stack), making it unsuitable for embedded deployments. **(iii)** *PPSI* [57] suffers from stability issues, including buffer overruns; we have filed a corresponding bug report [58]. **(iv)** *White Rabbit* [18], an open extension of PTP for sub-nanosecond synchronization, requires highly specialized hardware, including vendor-specific synchronous Ethernet switches and NICs with *syntonization*[12] support, making it impractical for use with commodity embedded systems. **(v)** *Statime* [59], a Rust-based implementation in early development (alpha stage), was released only after our study concluded and is therefore not included in our evaluation.

This left us with four suitable options: PTPd, LinuxPTP, SPTP, and Chrony. **(vi)** *PTPd* [60] has seen limited maintenance in recent years [61] and lacks modern features such as hardware timestamping. Despite this, it has inspired several derivative implementations (including commercial) and continues to see deployment, as indicated by package trackers [62], likely due to its simplicity and broad support across non-Linux UNIX systems. **(vii)** *LinuxPTP* [63] is the most widely deployed open-source PTP solution in Debian. It is designed for robustness and tight integration with the Linux kernel, leveraging hardware support and kernel features to enhance synchronization accuracy. **(viii)** *SPTP* [64] is a lightweight, PTP-inspired protocol developed at Meta to address practical deployment challenges with standard PTP. It aims to achieve similar synchronization performance while reducing resource usage and improving resilience in large-scale datacenter environments. **(ix)** Finally, we include *Chrony* [65], the state-of-the-art implementation of the NTP protocol. Chrony is the most feature-rich of the evaluated solutions and, as a general-purpose clock synchronization protocol, serves as a useful baseline for comparison against the more specialized PTP-based implementations.

[12]Syntonization is a specific form of phase synchronization in which the frequencies of two clocks are matched, but not their absolute time values.

We initially deployed our hardware testbed using three types of embedded platforms (Fig. 2): the Raspberry Pi 4 [66], the Xilinx ZUBoard 1CG [67], and the NVIDIA Jetson TK-1 [68]. Partway through our study, the Raspberry Pi 5 [69] became available; we include it as a bonus case to explore the improvements offered by next-generation hardware.

The Raspberry Pi 5 differs from the Raspberry Pi 4 by offering hardware timestamping support on its network interface and an integrated battery-powered real-time clock (RTC) [70]. The RTC allows the board to retain accurate time across power cycles and may offer better long-term stability than chipset oscillators, which are typically more susceptible to thermal drift or manufacturing variation. Many embedded SoCs omit RTCs to reduce cost, instead relying on less stable integrated oscillators to maintain the system clock.

We also use a cluster of four Xilinx ZUBoard 1CG boards, which feature dual ARM Cortex-A53 and dual Cortex-R5F cores. These were adapted to run Debian for software consistency, based on Xilinx's 5.15 kernel with patched R815X drivers to support a secondary Ethernet adapter. However, this kernel version only supported hardware timestamping on the TX path, not RX, which impaired synchronization accuracy and rendered SPTP nonfunctional. We resolved this by upgrading to Xilinx kernel 6.6.10. The NVIDIA Jetson TK-1 boards, based on 32-bit ARMv7, were painstakingly updated to Ubuntu 22.04 LTS for compatibility, despite relying on NVIDIA's legacy 3.10.40 JetPack kernel. All other platforms in our testbed run Debian 12.

Our hardware selection spans a decade of commodity embedded systems, with release dates from 2014 to 2024. It covers both 32- and 64-bit architectures, platforms with and without hardware timestamping or RTCs, and diverse network interface vendors. By including Raspberry Pi, Jetson, and Xilinx boards, we capture performance data across the most widely used and best-selling single-board computer (SBC) families for general-purpose, AI/GPU-focused, and dataflow-oriented applications [71].

VII. Baseline Evaluation

The baseline evaluation for each vendor serves as a reference point for analyzing modified configurations. We use the default PTP/NTP profiles wherever applicable. For PTP, we retain all three standard frequency settings: synchronization messages from master to slave at 1 Hz, path delay measurements at 1 Hz, and master discovery announcements at 0.5 Hz. Chrony is similarly configured using its default NTP parameters. Increasing these frequencies does not significantly improve synchronization accuracy; in fact, it may degrade software-based measurement stability. For example, on the R-Pi 4, LinuxPTP achieves its best synchronization performance at sampling rates ≤ 2 Hz. At a much higher rate of 128 Hz, both the median and P_{95} clock offsets degrade by $7\times$ and $109\times$, respectively. This is likely due to increased queuing delays and jitter at higher packet rates. While these figures may not directly imply worse synchronization accuracy (as this ultimately depends on each implementation's offset compensation strategy and would require hardware-level validation), they show that *EtherTime cannot estimate high-quality offsets when the sampling frequency is set excessively high*.

To ensure predictable synchronization behavior, we configure PTP's second stage (S2) to deterministically select the designated master node. We achieve this by assigning a fixed value to the `grandmasterPriority1` parameter, a configurable field used by S2's Best Master Clock Algorithm [13] to influence master clock election. Doing so eliminates a potential source of variability in our measurements that would otherwise arise from dynamic master selection. Static master assignment is also common in practice, as it allows operators to retain control over the synchronization topology. Our configuration mirrors this setup to ensure consistency and repeatability.

Separating *converging* and *converged* stages. For all experiments, we collect approximately 1200 samples over 20-minute runs, repeating each setup multiple times for robustness. Our primary interest lies in analyzing clock offset statistics after synchronization has stabilized, i.e., during the stable stage (S6). However, unlike systems such as TrueTime [4], PTP offers no standardized mechanism for detecting this convergence across different implementations.

While PTP defines a set of endpoint states (`LISTENING`, `MASTER`, `PASSIVE`, `UNCALIBRATED`, `SLAVE`), these do not reflect synchronization quality; e.g., a client in the `SLAVE` state may still have a millisecond-scale offset that is gradually being corrected through clock slewing, eventually settling into a microsecond-scale equilibrium, without ever changing PTP state. Some implementations, like LinuxPTP, offer internal servo states (`s0`: initial, `s1`: intermediate, `s2`: locked and synchronized) providing finer-grained insight into synchronization progress. However, these are often not exposed externally, are non-standardized, and may vary even within the same implementation depending on the servo configuration.

EtherTime uses a simple and consistent method that works across all PTP implementations, regardless of vendor or internal design, relying solely on the sequence of observed offset values. During convergence (stages S4 and S5), the offset consistently shifts in one direction (either ahead or behind the master clock) and typically decreases in magnitude. As the system nears its optimal precision, offsets may overshoot and reverse direction, though they generally continue to shrink. In the stable state (S6), these offset estimates oscillate around a small range without sustained directional drift, indicating minor corrections rather than active convergence. The number of direction changes in the offset estimate is thus an indicator: under normal conditions, convergence rarely involves more than a few direction changes. We define convergence as having occurred after five such reversals (configurable if needed). This approach reliably distinguishes the converging phase (characterized by large corrections) from the stable phase (minor adjustments only). As shown in Fig. 3, this heuristic aligns well with visual inspection. We have manually verified hundreds of traces without encountering anomalies.

Beyond separating the convergence and stable phases, this

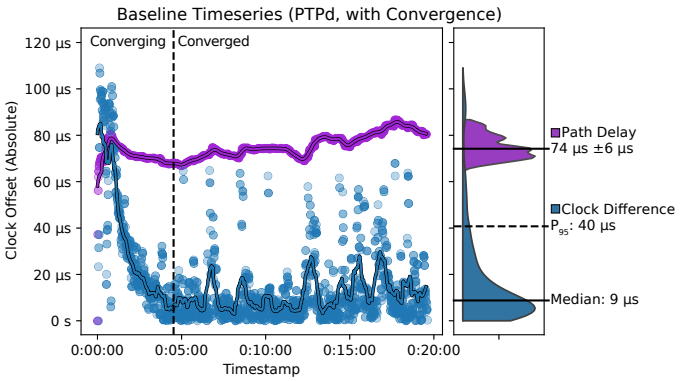


Fig. 3. A sample run of PTPd in its default configuration (left: scattered raw signal and denoised moving average, right: kernel density estimates) on the R-Pi 4. We use offset direction changes to predict when PTP transitions to the *stable* stage S6 (denoted using a dashed line). The clock offset is much lower than the path delay. This is true for all vendors across all platforms and shows the sophistication of the path delay compensation techniques used by clock synchronization protocols compared to a naive approach.

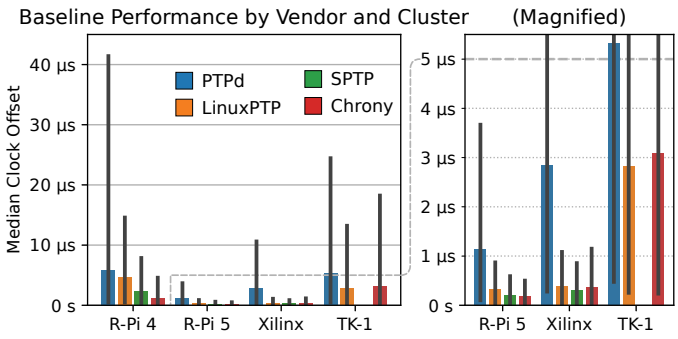


Fig. 4. Median baseline performance for all vendors, across all four platforms (left) and a magnified view for R-Pi 5, Xilinx, and Jetson TK-1 platforms (right). Error bars denote P_5 and P_{95} percentile values, respectively.

TABLE I
VENDOR- AND PLATFORM-SPECIFIC BASELINE CLOCK OFFSETS

System + Vendor		P_{50}	P_{95}	P_{99}	Max	
R-Pi 4	PTPd	5.9	41.4	64.0	153.7	µs
	LinuxPTP	4.7	14.6	19.5	70.9	µs
	SPTP	2.4	7.9	11.6	**25.8**	µs
	Chrony	**1.2**	**4.6**	**6.9**	157.8	µs
R-Pi 5	PTPd	1.1	3.7	4.9	13.2	µs
	LinuxPTP	0.3	0.9	1.1	28.2	µs
	SPTP	0.2	0.6	**0.8**	**8.9**	µs
	Chrony	**0.2**	**0.5**	6.1	9.2	µs
Xilinx	PTPd	2.8	10.6	15.2	28.0	µs
	LinuxPTP	0.4	1.1	1.4	17.8	µs
	SPTP	**0.3**	**0.9**	**1.1**	**1.7**	µs
	Chrony	0.4	1.1	1.6	10.8	µs
TK-1	PTPd	5.3	24.4	37.3	232.2	µs
	LinuxPTP	**2.8**	**13.2**	**20.3**	**141.8**	µs
	SPTP*	-	-	-	-	µs
	Chrony	3.1	18.2	30.8	167.9	µs

*SPTP could not run on TK-1 even after patching it for 32-bit compatibility, since the Tegra 3.10 kernel has no support for socket options needed by SPTP.

indicator allows us to measure time to convergence, i.e., the duration a node should wait after connecting to a master before serving time to applications. *PTPd is by far the slowest vendor to establish stable synchronization across all boards*. For example, correcting a 28 ms offset via PTPd requires roughly 13 minutes of clock slewing, whereas other implementations handle larger offsets in under a minute.

Metrics. As shown in Fig. 3, the observed clock offset estimates are inherently noisy, making it difficult to discern the true clock offset and its fluctuations. To address this, we use the median of the observed offset values as a representative estimate of the true offset. Wherever possible, we also include 95$^{\text{th}}$ percentile error bars or bands to capture the range of higher deviation, offering a more conservative view of synchronization accuracy. The path delay is computed by the PTP implementation and, in the absence of hardware timestamping, reflects delays across the full software stack.

Vendor- and platform-specific baselines. Fig. 4 illustrates the clock synchronization performance of all four vendors across the four hardware platforms, with detailed values listed in Table I. *PTPd stands out as a clear outlier, consistently exhibiting the worst synchronization offset across all platforms.* In the best case, PTPd shows a median clock offset that is 88% worse than the best-performing vendor on TK-1 boards (though all baselines perform relatively poorly on this platform). In the worst case, PTPd's median offset is 860% worse than the best-performing vendor on Xilinx boards.

While the differences among the remaining vendors are less pronounced, Chrony, SPTP, and LinuxPTP achieve the best synchronization accuracy on the Raspberry Pi, Xilinx, and TK-1 platforms, respectively. *It is particularly noteworthy that Chrony, an NTP client, can match, or even outperform, all evaluated PTP clients, despite PTP being designed specifically for high-precision synchronization.* Chrony represents the state of the art in NTP implementations. It benefits from hardware timestamping support comparable to that used by PTP clients and offers advanced features such as *falseticker* detection [72]. Although PTP is theoretically capable of surpassing NTP in precision, especially when used with transparent and boundary clocks that compensate for queuing delays in PTP-aware network hardware, such hardware remains impractical for embedded deployments. Specialized PTP-compatible switches typically carry four-figure price tags, making them unsuitable for cost-sensitive environments. To reflect realistic industrial and embedded use cases, we restrict our evaluation to standard Ethernet switches, which are far more commonly deployed.

Among all factors, the choice of hardware has the most pronounced impact on synchronization quality. Both R-Pi 5 and Xilinx boards support hardware timestamping, whereas R-Pi 4 and TK-1 boards do not. As a result, R-Pi 5 shows marked improvements over R-Pi 4, with median offset reductions ranging from 5× for PTPd to 14× for LinuxPTP. Similarly, the Xilinx board achieves better synchronization than the TK-1 board, improving from 2× for PTPd to 8× for Chrony. Interestingly, PTPd benefits from running on newer hardware, despite lacking hardware timestamping support, highlighting that hardware timestamps are not the sole determinant of synchronization performance. They help reduce timing variability but are *not* a complete solution on their own.

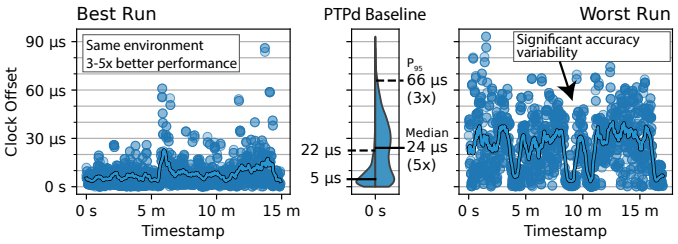


Fig. 5. Timeseries profiles for the best run and the worst run of PTPd on R-Pi 4 under identical conditions, showing significant differences in noise levels.

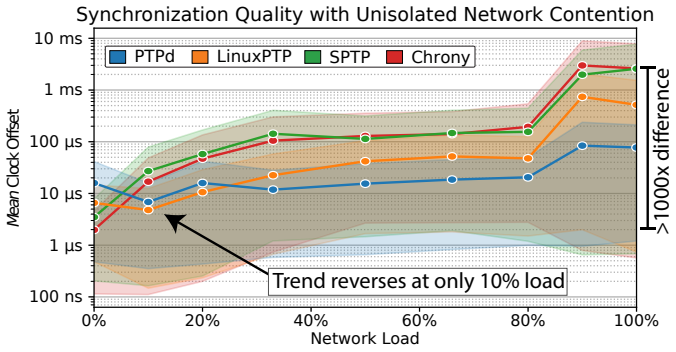


Fig. 6. Mean (P_5, P_{95}) clock synchronization accuracy with varying network interference on R-Pi 4. Unlike other graphs in the paper, we show mean values rather than median to highlight the significance of increasing outliers.

Hardware timestamping also reduces synchronization variance. Without it, the difference between the median and P_{95} offset ranges from 3× for LinuxPTP on R-Pi 4, to 7× for PTPd on R-Pi 4. This can be seen in the relatively tall error bars in Fig. 4 (left) for R-Pi 4 and TK-1 boards. With hardware timestamping, the difference drops to 3.7× for PTPd on Xilinx and 2.7× for LinuxPTP on R-Pi 5. *Thus, hardware timestamping not only improves average synchronization accuracy, but also significantly reduces the magnitude of outliers.*

Reproducibility. Despite careful setup, there is considerable noise within each run. The maximum observed clock offset typically ranges from only 2× of the median for SPTP on Xilinx to 34× for Chrony on R-Pi 4. Discrepancies can also occur between runs, even when the configuration remains unchanged. Fig. 5 illustrates this effect across multiple runs for different vendors on the R-Pi 4 platform. Among all vendors, PTPd shows the highest variance in both median and P_{95} offset values. Restarting the PTPd client can cause the median offset to jump from 5 µs to 24 µs, a 400% increase, that persists across the entire 20-minute run. In contrast, LinuxPTP yields more stable results: its median offset ranges only from 4 µs to 6 µs (42% difference). Chrony produces the most consistent results with a run-to-run variation of just 0.2 µs (12%).

We employ several mitigation strategies to mitigate noise. To reduce the impact of uncontrolled environmental factors (e.g., room temperature), we interleave measurements across PTP implementations, i.e., PTPd, LinuxPTP, SPTP, Chrony, PTPd, and so on, rather than executing all repetitions of one implementation before moving to the next. This ensures that any drift in external conditions affects all implementations equally, helping to avoid systematic bias. Each baseline test is repeated at least 15 times per vendor and platform, totaling approximately 40 hours of runtime and ∼144000 samples per benchmark. Between runs, the entire cluster is rebooted to eliminate residual state and ensure independence of observations. Other than that, the setup remains untouched, so any variation stems solely from internal protocol behavior. We also apply filters to discard runs with too many missing values, insufficient sample counts, or failures in basic consistency checks. The resulting baseline data is highly complete: fewer than 0.1% of samples are missing for any vendor or platform.

VIII. Network Contention

Network contention can arise when applications generate high volumes of traffic, such as video streams in surveillance systems, model updates in distributed AI, or high-frequency sensor data in industrial monitoring. In embedded systems connected via Ethernet, this kind of sustained or bursty load is common and can lead to queuing delays and jitter. Clock synchronization protocols must therefore remain robust and maintain accuracy, even under such network pressure.

We use iPerf [73] to introduce controlled network load, varying it between 0% and 100% of the nominal Gigabit Ethernet bandwidth (1000 Mbps). The master board runs the iPerf server, while the client board connects to it in bidirectional mode, transmitting and receiving traffic at the target bandwidth.[13] Fig. 6 shows the mean clock offset observed on R-Pi 4. Since the default configurations of all PTP implementations lack mechanisms to prioritize synchronization traffic, synchronization accuracy deteriorates with increasing network load. PTPd exhibits the smallest degradation: its mean clock offset increases by only 4.7×, from 11 µs to 51 µs at 100% load. In contrast, Chrony shows the worst degradation, with its mean clock offset increasing by 800×, from 1.7 µs to 1360 µs. Both Chrony and SPTP exhibit large P_{95} values under load (Fig. 6), which impacts their average offset even at just 10% load. We attribute this to their use of unicast message exchanges, in contrast to PTPd and LinuxPTP, which rely more on multicast. (Chrony's multicast support is disabled by default.) *Interestingly, PTPd, despite underperforming in baseline conditions, is the most resilient to network contention, maintaining significantly smaller clock offsets under heavy load (above 30%) compared to the others.*

Network load impacts clock synchronization accuracy primarily because synchronization depends on both the magnitude and variability of path delay, both of which tend to

[13]Due to inconsistencies in how different versions of `iPerf2` handle certain flags, and contradictory information in its documentation [73], it was initially unclear whether iPerf was generating TCP or UDP traffic in our experiments. Because this distinction affects traffic characteristics, we audited our open-source dataset (which includes all logs, including iPerf outputs) and confirmed that all traffic generated during our measurements was TCP.

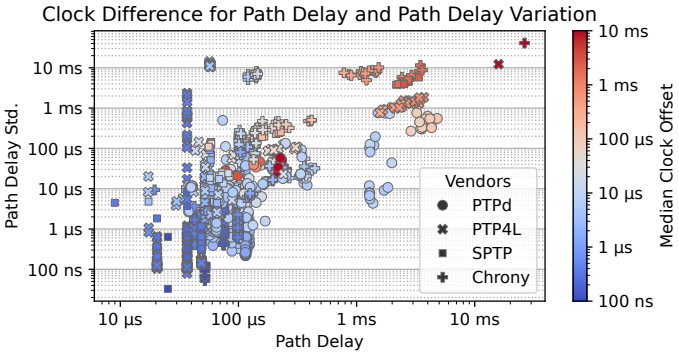


Fig. 7. Synchronization accuracy (shown on a color scale) with respect to path delay (x axis) and its variance (y axis), for all four vendors across all four platforms and under various tested circumstances.

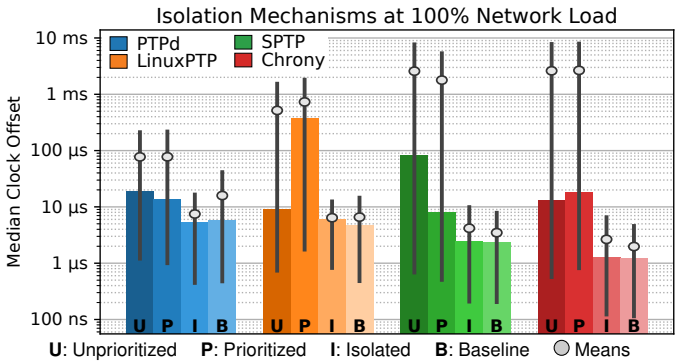


Fig. 8. Different possibilities of isolating network load, versus the baseline with no load (medians, with means for comparison of skew).

increase under load due to longer queuing times in hardware and software. To examine the relationship between increased path delay and degraded synchronization, we aggregate all 20-minute measurement profiles and analyze them based on path delay and path delay variation.[14] The results in Fig. 7 show that *high path delay and high path delay variation combined correlate with poor synchronization accuracy, but neither metric alone is a sufficient predictor*. Indeed, we observe some instances of good synchronization despite high path delay or high delay variation alone. However, the reverse is not necessarily true: poor synchronization does not always coincide with elevated path delay or variation. This is visible in the presence of a few poorly synchronized (red) data points within a cluster of better-performing (blue) points. Such cases arise from other factors independent of path delay, such as fault reconvergence, which we explore further in §X.

Two principal strategies exist to mitigate the impact of network interference on clock synchronization: prioritizing synchronization traffic in software or hardware, and physically isolating it by assigning it a dedicated network interface. While physical isolation may seem costly, especially for embedded systems, it is already common in industrial and datacenter environments, where a secondary management interface is often used to separate control traffic from application data. To emulate this setup on Raspberry Pi boards, we configure the routing table so that only clock synchronization traffic is routed through the dedicated (isolated) interface, while all other traffic uses a general-purpose interface. Fig. 8 illustrates this comparison, showing the unisolated default setup on the left and the baseline setup with no load on the right. *Physical isolation completely mitigates the negative impact of network load on synchronization accuracy.* Although cross-talk through the software network stack is theoretically possible, its effect appears negligible in practice.

[14]Under high network load, synchronization sometimes fails entirely due to repeated transmission timeouts. For instance, 26% of 20-minute runs with 100% load on R-Pi 4 failed to synchronize altogether; mostly with LinuxPTP, while Chrony did not fail at all. On R-Pi 5, LinuxPTP was consistently unable to synchronize across all trials. These failure cases are difficult to include in statistical analysis but are critical to highlight: complete loss of synchronization, especially without detection, is the worst-case outcome.

Traffic prioritization, however, does not provide the same level of isolation as physical separation. It is implemented using the Differentiated Services Code Point (DSCP) mechanism [74], which requires consistent support across all networking components, both software and hardware, along the communication path. In our case, although both the switch and operating system report DSCP support, this alone was insufficient for complete traffic segregation. For example, DSCP prioritization improved the median clock offset for SPTP by up to 10× compared to the unprioritized case, yet the average offset remained high due to persistent outliers. An anomaly was observed with LinuxPTP, where the median clock offset worsened (i.e., increased) by a factor of 45× with DSCP enabled, even though P_5 and P_{95} remained within a 3× deviation from their unprioritized baselines. We attribute this discrepancy to differences in queuing behavior across platforms. On R-Pi 4, Raspbian's default queuing policies are not tuned for PTP.[15] *We therefore caution that DSCP-based software prioritization must be explicitly tuned and thoroughly validated—merely assigning a DSCP value and trusting the OS may have counterproductive effects.*

IX. Other Resource Contention

Embedded systems often operate with limited compute resources. With edge ML workloads becoming increasingly common, applications frequently stress processing capabilities, placing additional pressure on the scheduler, pressure that clock synchronization must withstand. *Our data shows that contention for resources other than the network rarely causes synchronization degradation of comparable severity.*

CPU contention, for example, results in only modest degradation. On R-Pi 4, the worst observed median offset increase under CPU load is just 22% and 37% on P_{95} (Chrony). This limited interference stems from the fact that PTP clients

[15]Linux's queuing configuration varies by platform. On R-Pi 4, five hardware queues are available for transmission and two for reception, both with uniform priority mappings. However, software prioritization is enforced only within individual queues, not across them, leading to unpredictable packet scheduling and degraded performance. In contrast, R-Pi 5 uses a single hardware queue, making prioritization fully dependent on software, which leads to more consistent queuing behavior.

consume little processing time and are promptly scheduled by Linux's default scheduler, even under heavy load. As a result, real-time schedulers like `SCHED_FIFO` or `SCHED_RR` are not strictly necessary for robust PTP performance.

Interestingly, light CPU load can even improve synchronization. Reduced power-saving behavior under moderate load lowers scheduling jitter, leading to better performance. For instance, PTPd's median and P_{95} show improvements of 17% and 53%, respectively, on R-Pi 4. On R-Pi 5, Chrony, LinuxPTP, and SPTP remain mostly unaffected by CPU load (with a worst-case increase of only 17%). In contrast, PTPd shows noticeable degradation of up to 71%, still much less than under network contention. This suggests that tuning the Linux performance governor may improve synchronization, although at the cost of increased power consumption.

We further stress-tested clients using Stress-NG [75] to simulate cache and memory contention from applications such as video/image processing, AI, and communications. As clock synchronization is not data-intensive, we expect lower impact. Cache contention caused moderate degradation in some cases (up to 8.9× with PTPd on R-Pi 5), while memory bandwidth contention had a similar impact (also up to 9×, with the same vendor/platform). Finally, stressing time-related kernel resources, such as timers, alarms, or the use of cyclic tasks with `SCHED_DEADLINE`, had negligible effect.

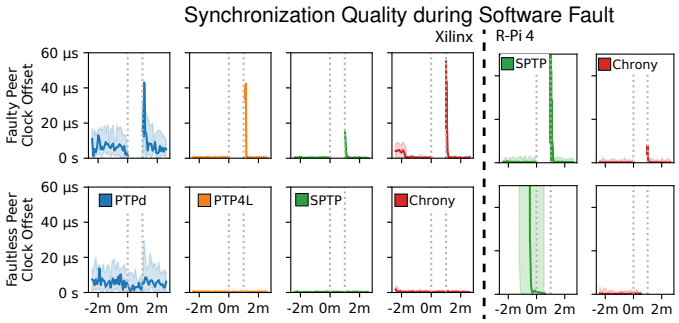
X. Fault Tolerance

We examine faults in both the PTP software and the underlying node hardware. Faults on the master node are particularly critical, as they can affect the entire synchronization domain, whereas faults on a slave are typically contained to a single node. To account for high-reliability deployments, we also evaluate scenarios in which a backup client (failover) assumes the role of master. Due to space constraints, we omit network fault injection results, they are available in our dataset.[16]

Software fault in slave. We emulate software faults by sending a `SIGKILL` signal to forcefully terminate the PTP client on the slave. In practice, such faults may result from bugs, out-of-memory conditions, or transient hardware issues such as bit-flips [25]. While the client is down, the system clock continues to drift, at a rate determined by the underlying hardware clock drift, offset by the last correction applied during synchronization. If synchronization was stable before the crash, the resulting drift is expected to be minimal. In the worst-case scenario, such as during the early convergence phase, PTP may apply maximum-rate clock slew, typically up to 0.05%. This can result in software-induced drift of 500 µs per second of downtime, or 30 ms per minute, on top of any hardware drift. We aim to empirically assess the actual drift during such faults when the clock was previously stable.

We find that the maximum observed offset after a 1-minute software fault across 10 trials is 60 µs for Chrony (Fig. 9, top).

[16]Ironically, a full network outage is often easier for PTP to handle than scenarios involving heavy congestion or partial node failure. Since no state is lost, servos remain in holdover mode, and PTP can simply reconverge on the common time once the network reconnects.



*SPTP: Failed packet timestamp: no TX timestamp found after 500 ms. Chrony: Can't synchronize: no selectable sources.

Fig. 9. Software crash induced 1-minute fault (dotted lines) on Xilinx (left), with a faulty slave (top) and a second control slave (bottom). After the fault, an increased offset can be observed as the clocks are resynchronized, but there is little service disruption. R-Pi 4 (right, for comparison) has issues on the control slave: for SPTP and Chrony, the second slave experiences large clock steps or entire service outage. Since clock drift occurs randomly, we superimpose 10 trials for each vendor to illustrate variability.

This is approximately 40× worse than the vendor's median baseline performance, but well below the theoretical bound. Interestingly, chance plays a role: in some trials, the observed offset remained low despite the full minute of downtime, e.g., just 16 µs in the best case for SPTP. In all cases, *all vendors rapidly reconverge on the clock signal within a few seconds of restarting*, often aided by quirks in the PTP protocol.[17]

In high-reliability deployments, faults on one slave must not propagate or affect the synchronization quality of other slaves. For example, in a smart access control system, repeated failures of a specific key to authenticate should not impair the ability of other keys to unlock the secured data. To evaluate this, we connect a second control node to the same master (Fig. 9, bottom). On Xilinx nodes, this setup poses no issue: the faulty slave resynchronizes correctly, and the control slave remains unaffected. However, the R-Pi 4 exhibits problematic behavior with both SPTP and Chrony. For SPTP, we observe temporary disconnects caused by repeated errors in multiple trials (see the note* under Fig. 9). Chrony shows outages at exactly the same times. reproducible to within one second. The visual differences arise because SPTP converges more slowly, placing most of its trajectory out of view. *Therefore, we cannot assume that a node failure is always contained; it may impact others in the synchronization domain.*[18]

Hardware fault in slave. A hardware fault on the slave is more disruptive than a software fault: not only is the

[17]PTP slaves request synchronization signal leases from the master with predetermined expiration times. When a slave crashes, the master continues sending synchronization messages to the now-defunct peer (a behavior known as *abandoned sync* [33]). Once the slave restarts and obtains a new lease, this overlap in message flow accelerates resynchronization by increasing the frequency of received sync messages.

[18]A notable caveat with PTPd is that multiple software faults, even if spaced apart, can lead to compounding failures. Specifically, the third PTPd fault consistently causes the network interface to fail, regardless of platform or trigger method. Restarting the network or reloading the NIC driver does not resolve the issue; only a full system reboot restores functionality. This can disrupt all network-dependent applications, highlighting the need for caution when deploying PTPd or any derivatives that may share this bug.

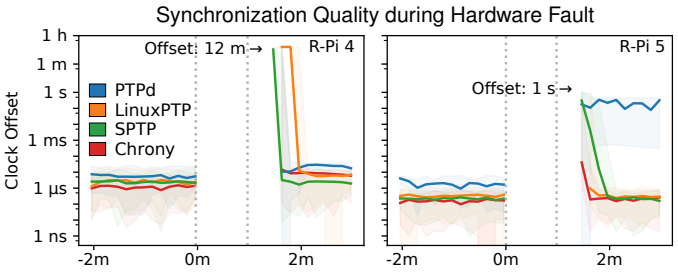


Fig. 10. A hardware fault on the slave of the R-Pi 4 cluster (left) and the R-Pi 5 cluster (right). Both slaves need to fully resynchronize after rebooting, but the R-Pi 5 has an advantage due to its hardware clock.

clock synchronization state lost, but also the kernel state, including the current system time and clock drift. As a result, clients must fully reconverge following such a fault, using the full step-and-slew approach as in the baseline. We simulate hardware faults using programmable PDUs, which emulate complete power loss. However, since Xilinx and TK-1 platforms cannot automatically power on after shutdown, our evaluation of hardware faults is restricted primarily to the Raspberry Pi platform. While hardware faults in production can result from various component failures, many of them effectively present the same symptoms as a full power loss.

Because the R-Pi 4 (Fig. 10, left) includes only a timestamp counter (TSC) and lacks a RTC, the system time after reboot defaults to either the last time persisted to disk (if `fake-hwclock` [76] is active) or a fallback value such as the UNIX epoch. On Raspbian (Debian-based), where `fake-hwclock` is enabled by default, we observe a temporary deviation of 12 min (on SPTP). This large offset is initially corrected via a clock step (which breaks the continuity of time), followed by a reconvergence phase lasting several minutes until previous accuracy levels are restored. In contrast, the R-Pi 5 (Fig. 10, right) is equipped with a RTC, which preserves the system time even while powered off. While RTCs generally operate at lower resolutions (commonly 32,768 Hz) compared to the internal TSC, they still provide stability in the parts-per-million range [77]. As a result, the maximum observed offset is only 1290 ms (on LinuxPTP), which is approximately 3 orders of magnitude smaller. This level of deviation can theoretically be corrected via maximum clock slew without breaking monotonicity or continuity within 33 min. Only PTPd avoids the initial clock step (as per its default profile), since the calibration offset remains below one second, choosing stability at the cost of slower reconvergence. *A simple RTC can thus effectively mitigate the impact of hardware faults. However, in deployments where an RTC is not feasible (e.g., due to cost), we recommend introducing a delay before relaunching applications to allow sufficient time for resynchronization.*

Hardware fault in master. A failure on the master inevitably leads to inconsistencies in the announced time, particularly problematic in embedded scenarios where no external clock source is available to serve as ground truth. On the R-Pi 4 (Fig. 11, center left), we observe that *when the master restarts with a different reference time, a large offset re-emerges between the master and the slaves*, resembling the behavior seen during slave-side faults (Fig. 11, left). This issue, which manifests as indefinite clock slew, can be addressed by reconfiguring the slave to permit clock steps after startup (disabled by default for safety). However, this comes with trade-offs: allowing large clock corrections risks breaking both monotonic and continuous time flow, and the system must be thoroughly tested for resilience. Our experiments with stress-test tools and real applications show that many applications misbehave during a clock step, even when using Linux's monotonic clock, highlighting that resilience cannot be assumed based solely on the clock API in use. As noted earlier, the issue is better mitigated by RTCs, which preserve system time during power loss (e.g., R-Pi 5). Importantly, the RTC does not need to be of particularly high quality to maintain sub-second accuracy over short downtimes. If constraints permit installing an external clock on only one node, it is most beneficial to equip the master, as it can distribute its stable time base to all connected slaves.

Hardware fault in master, with failover. For both LinuxPTP and Chrony, the designated failover master can assume control rapidly in the event of a master failure, resulting in virtually no disruption to the timing service (Fig. 12). In contrast, PTPd is unable to complete the failover successfully, instead reporting an error: *No active masters present, resetting port*. However, even with successful failover in LinuxPTP and Chrony, the underlying issue persists: if the original master restarts with an incorrect reference time, it may disseminate this erroneous time, which other clients will accept, leading to offset spikes and clock corrections of familiar magnitudes: 71 min on the R-Pi 4 and 530 ms on the R-Pi 5. We also observe that LinuxPTP is more prone to reconvergence timeouts in these scenarios compared to previous tests. Because stage S2 (BMCA) selects the grandmaster deterministically (unless explicit changes are made to the configuration to alter priority), the restarted original master may retake control, even if its clock is incorrect. In such cases, the failover master is unable to propagate the correct time to the newly rejoined master. To address this, *the failed master should either be permanently disabled, reconfigured with a lower priority (demoting it to a slave), or equipped with an external clock source to ensure that a power fault does not alter its reference time*. Finally, we note that changes in network topology (e.g., a master rejoining the network) are not always detected immediately. All three PTP implementations may exhibit delayed reactions to system state changes, such as when the original master attempts to take back control from the failover master.

XI. Resource Consumption

The resource footprint of clock synchronization should be kept minimal, ideally to the point of being negligible. For instance, a battery-powered sensor device collecting health data may become ineffective if its ability to store, process, or transmit data, or to operate over extended periods, is impaired due to PTP consuming excessive memory, CPU, network

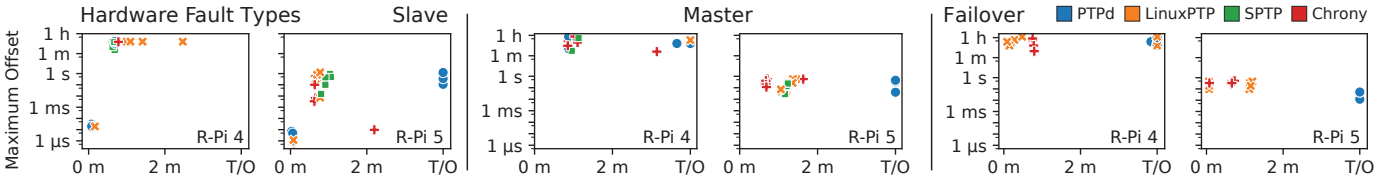


Fig. 11. The maximum clock offset observed and the time taken to reconverge to normal accuracy for different failures; timeout (T/O) is 4 minutes. Due to no RTC, Raspberry Pi 4 observes high offsets sometimes indefinitely. Raspberry Pi 5 has better resilience, but still slow reconvergence.

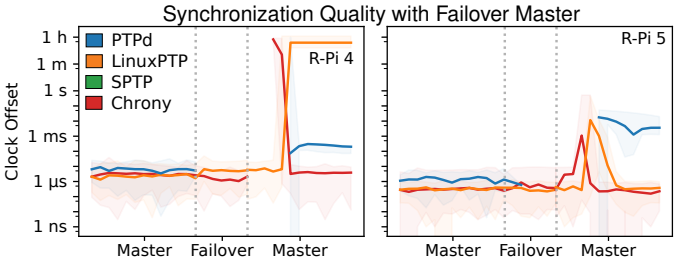


Fig. 12. Fault tolerance with a failover master. While the failover master works almost seamlessly (except for PTPd), problems arise when the original master eventually reboots and starts announcing the wrong time. We exclude SPTP from this evaluation, as masters and slaves run different binaries and dynamically switching roles is unsupported.

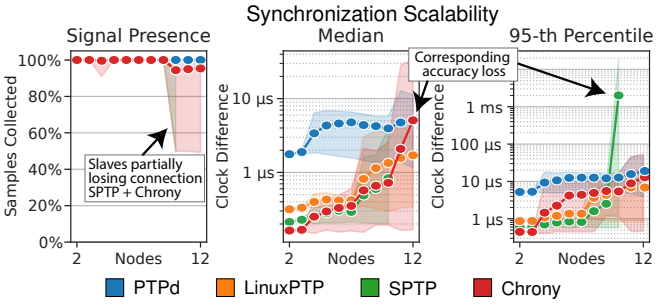


Fig. 13. Synchronization quality for increasing numbers of nodes, with slave synchronization percentages (left), clock difference median (center) and P_{95} (right) averaged across nodes (error band: best/worst node's value). As the number of nodes increases, some slaves consistently lose the synchronization signal, noticeably diminishing accuracy.

bandwidth, or power. In fact, SPTP was specifically designed with the goal of reducing resource consumption [33].

On low-capability devices, ROM/flash space is typically constrained, making binary footprint a critical concern. To evaluate it, we strip packages of documentation and omit dependencies that are likely already present (e.g., lib-c), but include all other required dependencies in our measurements. *PTPd and LinuxPTP are suitable for microcontroller-class systems, with total sizes (executables + data + dependencies) of 840 KB and 970 KB, respectively, whereas SPTP (21 MB) and Chrony (12 MB) are more than an order of magnitude larger*[19]. Closely related is memory usage, which also limits which boards can support a given PTP implementation. LinuxPTP uses approximately 250–400 KB, Chrony 800 KB–1 MB, and PTPd around 1 MB in unique and resident set sizes, all of which fit within a typical ∼2–4 MB memory budget. In contrast, *SPTP demands significantly more memory, starting at 8-10 MB for the master and 15-16 MB for the slave. Moreover, SPTP's memory usage grows considerably faster than the others*, increasing by roughly 240 KB for each additional slave. Notably, SPTP allocates large amounts of virtual memory—3 GB for the master and 1 GB for the slave—exceeding the addressable space of 16-bit platforms and rendering it unsuitable for deployment on microcontrollers without large virtual memory support or overcommit capabilities. Given these ROM and RAM requirements, the more lightweight implementations (LinuxPTP, PTPd, and to some extent Chrony) are viable candidates for deployment on 16 MB flash controllers such as the STM32MP1 or NXP RT1052. However, sufficient headroom must be reserved for the main application.

Power consumption is a concern for mobile or battery-powered deployments. Using consumed CPU time as a proxy for energy usage, we observe that LinuxPTP and PTPd require the least compute, while Chrony uses 11%–180% more. *SPTP exhibits the highest CPU usage*, consuming between 140% and 390% times more CPU than LinuxPTP or PTPd. This increase in compute correlates with higher thermal output: the master node runs on average 1.4 °C hotter, based on onboard temperature sensors, indicative of greater power draw.

Network bandwidth is another concern, particularly in low-throughput or cost-sensitive environments. While PTP is often advertised as having negligible bandwidth usage [78], our measurements show that, although the differences are modest, *SPTP still generates more traffic than other implementations*, averaging approximately 1.1 MB and 9 K packets per hour per client. Chrony is the most efficient in terms of network usage, but the advantage over LinuxPTP and PTPd, which are nearly identical due to implementing the same protocol, is small and unlikely to have significant practical impact.

XII. Scalability

To evaluate scalability, we incrementally increase the number of nodes (Fig. 13), scaling heterogeneously from two nodes up to the full cluster. Nodes are added in order of decreasing capability: the R-Pi 5 acts as the master, while slaves are added in the following order: R-Pi 5, Xilinx, R-Pi 4, and TK-1.

[19]SPTP's footprint can be reduced by 40% if only the master or slave executable is deployed. Chrony's size is also significantly reduced if core dependencies, such as *iproute2*, *libgnutls30*, and *tzdata* (totaling roughly 10 MB), are already available on the system.

With each additional node, the median clock offset increases, a trend most pronounced with Chrony. Notably, the first TK-1 board added experiences significantly worse synchronization than the second, with a degradation factor of approximately 4×. One R-Pi 4 board also frequently disconnects, showing a drop in signal presence to roughly 50%, a problem not observed on the other boards. Despite these issues, *boards using software timestamping still benefit from the R-Pi 5 master's hardware timestamping*, particularly the R-Pi 4 boards ($n \in [8, 10]$), which achieve synchronization performance substantially better than the baseline.

A distinct spike in offset is observed for SPTP at $n = 10$, driven by incompatibility with the TK-1 boards. The root cause appears to be the same R-Pi 4 node showing anomalous behavior under Chrony, accompanied by a similar drop in signal presence. Log analysis reveals timestamping timeouts, suggesting that capacity limits may be emerging. However, this board consistently performs worse than other R-Pi 4 boards despite identical software configuration. Such discrepancies in supposedly identical hardware can be attributed to manufacturing variances or aging effects [79], which are known to impact synchronization quality and stability, though we can only speculate that this is the cause here. This anomaly also results in the worst scalability being observed with Chrony, whose synchronization quality degrades by a factor of 5000×.

LinuxPTP consumes on-board resources most efficiently, matches others in network traffic usage, and significantly outperforms the only other lightweight vendor, PTPd, in synchronization quality. Despite SPTP's promise of improved resource efficiency in datacenter-scale deployments with up to 100K clients, we find that it requires substantially more ROM, RAM, and compute resources, making it less suitable for embedded environments than alternative PTP implementations. This discrepancy arises from SPTP's design focus on server-class systems, with limited consideration for embedded platforms or legacy hardware, such as the lack of support for 32-bit architectures and older kernels. Its implementation in Go introduces further overhead in terms of dependencies, runtime environment, and memory footprint [80], whereas traditional C-based implementations, particularly LinuxPTP, offer a more lightweight, performant, and accurate solution for resource-constrained deployments.

XIII. Learnings and Conclusion

Choice of vendor. PTPd, while simple and mature, suffers from serious drawbacks, e.g., it can soft-brick the network driver and is often an order of magnitude slower in clock (re-)convergence after faults. SPTP, developed for datacenter environments, claims resource efficiency but fails to deliver on embedded hardware. It has limited board support, lacks maturity, and incurs significant ROM, RAM, and compute overhead due to its Go runtime. This leaves LinuxPTP and Chrony. Chrony surprisingly matches or outperforms LinuxPTP in synchronization on 3 of 4 platforms, but its higher ROM/RAM footprint and occasional connectivity issues under load make it less suitable for constrained deployments. LinuxPTP, though efficient, may suffer from misbehaving traffic prioritization and failover-induced convergence stalls. Both are mature; suitability depends on deployment specifics.

Guarding against resource contention. Network congestion is the dominant cause of degradation. The ideal mitigation is physical separation of PTP and application traffic via a dedicated network—often too costly for embedded setups. Software prioritization (e.g., DSCP) is a fallback, but not foolproof: it can reduce synchronization quality and fails to eliminate outliers. Results depend on DSCP settings, NIC queues, and hardware behavior, so configurations must be tested carefully. Default unisolated setups are discouraged. Other shared resources, like memory bandwidth and CPU cache, play minor roles in synchronization accuracy.

Resilience against faults. An external time source is more critical than hardware timestamping. We strongly recommend using at least a real-time clock as opposed to relying solely on internal oscillators, ideally on both master and slaves, to avoid large time drifts after node faults. In isolated environments, PTP must be configured carefully, as default profiles may ignore even large time deviations. Failovers alone won't correct such divergence.

Conclusion. The main usability challenge lies in the *observability* and *verifiability* of synchronization. While all vendors support advanced features, correct configuration, especially for fault tolerance, is left to the user. This is not a common skill among engineers or IT operators, and defaults are neither safe nor broadly applicable. Efforts toward plug-and-play configurations could improve accessibility. Determining when and where synchronization fails also requires dedicated monitoring and testing, an often-missing component in open-source implementations. EtherTime addresses this gap by automating testing and monitoring in a cross-platform setup and offering configuration guidance. We aim to extend it to evaluate PTP security (still under standardization) and explore hardware probe integration for even higher measurement fidelity, at the cost of deployment simplicity. To enable broader adoption, future implementations should offer guided configuration, resilient default behavior, and accessible synchronization state. Until then, the promise of tightly synchronized distributed real-time algorithms remains aspirational, limited by current infrastructure's fragility and lack of guarantees.

XIV. Acknowledgements

This work was supported by NSERC Discovery and RTI grants. We thank the Visiting International Research Students program at UBC and Prof. Martin Schulz (TU Munich) for making it possible for Vincent Bode (PhD student, TU Munich) to conduct a research internship at UBC. We also acknowledge the Science Undergraduate Research Experience program from the UBC Faculty of Science for supporting William Shen (undergraduate student, UBC Computer Science) during his summer internship on this project. Finally, we thank all our reviewers for their insightful comments. OpenAI ChatGPT was used to assist with language editing, based on a complete draft written by the authors.

References

[1] R. B. Langley et al. “Introduction to GNSS”. In: *Global Navigation Satellite Systems*. 2017.

[2] M. E. Acer et al. “Where the Wild Warnings Are: Root Causes of Chrome HTTPS Certificate Errors”. In: *SIGSAC 2017*.

[3] R. Steinmetz. “Synchronization Properties in Multimedia Systems”. In: *IEEE Journal on Selected Areas in Communications* 8.3 (1990).

[4] J. C. Corbett et al. “Spanner: Google’s Globally Distributed Database”. In: *ACM TOCS* 31.3 (2013).

[5] A. L. Hopkins et al. “FTMP—A Highly Reliable Fault-tolerant Multiprocess for Aircraft”. In: *IEEE* 66.10 (1978).

[6] A. Gujarati et al. “Real-time Replica Consistency over Ethernet with Reliability Bounds”. In: *RTAS 2020*.

[7] A. Gujarati et al. “In-ConcReTeS: Interactive Consistency meets Distributed Real-Time Systems, Again!” In: *RTSS 2022*.

[8] N. Gandhi et al. “REBOUND: Defending Distributed Systems Against Attacks with Bounded-time Recovery”. In: *EuroSys 2021*.

[9] A. Loveless et al. “IGOR: Accelerating Byzantine Fault Tolerance for Real-time Systems with Eager Execution”. In: *RTAS 2021*.

[10] J. C. Eidson et al. “IEEE-1588™ Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems”. In: *PTTI 2002*.

[11] D. Mills et al. *RFC 5905: Network Time Protocol Version 4: Protocol and Algorithms Specification*. 2010.

[12] S. Schriegel et al. “Investigation in Automatic Determination of Time Synchronization Accuracy of PTP Networks with the Objective of Plug-and-Work”. In: *ISPCS 2014*.

[13] D. Arnold. *BMCA Deep Dive: Part 1*. Meinberg Funkuhren GmbH. 2022. URL: https://blog.meinbergglobal.com/2022/02/01/bmca-deep-dive-part-1/.

[14] D. Mills. *RFC 4330: Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI*. 2006.

[15] R. Cochran et al. “Design and Implementation of a PTP Clock Infrastructure for the Linux Kernel”. In: *ISPCS 2010*.

[16] M. Maróti et al. “The Flooding Time Synchronization Protocol”. In: *ICENSS 2004*.

[17] C. Lenzen et al. “PulseSync: An Efficient and Scalable Clock Synchronization Protocol”. In: *IEEE/ACM Transactions on Networking* 23.3 (2014).

[18] M. Lipiński et al. “White Rabbit: PTP Application for Robust Sub-nanosecond Synchronization”. In: *ISPCS 2011*.

[19] F. Gong et al. “CESP: A Low-power High-accuracy Time Synchronization Protocol”. In: *IEEE Transactions on Vehicular Technology* 65.4 (2015).

[20] D. T. Bui et al. “Packet Delay Variation Management for a Better IEEE1588V2 Performance”. In: *ISPCS 2009*.

[21] R. Exel et al. “Asymmetry Mitigation in IEEE 802.3 Ethernet for High-Accuracy Clock Synchronization”. In: *IEEE Transactions on Instrumentation and Measurement* 63.3 (2014).

[22] A. Flammini et al. “Clock Synchronization of Distributed, Real-time, Industrial Data Acquisition Systems”. In: *Data Acquisition* (2010).

[23] T. Neagoe et al. “NTP versus PTP in Computer Networks Clock Synchronization”. In: *ISIE 2006*.

[24] S. J. Wissow. “Time Enough: Synchronization for Latency Measurement”. PhD thesis. University of New Hampshire, 2020.

[25] B. Ferencz et al. “Effects of Runtime Failures in IEEE 1588 Clock Networks”. In: *I2MTC 2017*.

[26] P. Ramanathan et al. “Fault-tolerant Clock Synchronization in Distributed Systems”. In: *Computer* 23.10 (1990).

[27] A. Nasrullah et al. “Trusted Timing Services with TimeGuard”. In: *RTAS 2024*.

[28] W. Alghamdi et al. “An Analysis of Internal Attacks on PTP-based Time Synchronization Networks”. In: *NUI Galway* (2022).

[29] S. Shi et al. “MS-PTP: Protecting Network Timing from Byzantine Attacks”. In: *WiSec 2023*.

[30] N. Kerö et al. “How to Effectively Enhance PTP Redundancy Using Dual Ports”. In: *SMPTE 2023*.

[31] S. K. Mani et al. “A System for Clock Synchronization in an Internet of Things”. In: *arXiv preprint arXiv:1806.02474* (2018).

[32] D. Fontanelli et al. “Accurate Time Synchronization in PTP-based Industrial Networks with Long Linear Paths”. In: *ISPCS 2010*.

[33] O. Obleukhov et al. “Simple Precision Time Protocol (SPTP)”. In: *ISPCS 2023*.

[34] P. Chen et al. “Understanding Precision Time Protocol in Today’s Wi-Fi Networks: A Measurement Study”. In: *ATC 2021*.

[35] C. Andrich et al. “Measurement of Drift and Jitter of Network Synchronized Distributed Clocks”. In: *IFCS-ISAF 2020*.

[36] E. Kyriakakis et al. “Hardware Assisted Clock Synchronization with the IEEE 1588-2008 Precision Time Protocol”. In: *RTNS 2018*.

[37] B. Ferencz et al. “Hardware Assisted COTS IEEE 1588 Solution for x86 Linux and its Performance Evaluation”. In: *ISPCS 2013*.

[38] A. Kern et al. “Accuracy of Ethernet AVB Time Synchronization under Varying Temperature Conditions for Automotive Networks”. In: *DAC 2011*.

[39] M. Lévesque et al. “A Survey of Clock Synchronization Over Packet-Switched Networks”. In: *IEEE Communications Surveys and Tutorials* 18.4 (2016).

[40] R. Reussner et al. “SKaMPI: A Comprehensive Benchmark for Public Benchmarking of MPI”. In: *Scientific Programming* 10.1 (2002).

[41] V. Bode et al. “Systematic Analysis of DDS Implementations”. In: *Middleware 2023*.

[42] F. M. Anwar et al. “OpenClock: A Testbed for Clock Synchronization Research”. In: *ISPCS 2018*.

[43] F. Anwar et al. “Timeline: An Operating System Abstraction for Time-Aware Applications”. In: *RTSS 2016*.

[44] L. Schürmann et al. *Implementation and Evaluation of Time Synchronization Mechanisms for Generic Embedded Systems for Time Sensitive Networking (TSN)*. Tech. rep. University of Stuttgart, 2021.

[45] J. Coleman et al. “Emerging COTS Architecture Support for Real-time TSN Ethernet”. In: *SAC 2019*.

[46] F. Rezabek et al. “PTP Security Measures and their Impact on Synchronization Accuracy”. In: CNSM. ’22.

[47] A. Finkenzeller et al. “PTPsec: Securing the Precision Time Protocol Against Time Delay Attacks Using Cyclic Path Asymmetry Analysis”. In: *INFOCOM 2024*.

[48] W. Alghamdi et al. “A Security Enhancement of the Precision Time Protocol Using a Trusted Supervisor Node”. In: *Sensors* 22.9 (2022).

[49] F. M. Anwar et al. “Applications and Challenges in Securing Time”. In: *CSET 2019*.

[50] A. Libri et al. “Evaluation of Synchronization Protocols for Fine-grain HPC Sensor Data Time-stamping and Collection”. In: *HPCS 2016*.

[51] H. Shaygan. “Time Synchronization for Large Volume Metrology in Industrial Networks”. MA thesis. Lappeenranta–Lahti University of Technology, 2023.

[52] E. H. Langemeijer. “Clock Synchronization For Radio Interferometry”. MA thesis. Eindhoven University of Technology, 2024.

[53] D. Ingram et al. “Assessment of Real-time Networks and Timing for Process Bus Applications”. In: *SEAPAC 2013*.

[54] B. Briscoe et al. “Reducing Internet Latency: A Survey of Techniques and Their Merits”. In: *IEEE Communications Surveys and Tutorials* 18.3 (2016).

[55] Stefan Tauner. *OpenPTP - Precision Time Protocol Implementation*. 2012. URL: https://github.com/stefanct/openptp.

[56] *Timebeat Installation Overview*. Timebeat. 2022. URL: support.timebeat.app/hc/en-gb/articles/360021334279-Timebeat-Installation-Before-you-begin.

[57] P. Fezzardi et al. “PPSi - A Free Software PTP Implementation”. In: *ISPCS 2014*.

[58] V. Bode. *Potential Bug in PPSi*. Private Communication. E-Mail to Maintainer A. Wujek. Jan. 2024.

[59] *Statime: Implementation of the Precision Time Protocol (PTP) in Rust*. 2024. URL: https://github.com/pendulum-project/statime.

[60] W. Owczarek. *ptpd(8) Precision Time Protocol Daemon User’s Manual*. 2.3.1. 2015.

[61] J. Breuer. *PTPd Issue Tracker*. 2023. URL: https://github.com/ptpd/ptpd/issues.

[62] I. Genibel et al. *Popularity Contest Statistics – Debian Quality Assurance*. Packages: PTPd, LinuxPTP, Chrony. 2015. URL: https://qa.debian.org/popcon.php.

[63] *Welcome to The Linux PTP Project*. Network Time Foundation. 2024. URL: https://www.linuxptp.org/.

[64] O. Obleukhov et al. *Simple Precision Time Protocol at Meta*. Meta Platforms, Inc. 2024. URL: https://engineering.fb.com/2024/02/07/production-engineering/simple-precision-time-protocol-sptp-meta/.

[65] A. E. Dinar et al. “NTP Server Clock Adjustment with Chrony”. In: *ICCCIOT 2020*.

[66] *Raspberry Pi 4 Spec*. 2024. URL: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/.

[67] *Avnet ZUBoard 1CG Development Board*. 2024. URL: https://www.xilinx.com/products/boards-and-kits/1-1pusyun.html.

[68] *Nvidia Jetson TK1: The World’s First Embedded Supercomputer*. 2024. URL: https://www.nvidia.com/content/tegra/automotive/pdf/jetson-tk1-brochure-web.pdf.

[69] *Raspberry Pi 5*. 2024. URL: https://www.raspberrypi.com/products/raspberry-pi-5/.

[70] *Raspberry Pi 4 and Raspberry Pi 5 Data Sheets*. 2024. URL: https://datasheets.raspberrypi.com/.

[71] S. J. Johnston et al. “Commodity Single Board Computer Clusters and their Applications”. In: *Future Generation Computer Systems* 89 (2018).

[72] R. Curnow et al. *chrony.conf(5) Chrony Configuration User’s Manual*. 4.4. 2023. URL: https://chrony-project.org/doc/4.4/chrony.conf.html.

[73] J. Dugan et al. *iPerf - The ultimate speed test tool for TCP, UDP and SCTP*. 2022. URL: https://iperf.fr/.

[74] K. Nichols et al. *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*. Request for Comments: 2474. 1998.

[75] C. I. King. *stress-ng (Stress Next Generation) - A Tool to Load and Stress a Computer System User’s Manual*. 0.15.06. 2023. URL: https://github.com/ColinIanKing/stress-ng.

[76] *fake-hwclock(8) Fake Hardware Clock System Manager’s Manual*. 0.11. Debian Distribution. 2014.

[77] H. Marouani et al. “Internal Clock Drift Estimation in Computer Clusters”. In: *Journal of Computer Systems, Networks, and Communications* (2008).

[78] D. Arnold. *Five Minute Facts About Packet Timing*. Meinberg Funkuhren GmbH. 2023. URL: https://blog.meinbergglobal.com/2013/10/28/one-step-two-step/.

[79] H. A. Abdelhafez et al. “Snowflakes at the Edge: A Study of Variability among NVIDIA Jetson AGX Xavier Boards”. In: *EdgeSys 2021*.

[80] D. Lion et al. “Investigating Managed Language Runtime Performance: Why JavaScript and Python are 8x and 29x slower than C++, yet Java and Go can be Faster?” In: *ATC 2022*.