Application-Level Evaluation of IEEE 802.1AS Synchronized Time and Linux for Distributed Real-Time Systems

HECTOR PEREZ TIJERO, Universidad de Cantabria, Santander, Spain
J. JAVIER GUTIÉRREZ GARCÍA, Universidad de Cantabria, Santander, Spain
DIEGO GARCÍA PRIETO, Universidad de Cantabria, Santander, Spain

The use of Ethernet and Linux is becoming common in industrial applications, even for those with real-time requirements, although neither of them were originally designed for this purpose. The emergence of Industry 4.0 (also known as Industrial Internet of Things, IIoT) has encouraged the evolution of these technologies to better handle real-time issues. On the one hand, Linux now supports mechanisms to configure certain real-time parameters, as well as core isolation and interrupt allocation facilities in multicore processors. On the other hand, the set of Ethernet standards IEEE 802.1 Time-Sensitive Networking (TSN) includes a high precision clock synchronization protocol (IEEE 802.1AS). The purpose of this work is to outline an execution framework for distributed systems based on TSN and Linux, which allows the execution of time-aware applications. We have studied and evaluated different configurations available for the proposed execution framework. In particular, a detailed characterization of the clock synchronization mechanism, from the application point of view, has been performed. Some conclusions about the current real-time capabilities of these technologies are also presented.

CCS Concepts: • **Computer systems organization → Real-time systems**;

Additional Key Words and Phrases: Clock synchronization, distributed real-time systems, IEEE 802.1AS, Linux, TSN, Industry 4.0

ACM Reference Format:
Hector Perez Tijero, J. Javier Gutiérrez García, and Diego García Prieto. 2024. Application-Level Evaluation of IEEE 802.1AS Synchronized Time and Linux for Distributed Real-Time Systems. *ACM Trans. Embedd. Comput. Syst.* 24, 1, Article 13 (December 2024), 16 pages. https://doi.org/10.1145/3701300

1 Introduction

Nowadays, there is a growing interest in using **Commercial Off-The-Shelf (COTS)** components within the real-time industrial domain. This trend is mainly driven by the need to minimize development costs and time to market along with the increase of non-functional requirements of real-time applications. In this context, the main benefits of using an operating system such as Linux become apparent: It provides comprehensive support for processors and peripherals,

This work was partially supported by MCIN/ AEI /10.13039/501100011033/ FEDER "Una manera de hacer Europa" under grants TIN2017-86520-C3-3-R (PRECON-I4) and PID2021-124502OB-C42 (PRESECREL).
Authors' Contact Information: Héctor Pérez Tijero, Universidad de Cantabria, Santander, Cantabria, Spain; e-mail: perezh@unican.es; J. Javier Gutiérrez García, Universidad de Cantabria, Santander, Cantabria, Spain; e-mail: gutierjj@unican.es; Diego García Prieto, Universidad de Cantabria, Santander, Cantabria, Spain; e-mail: diego.garciap@alumnos.unican.es.

ACM 1539-9087/2024/12-ART13
https://doi.org/10.1145/3701300

facilities for customization and high availability of open-source and proprietary software, including device drivers and development tools. Although the Linux kernel prioritizes throughput over determinism, its real-time capabilities have been improved over the past years [1]. Two of the most notable approaches include (1) the PREEMPT_RT kernel patch [2], which reduces the latency by making most of the kernel preemptable; and (2) the CPU isolation capabilities [3, 4], which aim to isolate a core in a multicore processor to execute specific tasks. Furthermore, there are already some previous experiences where real-time applications have been executed over Linux, such as in the case of **High-Performance Computing (HPC)** applications [1].

A recent survey [5] conducted on 120 industry practitioners in the field of real-time embedded systems showed that Linux is present in 55.88% of respondents' applications. Most of the applications analyzed employed multiple operating systems, and the combination of Linux and a **Real-Time Operating System (RTOS)** was used by 42% of respondents. These values show a relevant presence of Linux in industry and motivates our work to extend its usability to the kind of collaborative applications envisaged by Industry 4.0 [6, 7], where **Time Sensitive Networking (TSN)** protocols are key to build distributed application with real-time requirements [8]. For instance, high clock synchronization accuracy can be considered a key requirement in power and smart grid applications [9]. This accuracy requirement is also present in factory assembly lines where the coordination of robotic arms and specific industrial machinery requires communication and control with timing requirements below 1 millisecond.

The set of TSN standards defines IEEE 802.1AS [10] for precise time synchronization of devices on Ethernet networks, thus introducing the notion of a *global clock* in a distributed system. IEEE 802.1AS can be considered a profile of the IEEE 1588 standard [11], which defines the basis of a protocol for synchronizing the clocks of a distributed system known as **Precision Time Protocol (PTP)**. IEEE 1588 is a more general standard that can be applied to various domains beyond TSN-enabled Ethernet networks.

The fact of synchronizing precisely the clocks of a distributed system enables the development of time-aware applications that require global timestamping (e.g., sensor data [8]) or runtime checking of end-to-end deadlines. Furthermore, it can also play a significant role in kernel scheduling services for distributed real-time systems that aim to leverage global scheduling policies such as **EDF (Earliest Deadline First)**. Compared to more common policies based on static scheduling [12, 13, 14], EDF schedulers can optimize processors utilization when a global clock is available [15, 16], thus increasing the overall schedulability of the distributed system.

This article outlines a feasible execution framework designed for industrial distributed real-time applications in which nodes requires accurate time synchronization and some parts of the system rely on a Linux-based OS. The key contributions of this work include:

- Identification and analysis of different features within the Linux kernel that can be used to execute applications with real-time requirements while minimizing the interactions with other non-real-time applications.
- Characterization of the 802.1AS clock synchronization mechanism from applications' perspective, which includes estimating the overheads, the latencies, and the effective synchronization of the global clock at the application level.
- A set of recommended settings for both the Linux kernel and the 802.1AS clock synchronization protocol.

Unlike other works that are focused on evaluating the internals of the clock synchronization protocol closer to the physical layer (granularity, drift, PHY jitter, etc.) [17, 18], our work deals with measurements at the software layer (operating system and application). These measurements

will provide us with insights on the practical applicability of the proposed execution framework in industrial applications.

The document is organized as follows: Section 2 introduces the concepts of TSN that are relevant for this work, and it also describes the related work. The execution framework proposed in this article together with the available options and configuration details are presented in Section 3. Section 4 presents the tests used to measure clock overheads, event-handling latencies, and an estimation of the real synchronization between nodes. The evaluation of the global clock for the different configurations considered is presented in Section 5. Finally, we draw out our conclusions and possible future work in Section 6.

2 Background

2.1 Background on TSN

TSN is a set of standards that defines mechanisms for robust time-sensitive transmission of data over deterministic Ethernet networks. It was developed by the TSN task group of the 802.1 working group, and its target is to provide deterministic services through IEEE 802 networks, i.e., guaranteed packet transport with bounded latency, bounded jitter, and zero congestion loss.

One key functionality of TSN networks is provided by IEEE 802.1AS [10], which focuses on clock synchronization over bridged local area networks. Among other functionalities, it includes (1) the transport of synchronized time, (2) the selection of a timing source, and (3) the indication of timing errors. This standard ensures that the jitter and time synchronization requirements are met for time-sensitive applications, such as audio and video, as long as specific TSN hardware is present in **network cards (NICs)** and switches involved in the distributed real-time system (e.g., hardware for timestamping incoming and outgoing network packets).

The synchronization protocol proposed by 802.1AS is called **gPTP (generalized PTP)**, which can be considered a constrained profile of IEEE 1588 (widely known as PTP) [11] in which new timing features are also added. Networking devices such as *end-stations* (computing nodes) or *bridges* (switches) that are compliant with IEEE 802.1AS are referred to as time-aware systems by the specification.

This protocol generates a master-slave hierarchy among the clocks of the network. One clock is used as a reference time source (i.e., **GrandMaster (GM)**) and the remaining clocks are used as slaves. The choice of GM is usually performed through the **Best Master Clock Algorithm (BMCA)**. Once the GM is selected, the slaves adjust their clocks to the time broadcasted by the GM. The synchronization procedure is done periodically to refresh the slave clock and avoid clock drift over time. If the current GM is removed or a new better one is added to the network, then BMCA will automatically select a new GM again. According to IEEE 802.1AS, both switches and computing nodes can be selected as GM. Finally, other alternatives to BCMA can be applied such as the External Port Configuration Mechanism [10].

2.2 Related Work

Nowadays, there is a significant amount of research work related to IEEE 802.1AS focused on estimating the clock accuracy at the physical layer. For instance, the authors in Reference [19] provide a preliminary performance evaluation based on simulations and initial hardware implementations. Their evaluation included networks up to 7 hops (8 nodes) between master and slaves, and it also considered the influence of background traffic. The results show that a synchronization requirement of 500 ns can be easily met on 1 Gbit/s Ethernet links. A significant amount of research is currently focused on evaluating the accuracy of simulations. The works in References [20] and [21] showed that simulations generally obtain lower latencies compared to TSN hardware and propose several enhancements to available network models. Similarly, Reference [22] addressed

the creation of digital replicas able to accurately model the behavior of time synchronization and traffic shaping in TSN systems.

Another common approach to validate TSN systems is through experimental frameworks that rely on real hardware. For instance, the TSN-FlexTest framework [23] obtained sub-microsecond precision for time synchronization in the presence of cross network traffic. Similar results were obtained by Reference [24] with a custom hardware-assisted implementation of the PTP protocol.

In the context of automotive systems, the performance of IEEE 802.1AS was evaluated using hardware timestamping [25]. On average, the synchronization offset between master and slaves was 5 μs (with a maximum of 8 μs). Other research work [17] analyzed IEEE 802.1AS in large-scale networks. Their simulations showed a synchronization precision of around 2 μs for the last time-aware system in a chain of 100 hops. Sub-microsecond precision was obtained using a **Wide Area Network (WAN)** in smart grid applications [26]. The authors in Reference [27] ran a set of TSN-related experiments in a cloud-based environment and obtained synchronization offsets below 6 μs. In the industrial automation domain, Reference [28] explored the use of OPC UA distribution facilities over TSN networks.

The use of IEEE 802.1AS in wireless networks is also a notable field of research. The work in Reference [29] outlines the general issues of deploying clock synchronization protocols to wireless networks. The combination of TSN and popular broadband technologies (Wi-Fi, 4G, and 5G) is studied in Reference [30]. The authors in Reference [31] explored hybrid TSN networks (wired–wireless). Finally, Reference [32] described the requirements and challenges for time synchronization in different wireless use cases.

Reference [33] presents a method for practical performance evaluation of PTP clock synchronization within a distributed environment consisting of nodes running the QNX real-time operating system. Similarly to our work, the authors in Reference [34] presented an approach to synchronize the Linux system clock to the PTP clock, as well as the estimation of latencies through *cyclictest* [35]. In their opinion, having a synchronization accuracy of 100 μs or lower should be sufficient for a wide range of applications. Using IEEE 802.1AS for virtualized distributed real-time systems is an interesting approach presented in Reference [36], where global time data is read by virtual machines via shared memory. According to their measurements, the virtual machine with direct access to the network hardware can achieve sub-microsecond synchronization precision. Furthermore, the authors in Reference [37] outline a distributed real-time system based on Linux, but using specialized hardware and focusing on the scheduling of the network traffic through IEEE 802.1Qbv. Our work complements these efforts by proposing a different execution framework based on native Linux with the configuration of its isolation capabilities and evaluating how the global time is managed at the application level.

3 Real-time Distributed Execution Framework

3.1 Overview

The Linux kernel is attracting more and more adopters in the industry [5] due to its broad platform support, customization facilities, and high availability of software, including device drivers and development tools. Although Linux is geared towards general-purpose computing, the real-time performance has steadily improved over the past few years by means of different mechanisms such as the isolation of one or more cores for executing specialized applications or making the kernel fully preemptible [38].

Figure 1 illustrates the proposed software execution framework for a multicore system. Under this execution framework, real-time applications are executed as a standard Linux process, and they can rely on different Linux facilities, such as networking or timing services. Isolating a core

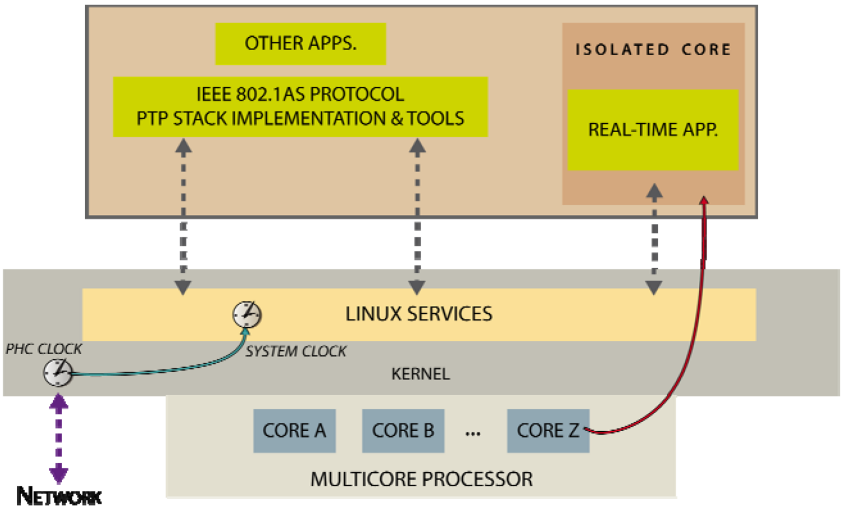

Fig. 1. The execution framework.

Table 1. Configuration Flags for PTP Tools

Tool	Flag	Purpose
ptp4l	*i*	NIC interface to use
ptp4l	*f*	Configuration file used
ptp4l	*step_threshold*	Change clock frequency instead of stepping the clock
phc2sys	*a*	Timing source from ptp4l
phc2sys	*r*	Synchronize the system clock
phc2sys	*transportSpecific*	Select between the IEEE 1588 and the IEEE 802.1AS domains

can enhance the predictability of the real-time application by limiting user and kernel preemptions. Modern kernels provide different mechanisms to facilitate the execution of threads in isolated cores, which will be presented and evaluated in subsequent sections. Furthermore, most of the interrupts should be also removed from the isolated core to reduce possible sources of interferences, although a few of them are bounded to each core and cannot be migrated.

The proposed execution framework provides the notion of global time by means of the implementation of the IEEE 802.1AS protocol [10], which is responsible for synchronizing all the **PTP Hardware Clocks (PHC)** for each node belonging to the distributed system.

Modern Linux kernels can execute applications with some kind of real-time requirements as long as the kernel can be configured accordingly [1]. To this end, the recommended settings for the proposed platform are detailed next.

3.2 Configuration Details

The platform configuration requires tuning both Linux compilation and runtime parameters, as well as configuring the PTP service, which are described next.

3.2.1 PTP Configuration. The Linux PTP Project implements the PTP standard for Linux, including the gPTP profile proposed by IEEE 802.1AS. Linux PTP provides two relevant tools whose main configuration options are shown in Table 1:

— The *ptp4l* tool is responsible for implementing the synchronization protocol. Besides using the default gPTP profile, this tool is also configured to provide monotonically increasing times (i.e., clock corrections are performed by changing the clock frequency).
— The *phc2sys* tool enables the synchronization of two local clocks in the computing node [34]. For instance, it has been used to synchronize the system clock to the PHC clock provided by the NIC. This increases the portability of the approach, and it also allows using some POSIX calls such as *clock_nanosleep* that cannot directly rely on the PHC clock device.

The platform relies on version 3.0 of Linux PTP. To minimize interruptions on clocks' synchronization, both ptp4l and phc2sys are executed as real-time processes and they are scheduled under the *SCHED_FIFO* real-time policy.

3.2.2 Setting up the Isolation of Cores. By executing a real-time process in an isolated core, the competition with the rest of the system workload is avoided and thus interferences may be minimized. For our purposes, the platform requires allocating one or more cores to execute the real-time application, while leaving the remaining cores to execute the general-purpose workload of the system. Core isolation capabilities in Linux are provided by *cpuset* and *isolcpus* facilities. Both mechanisms are widely available in modern kernels and therefore they will be evaluated and compared in Section 5 to determine the advantages of applying one or another. A brief overview of each mechanism is described next:

— The *cpuset* facilities provide support for attaching processes to cores and memory node subsets. This means that a Linux process can only be executed on the cores belonging to the cpuset.
Two cpusets are created in our platform. By default, Linux boots with a single cpuset so the new configuration is applied explicitly at runtime. Some kernel threads are tied to a specific core and may not be migrated.
— Unlike cpuset, the *isolcpus* facilities are applied during system startup as a boot parameter. In the context of this article, isolcpus is applied as part of the *TuneD* service. This Linux service is based on scenario profiles to facilitate the proper configuration of different system settings. There are profiles for predefined scenarios such as low-latency, high-throughput, powersave, or CPU partitioning. The latter profile has been applied to the platform to remove user and kernel workload from the isolated core.

To enhance the isolation, the isolated cores should be specified in the *nohz_full* kernel parameter [38] to enable the adaptative tick mode. This mode offloads the kernel tick to a non-isolated core whenever some conditions are met. For instance, there should be only one runnable thread at most in the isolated core. However, it is worth noting that this parameter cannot offload a residual 1 Hz tick, which remains as a source of interference.

3.2.3 Other Configurations. To further reduce the source of interferences, other additional settings can be applied to the platform. For instance, some power management mechanisms such as ***Power performance states* (P-states)** and the ***Processor idle sleep states* (C-states)** may affect the determinism of the kernel [39] (i.e., a real-time application may suffer from unexpected delays when the CPU is switching from a power-save mode). The tradeoff between determinism and power-saving is specific to the real-time requirements of each system and it should be analyzed on a per-case basis.

As it was commented earlier, interrupts should be migrated to the non-isolated cores whenever it is possible. This can be accomplished by modifying the corresponding *smp_affinity* property of each interrupt. In Linux, interrupts are processed in interrupt context. This context is not

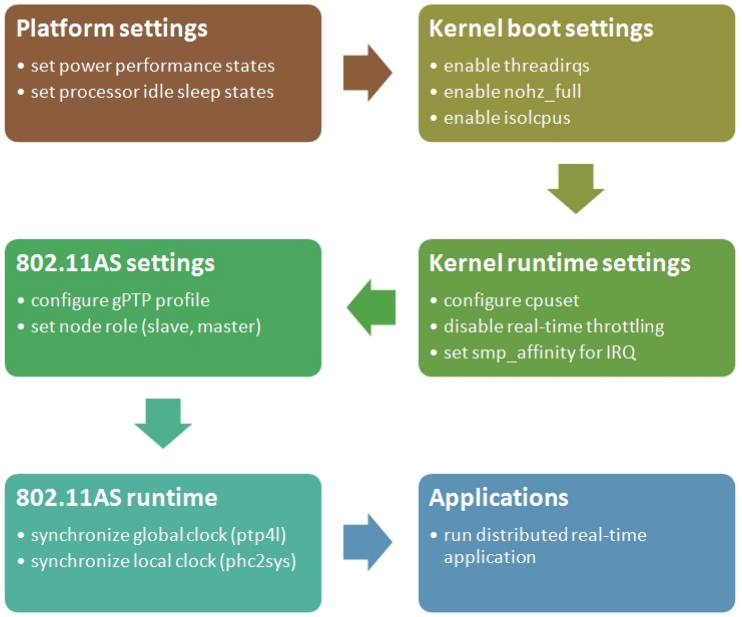

Fig. 2. Configuration process for the execution framework.

preemptible and therefore interrupt handlers should be configured to run in a threaded context through the *threadirqs* kernel parameter.

Finally, another feature that can impact the response times of real-time applications in Linux is *real-time throttling* (i.e., a safeguard limit for the execution of real-time tasks) [38], which should be disabled by means of the *kernel.sched_rt_runtime_us* kernel parameter.

A summary of the process followed to configure the execution framework is illustrated in Figure 2.

4 Benchmarking

This section proposes a set of tests to characterize the real-time behavior associated with the IEEE 802.1AS global clock from the application point of view. In particular, the tests have been developed to obtain the following key metrics:

—The overhead of getting the global time.
—The latency in the handling of events.
—The timing offset between the clocks of two computing nodes when they are synchronized.

Furthermore, it is also worthy to evaluate the various settings that can be applied to implement the proposed execution framework. To this end, different types of scenarios can be considered based on the configurations used for core isolation and clock synchronization, as follows:

—Three types of scenarios are defined based on the isolation mechanism applied: *no isolation, cpuset,* and *TuneD*. The former scenario can be considered the traditional approach to build and execute a real-time application in Linux and it is used as a reference in the context of this article.
—Two types of scenarios are defined based on the settings used for clock synchronization: ***SMS* (Slave-Master-Slave)**, where the switch is the GM and propagates the global time to

Table 2. Scenarios for the Analysis

Scenario Id	Core isolation	Clock sync	Target Node
RT-SMS-SLAVE	No isolation	SMS	Slave
RT-MBS-MASTER	No isolation	MBS	Master
RT-MBS-SLAVE	No isolation	MBS	Slave
TUNED-SMS-SLAVE	TuneD	SMS	Slave
TUNED-MBS-MASTER	TuneD	MBS	Master
TUNED-MBS-SLAVE	TuneD	MBS	Slave
CPUSET-SMS-SLAVE	cpuset	SMS	Slave
CPUSET-MBS-MASTER	cpuset	MBS	Master
CPUSET-MBS-SLAVE	cpuset	MBS	Slave

the slaves (i.e., the computing nodes), and ***MBS* (Master-Bridge-Slave)**, where one of the computing nodes is the GM. In the case of SMS, tests can be executed only in one of the slave nodes, since both nodes share the same role in the distributed system. In the case of MBS, both nodes should run the tests as they have different roles (i.e., master and slave).

As a result of the combination of both types of scenarios, the nine scenarios summarized in Table 2 are defined. The set of tests proposed in this section will be executed in each one of these scenarios.

Test 1: Estimation of the clock overhead
This test measures the extra time used by the kernel to perform a clock operation. In particular, the test measures the overhead of getting the time from the local clock (i.e., the POSIX monotonic clock) and from the global clock when it is accessed through the POSIX real-time clock by configuring *phc2sys* accordingly. To determine the overhead, the test calculates the difference between two consecutive accesses to the same clock. The pseudo-code for the test is as follows:

```
For 1 to 1000000:
      t1 = gettime(clock)
      t2 = gettime(clock)
      overhead = t2 – t1
```

Test 2: Cross estimation of clock overhead
This test aims at measuring the overhead of one clock using the opposite as reference; that is, the local clock is measured with the global clock and vice versa. The motivation behind this test is checking possible variations in the measurements depending on the clock used. The pseudo-code associated with this test is as follows:

```
For 1 to 1000000:
      t1 = gettime (ref_clock)
      gettime(clock_under_analysis)
      t2 = gettime (ref_clock)
      overhead = t2 – t1
```

Test 3: Event-handling latency
In the context of this article, the notion of event-handling latency is considered as the time interval between the triggering of an event (e.g., the firing of a timer interrupt) and the time when that event can be actually handled. This latency includes the overhead associated with several factors such as IRQ handling or scheduling.

This kind of latency can be measured using *cyclictest* [35], a tool available in a suite used for benchmarking a set of real-time Linux features. Among other configurable options, this tool allows selecting the clock to use (e.g., POSIX monotonic or real-time clock), the event-triggering period, or the number of iterations, which perfectly suits to perform this test.

Test 4: Synchronization between nodes
This test aims at evaluating the synchronization of clocks in a distributed system by measuring the time difference of periodically setting/clearing a digital signal triggered at the same time in the two computing nodes. In this case, the metrics are taken by measuring the delay between the two digital signals using an oscilloscope.

To better evaluate the clock synchronization, the overhead of setting/clearing the **General Purpose I/O (GPIO)** pins should also be taken into account. The pseudo-code associated with this overhead measurement is as follows:

```
For 1 to 1000000:
      t1 = gettime(clock)
      pulse up
      pulse down
      t2 = gettime(clock)
      time = t2 − t1
```

Test 5: Stressing situations
This test adds workload to Test 3 and Test 4 for the SMS scenario to check how the isolation is affected. The workload is added through *stress-ng* [41], a tool that allows synthetic workloads to be generated to stress different parts of the system (e.g., CPU or I/O). In this test, *stress-ng* has been configured to execute a total of 6 threads: 3 threads run CPU synthetic workload (which add approximately a 25% of CPU workload per thread), 2 threads perform I/O operations such as committing filesystem caches to disk or writing, reading, and removing files, and 1 thread creates timer clock interrupts at a rate of 1 MHz. Each thread can only be run in any of the available non-isolated cores.

5 Performance Evaluation

To obtain the proposed metrics, the tests described in the previous section are executed on a platform using the execution framework depicted in Section 3. In particular, the hardware platform consists of two computing nodes interconnected via a network switch with IEEE 802.1AS support (NXP LS1021ATSN). Each computing node has a quad-core Intel Atom E3845 processor at 1.91 GHz and a Gigabit Ethernet interface (Intel I210-IT). Both the switch and the computing nodes provide support for hardware timestamping (i.e., timestamping is performed by the PHC clock available in the NIC). To further reduce the source of interferences and obtain comparable metrics, the power management and frequency scaling have been disabled for the platform (i.e., disabling the *Power performance states* (*P-states*) and the *Processor idle sleep states* (*C-states*)) [39].

The switch executes the **Open Industrial Linux (OpenIL)** operating system, a special distribution of Linux for industrial environments. The computing nodes use a Linux kernel v.4.9 in which the PREEMPT_RT patch has been applied.

The set of tests are written in C. Besides applying the kernel configurations described in Section 3, tests have been coded following the latest recommendations in the community to develop real-time applications in a Linux kernel [1, 4, 40]. To better characterize the real-time behavior of the global clock, tests are executed without any extra computational workload added to the regular kernel workload except in those cases where it is explicitly stated (Test 5). Each test collects at least one million measurements at maximum priority. As each test uses a different logic, its duration

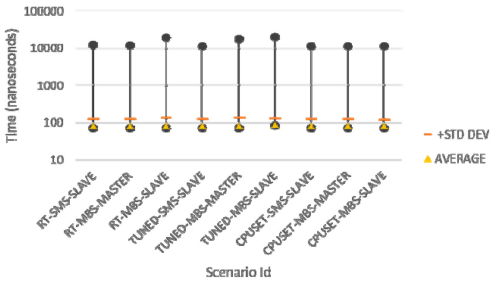

Fig. 3. Test 1- LC overhead.

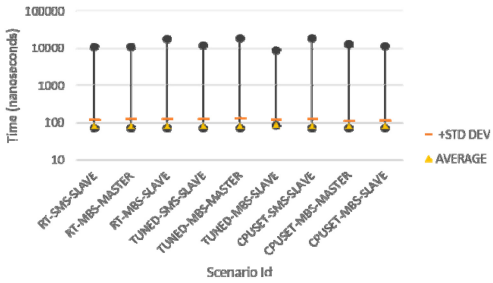

Fig. 4. Test 1- GC overhead.

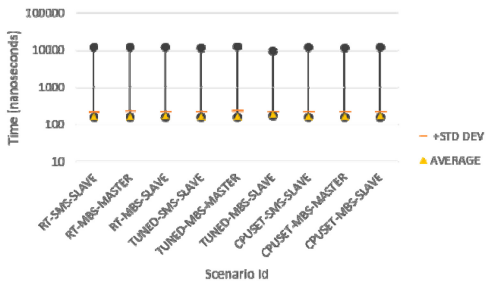

Fig. 5. Test 2- LC overhead measured with GC.

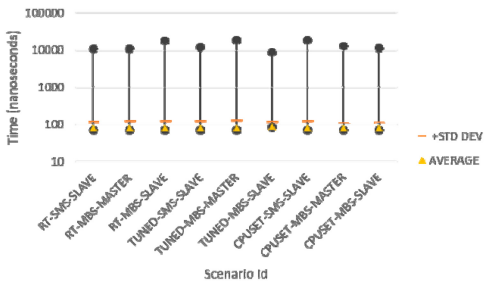

Fig. 6. Test 2- GC overhead measured with LC.

varies (e.g., some tests trigger a periodic signal to obtain each measurement). Measurements are stored in an array and post-processed to calculate the required performance metrics.

To better illustrate the results, the Y-axis is presented with a logarithmic scale in all figures except Figures 12 and 13. Furthermore, the abbreviations GC and LC stand for global clock and local clock, respectively. Last, each figure represents the maximum, minimum, and average values together with the average plus standard deviation except in the following cases:

—Test 3 does not include standard deviation, as the *cyclictest* tool does not provide it.
—The tests that rely on an oscilloscope to perform the measurements do not include minimum times as negative values obtained during the test (interpreted as minimum by the device) just indicate that the second signal is triggered first.

Figure 3 shows the overhead results for Test 1 using the local clock, which presents similar values for all the scenarios: around 13 μs of maximum, 71 ns of minimum, and 50 ns of standard deviation. Results for the global clock are depicted in Figure 4, which shows slightly higher maximum values than those for the local clock. In this case, maximum times were around 18 μs. Figure 5 and Figure 6 show the overhead results for Test 2 (i.e., cross estimation of clock overhead). They show similar overhead results for all the scenarios in both approaches.

For Test 1 and Test 2, there is a significant difference between maximum and minimum/average values, even in the scenarios in which isolation has been applied. It suggests that there are still sources of interferences from the Linux kernel, which can lead to add extra latencies in the range of tens of microseconds [2].

Figure 7 and Figure 8 show the results for Test 3 (event-handling latency) using local and global clock, respectively. In this case, the event-triggering period has been set to 10 milliseconds. The results show that the scenarios without isolation have maximum values higher than the ones that are isolated, while the average and minimum values are similar in all the scenarios.

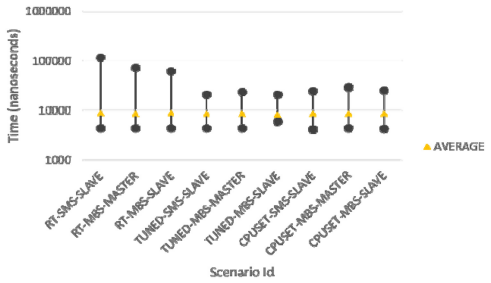

Fig. 7. Test 3- Event-handling latency with LC.

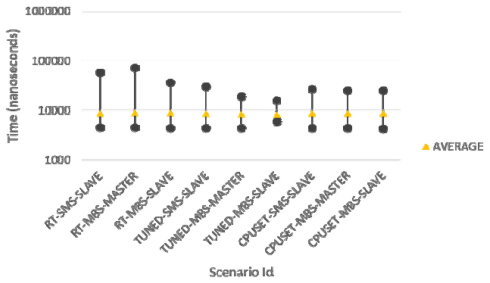

Fig. 8. Test 3- Event-handling latency with GC.

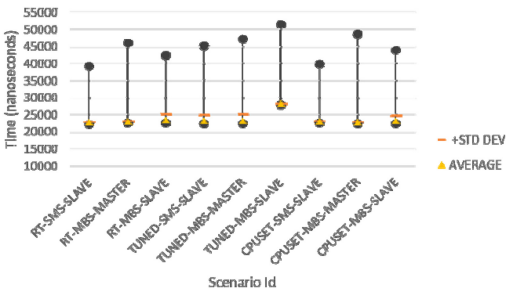

Fig. 9. Test 4- GPIOs overhead.

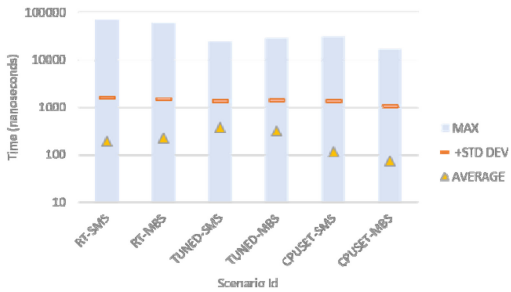

Fig. 10. Test 4- Nodes synchronization.

Before evaluating the timing offset in the synchronization of the global clock through Test 4, the overhead of using the GPIO pins was measured, and the results are depicted in Figure 9. These results show an average of 24 μs and a maximum and a minimum of 50 μs and 22 μs, respectively.

It is worth noting that Test 4 only considers 6 scenarios instead of 9, because master-and-slaves scenarios are equivalent for the MBS configuration, as end-to-end delays are measured through an oscilloscope. Figure 10 shows the results when a square wave signal that spends 10 ms at high value and 10 ms at low value is used (i.e., the signal has a period of 20 ms). While the standard deviation is similar for isolated and non-isolated scenarios (around 1 μs), slightly higher maximum values are obtained for the non-isolated scenarios.

For the first part of Test 5, Figure 11 shows the histogram obtained by adding workload to Test 3 using the global clock for RT-SMS and CPUSET-SMS scenarios. As can be seen, the isolated scenario obtains lower values for the maximum and the standard deviation. In both scenarios, the added workload has no significant impact on the maximum values obtained in Test 3.

Figure 12 and Figure 13 show the results obtained for the second part of Test 5, in which workload is added to Test 4 for the RT-SMS and CPUSET-SMS scenarios, respectively. For the former scenario, the maximum delay observed between the two digital signals is 60 μs, with a standard deviation of 3.2 μs. The CPUSET-SMS scenario obtained a maximum delay of 56 μs, with a standard deviation of 4 μs. Note that the maximum time should be obtained as the maximum absolute value of the Max and Min times illustrated in the figures, as negative values simply indicate that the second signal is raised first. According to the results, CPUSET-SMS obtains a slightly lower maximum value and better average times than the reference scenario. In both scenarios, the added workload slightly increases the average and standard deviation values obtained in Test 4. Comparing the results in Figures 10, 12, and 13, we can observe that the maximum overhead comes mainly from setting the signals and not from using the global clock. Furthermore, the overhead variability is not drastically affected by using the global clock (i.e., the standard deviation is around 4 μs).

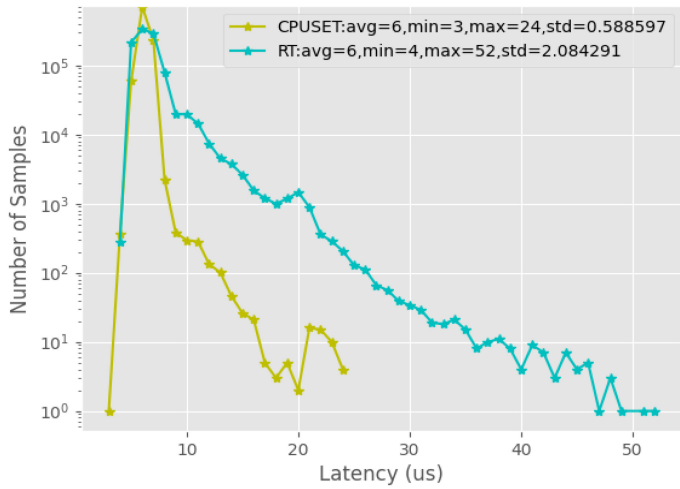

Fig. 11. Test 5- Event-handling latency with GC and workload.

In general, we can conclude that isolated scenarios obtain better timing results. However, we cannot identify an isolation mechanism (i.e., TuneD or cpuset) whose latencies are the lowest in any case. The main advantage of using the cpuset mechanism is flexibility, as cpusets can be created/removed at runtime. However, the TuneD service allows the modification of other non-functional aspects, such as power saving or low-latency, in addition to the core isolation facilities analyzed in this article.

Adding workload does not have much impact on the maximum values when the real-time application is isolated and/or executed at maximum priority. Despite the isolation capabilities provided by the Linux kernel, it is important to note that they do not provide full isolation, as there are still sources of interferences such as shared caches, global work queues, or interprocessor interrupts. Additionally, there are also a few kernel threads bound to each core whose workload cannot be migrated. Finally, the role of the node in SMS and MBS scenarios seems to have little effect on the performance evaluation.

Despite the fact that this type of measurement can vary depending on the hardware platform, system workload, and kernel version, our results obtained at the application level complement and are consistent with those reported by similar studies [33, 34, 37]. Relevant testbeds, such as TSN-FlexTest [23] or EnGINE [21], have reported master-slave clock deviation in the nanosecond range, which is an order of magnitude lower than the values obtained in our tests. However, it is important to note that our approach is slightly different, as it focuses on estimating the effective synchronization achieved by the entire execution framework. This involves (1) collecting measurements at the application level, (2) using standard kernel tools and configurations, and (3) adding extra computational workload to the regular kernel workload to mimic an industrial context and provide insights into practical applicability. Furthermore, a synchronization accuracy in the microseconds range can be sufficient for many different applications [34].

6 Conclusions and Future Work

Nowadays, Linux is actually present in many real-time industrial applications and Industry 4.0 envisages new possibilities for it. Traditionally, the presence of a global clock in distributed real-time systems was limited to critical applications using time-triggered or table-driven schedulers. Nevertheless, the set of Ethernet standards TSN introduces a high-precision clock synchronization protocol that can widely extend its potential applications.

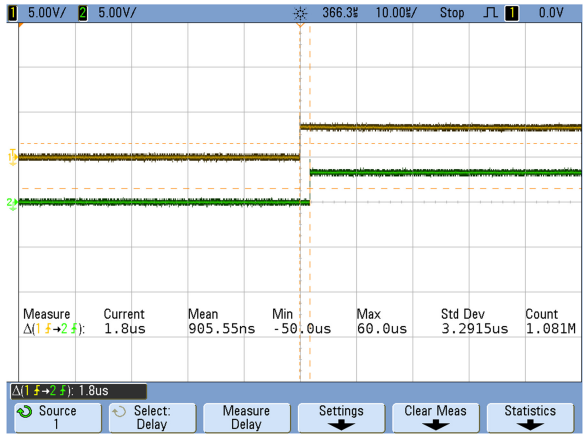

Fig. 12. Test 5- Nodes synchronization with workload (RT-SMS).

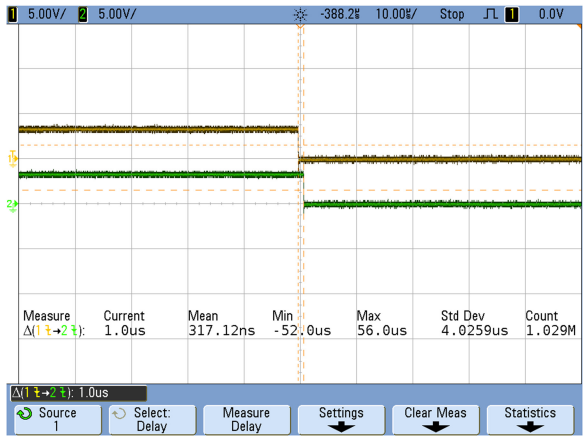

Fig. 13. Test 5- Nodes synchronization with workload (CPUSET-SMS).

In this research, we have outlined an execution framework designed for distributed real-time systems in multicore processors. To this end, this article has identified and analyzed two different mechanisms within the Linux kernel (i.e., *cpusets* and *isolcpus*) for core isolation to enhance the real-time behavior by minimizing the interactions with the remaining non-real-time applications running in the same processor. Considering the results obtained, none of these two mechanisms strictly provides full isolation, and therefore real-time applications may suffer from some unexpected latencies. Furthermore, as none of the isolation mechanisms performed better in all the tests, the decision to select one or another should be based on other factors such as flexibility or the ability to easily modify other non-functional aspects.

Furthermore, this article has provided a characterization of the 802.1AS clock synchronization mechanism at the application level. We have evaluated the global clock performance by measuring overheads and latencies, as well as the effective synchronization of the clock synchronization mechanism. This evaluation has been performed by characterizing the different configurations through specific tests and also through *cyclictest*, a common tool for measuring kernel latencies in Linux. This article has obtained the following results for the set of proposed key metrics:

— *Overheads in reading the global clock.* We have found that applications running in the execution framework have similar overheads for accessing the global clock as for accessing local clocks.
— *Latencies in the handling of events.* The tests have confirmed that their maximum values are lower when an isolation mechanism is applied, as expected.
— *Offsets in the synchronization between nodes.* When both nodes are synchronized, it is observed that using isolation minimizes the interference from the kernel and other workload, thus decreasing the maximum offset between two synchronized digital signals. According to the results, the maximum offset measured is in the range of tens of microseconds.

Finally, the article also provides a set of recommended settings for both the Linux kernel and the 802.1AS clock synchronization protocol. Our recommendations aim to bound response times in distributed real-time applications, and they also facilitate the replication of our software environment for testing purposes.

As a general conclusion, the results from the evaluation show that devices' clocks in a network can be synchronized at the application level with a precision in the range of tens of microseconds. Therefore, it is possible to execute a wide range of industrial real-time applications with the proposed execution framework. Future lines of work on this topic may include exploring other Linux kernel features to enhance determinism at the network level (such as configuring the network stack) or incorporating other TSN standards to the proposed framework.

References

[1] Michael M. Madden. 2019. Challenges using Linux as a real-time operating system. In *AIAA SciTech Forum: Software Challenges in Aerospace*. DOI: 10.2514/6.2019-0502
[2] F. Reghenzani, G. Massari, and W. Fornaciari. 2019. The real-time Linux kernel: A survey on PREEMPT_RT. *ACM Comput. Surv* 52, 1 Article 18, (2019). DOI: https://doi.org/10.1145/3297714
[3] A. Pérez, M. Aldea Rivas, and M. González Harbour. 2015. CPU isolation on the Android OS for running real-time applications. In *13th International Workshop on Java Technologies for Real-time and Embedded Systems*.
[4] Red Hat Inc. 2020. Red Hat enterprise Linux for real time 7 - tuning guide—Advanced tuning procedures to optimize latency in RHEL for real time. Revision 1.8. Retrieved from https://docs.redhat.com/en/documentation/red_hat_enterprise_linux_for_real_time/7/html/tuning_guide/index
[5] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis. 2022. A comprehensive survey of industry practice in real-time systems. *Real-time Syst.* 58 (2022), 358–398. https://doi.org/10.1007/s11241-021-09376-1
[6] DIN and DKE German Commission for Electrical, Electronic & Information Technologies. 2023. "German standardization roadmap industrie 4.0", version 5. Retrieved from https://www.din.de/en/innovation-and-research/industry-4-0/german-standardization-roadmap-on-industry-4-0-77392
[7] Karsten Schweichhart. 2024. RAMI 4.0, Reference architectural model industrie 4.0. Retrieved from https://www.plattform-i40.de/PI40/Redaktion/EN/Downloads/Publikation/rami40-an-introduction.html
[8] T. Fedullo, A. Morato, F. Tramarin, L. Rovati, and S. Vitturi. 2022. A comprehensive review on time sensitive networks with a special focus on its applicability to industrial smart and distributed measurement systems. *Sensors* 22 (2022), 1638. DOI: https://doi.org/10.3390/s22041638
[9] S. Scanzio, L. Wisniewski, and P. Gaj. 2021. Heterogeneous and dependable networks in industry—A survey. *Comput. Industr.* 125 (2021), ISSN 0166–3615, DOI: https://doi.org/10.1016/j.compind.2020.103388
[10] IEEE Inc. 2020. IEEE standard for local and metropolitan area networks–timing and synchronization for time-sensitive applications. IEEE Std 802.1AS-2020 (Revision of IEEE Std 802.1AS-2011), 1–421.
[11] IEEE Inc. 2019. IEEE standard for a precision clock synchronization protocol for networked measurement and control systems. IEEE Std 1588-2019 (Revision of IEEE Std 1588-2008), 1–499.
[12] H. Kopetz. 1998. The time-triggered model of computation. In *19th IEEE Real-Time Systems Symposium*. 168–177. DOI: 10.1109/REAL.1998.739743
[13] H. Kopetz. 2011. *Real-time Systems: Design Principles for Distributed Embedded Applications (Real-time Systems Series)*, 2nd ed. Springer.
[14] Airlines Electronic Engineering Committee, Aeronautical Radio INC. 2019. "Avionics Application Software Standard Interface, required Services". *ARINC Specification* 653, Part 1, revision 5, SAE Industry Technologies Consortia.

[15] J. M. Rivas, J. J. Gutiérrez, J. C. Palencia, and M. G. Harbour. 2015. Deadline assignment in EDF schedulers for real-time distributed systems. *IEEE Trans. Parallel Distrib. Syst.* 26, 10 (2015), 2671–2684. DOI: 10.1109/TPDS.2014.2359449

[16] H. Pérez Tijero and J. J. Gutiérrez. 2022. EDF scheduling for distributed systems built upon the IEEE 802.1AS clock - A theoretical-practical comparison. *J. Syst. Arch.* 132 (2022). DOI: https://doi.org/10.1016/j.sysarc.2022.102742

[17] M. Gutiérrez, W. Steiner, R. Dobrin, and S. Punnekkat. 2017. Synchronization quality of IEEE 802.1AS in large-scale industrial automation networks. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'17)*. 273–282. DOI: 10.1109/RTAS.2017.10

[18] S. Fuchs, H. Schmidt, and S. Witte. 2016. Test and on-line monitoring of real-time ethernet with mixed physical layer for Industry 4.0. In *IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA'16)*. 1–4. DOI: 10.1109/ETFA.2016.7733518

[19] G. M. Garner, A. Gelter, and M. J. Teener. 2009. New simulation and test results for IEEE 802.1AS timing performance. In *International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*. 1–7. DOI: 10.1109/ISPCS.2009.5340214

[20] H. -H. Liu et al. 2024. Improving TSN simulation accuracy in OMNeT++: A hardware-aligned approach. *IEEE Access* 12 (2024), 79937–79956. DOI: 10.1109/ACCESS.2024.3410109

[21] M. Bosk, F. Rezabek, J. Abel, K. Holzinger, M. Helm, G. Carle et al. 2023. Simulation and practice: A hybrid experimentation platform for TSN. In *IFIP Networking Conference (IFIP Networking'23)*. 1–9.

[22] D. Andronovici, I. Turcanu, J. Bigge, and C. Sommer. 2024. Cross-validating open source in-vehicle TSN simulation models with a COTS hardware testbed. In *IEEE Vehicular Networking Conference (VNC'24)*. 172–179. DOI: 10.1109/VNC61989.2024.10575954

[23] M. Ulbricht, S. Senk, H. K. Nazari, H. Liu, M. Reisslein, and G. T. Nguyen. 2024. TSN-FlexTest: Flexible TSN measurement testbed. *IEEE Trans. Netw. Service Manag.* 21, 2 (2024), 1387–1402. DOI: 10.1109/TNSM.2023.3327108

[24] E. Kyriakakis, J. Sparsø, and M. Schoeberl. 2018. Hardware assisted clock synchronization with the IEEE 1588-2008 precision time protocol. In *26th International Conference on Real-Time Networks and Systems (RTNS'18)*. Association for Computing Machinery, New York, NY, USA, 51–60. DOI: https://doi.org/10.1145/3273905.3273920

[25] Y. J. Kim, J. H. Kim, B. M. Cheon, Y. S. Lee, and J. W. Jeon. 2014. Performance of IEEE 802.1AS for automotive system using hardware timestamp. In *18th IEEE International Symposium on Consumer Electronics (ISCE'14)*. 1–2. DOI: 10.1109/ISCE.2014.6884384

[26] M. Kassouf, L. Dupont, J. Béland, and A. Fadlallah. 2013. Performance of the precision time protocol for clock synchronisation in smart grid applications. *Trans. Emerg. Tel. Tech* 24 (2013), 476–485. DOI: https://doi.org/10.1002/ett.2660

[27] G. Miranda, E. Municio, J. Haxhibeqiri, D. F. Macedo, J. Hoebeke, and I. Moerman. 2022. Time-sensitive networking experimentation on open testbeds. In *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS'22)*. 1–6. DOI: 10.1109/INFOCOMWKSHPS54753.2022.9798073

[28] O. Konradi, A. Mankowski, L. Wisniewski, and H. Trsek. 2022. Towards an industrial converged network with OPC UA PubSub and TSN. In *27th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'22)*. 1–4. DOI: 10.1109/ETFA52439.2022.9921646

[29] A. Mahmood, R. Exel, H. Trsek, and T. Sauter. 2017. Clock synchronization over IEEE 802.11—A survey of methodologies and protocols. *IEEE Trans. Industr. Inform.* 13, 2 (2017), 907–922. DOI: 10.1109/TII.2016.2629669

[30] A. B. D. Kinabo, J. B. Mwangama, and A. A. Lysko. 2021. An evaluation of broadband technologies from an industrial time sensitive networking perspective. In *7th International Conference on Advanced Computing and Communication Systems (ICACCS'21)*. 715–722. DOI: 10.1109/ICACCS51430.2021.9441748

[31] I. Val, O. Seijo, R. Torrego, and A. Astarloa. 2022. IEEE 802.1AS clock synchronization performance evaluation of an integrated wired-wireless TSN architecture. *IEEE Trans. Industr. Inform.* 18, 5 (2022), 2986–2999. DOI: 10.1109/TII.2021.3106568

[32] M. Gundall, C. Huber, S. Melnyk, and H. D. Schotten. 2021. Extending reference broadcast infrastructure synchronization protocol in IEEE 802.11 as enabler for the industrial internet of things. In *4th IEEE International Conference on Industrial Cyber-Physical Systems (ICPS'21)*. 118–124. DOI: 10.1109/ICPS49255.2021.9468146

[33] J. Pacner, O. Rysavý, and M. Svéda. 2013. On the evaluation of clock synchronization methods for networked control systems. In *3rd Eastern European Regional Conference on the Engineering of Computer Based Systems*. 161–162. DOI: 10.1109/ECBS-EERC.2013.30

[34] R. Cochran, C. Marinescu, and C. Riesch. 2011. Synchronizing the Linux system time to a PTP hardware clock. In *IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communicatio*. 87–92. DOI: 10.1109/ISPCS.2011.6070158

[35] Cyclictest manual page: http://manpages.ubuntu.com/manpages/cosmic/man8/cyclictest.8.html. Source code can be Retrieved October 2024 from https://git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git/

[36] J. Ruh, W. Steiner, and G. Fohler. 2021. Clock synchronization in virtualized distributed real-time systems using IEEE 802.1AS and ACRN. *IEEE Access* 9 (2021), 126075–126094. DOI: 10.1109/ACCESS.2021.3111045

[37] J. Lázaro, J. Cabrejas, A. Zuloaga, L. Muguira, and J. Jiménez. 2022. Time sensitive networking protocol implementation for Linux end equipment. *Technologies* 10 (2022), 55. DOI: https://doi.org/10.3390/technologies10030055
[38] The Linux Kernel documentation. Available online: https://www.kernel.org/doc/Documentation/. Last access: October 2024.
[39] R. Schöne, D. Molka, and M. Werner. 2015. Wake-up latencies for processor idle states on current x86 processors. *Computer Science - Research and Development* 30 (2015), 219–227. https://doi.org/10.1007/s00450-014-0270-z
[40] J. Ogness. 2020. A checklist for writing Linux real-time applications. In *Embedded Linux Conference (ELC'20).*
[41] Stress-ng tool. 2024. Manual available online: https://wiki.ubuntu.com/Kernel/Reference/stress-ng. Source code can be Retrieved October 2024 from https://github.com/ColinIanKing/stress-ng

Received 4 April 2024; revised 31 July 2024; accepted 29 August 2024