OpenClock: A Testbed for Clock Synchronization Research

Fatima M. Anwar, Amr Alanwar, and Mani B. Srivastava
University of California, Los Angeles, CA, USA
{*fatimanwar, amr.alanwar, mbs*}*@ucla.edu*

***Abstract*—Clock synchronization protocols have always been tested and compared in controlled environments. The hardware variability of different platforms, and network variability in communication channels is often ignored. Most of the protocols are not compared for faults, failures or adversarial attacks because it is hard to reproduce them on different devices. The presence of few clocks on a single device limits the use of one device to test multiple synchronization protocols at once. For fair comparison of multiple synchronization protocols, it is essential that their disciplinable clocks are all derived from the same clock hardware, and they process the same network traffic. We propose OpenClock, a clock synchronization testbed that manages synchronization resources and provides multiple disciplinable clocks on a single platform. OpenClock features a rich set of clocks for modular and extensible design, and an attack simulator for testing algorithmic resilience. Users can leverage the attack capability to find vulnerabilities in the synchronization algorithms, and test the resilience of secure algorithms. We prototype OpenClock on an embedded platform and x86 desktop. We evaluate variants of PTP and NTP protocols on the embedded platform under various clock parameters, disciplining mechanisms, and attack scenarios.**

I. Introduction

Hardware capabilities required for clock synchronization have developed significantly in the past decade; hardware timestamping feature is introduced for many processors, co-processors [1], and network interface cards [2], and new timing abstractions have been added in operating systems for precise timing [3]. Systems have extensively made use of time based technology developments to push for higher timing accuracy. These systems however lack a comprehensive testing environment to test and compare synchronization algorithms.

Multiple algorithms can be tested fairly if they experience the same clock model. Clock models are quite complex; they are affected by short and long term variations in clock jitter, wander, and skew due to physical characteristics of oscillators and environmental variations due to temperature and aging. It is known in clock literature that no two clocks are the same, Cho et al. [4] has even used unique clock characteristics for fingerprinting electronic control units in cars.

Industrial and automotive applications heavily rely on clock synchronization. These applications operate under uncertain environments and prone to hardware faults, network failures, or man in the middle attacks. To develop a clock synchronization algorithm that is robust to faults, failures, and attacks, comprehensive testing is necessary before practical deployments in safety critical applications. Unfortunately, many algorithms are not tested for faults and attacks as it is hard to reproduce them on distributed devices.

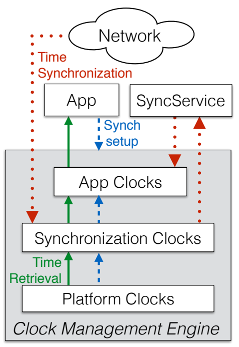


Fig. 1: Testbed architecture

For fair algorithmic comparison, and the ability to test algorithms under faults and attacks, we assert that it is essential to provide a clock synchronization testbed on a single host, as a single platform is subject to same hardware and network conditions. The big challenge however in testing clock synchronization algorithms on a single platform is the absence of multiple similar clocks. We propose OpenClock, a testbed that supports multiple virtual clocks derived from the same physical clock, alleviating the bias in results from the physical and network characteristics. OpenClock consists of multiple components as shown in Figure 1. A clock management engine initializes and manages three layers of clocks: platform clocks that define the hardware timing capabilities, synchronization clocks that assist time synchronization protocols, and application clocks that provide the notion of time to applications.

As shown in Figure 1, OpenClock supports multiple clock synchronization protocols in three key steps. First, the dashed blue line represents the initial setup that bootstraps the required clocks for synchronization. Second, the dotted red line shows the interactions among clocks, synchronization services, and the network to perform clock synchronization and discipline the virtual clocks. Finally, the solid green line retrieves time from disciplined virtual clocks and transfer it to applications. These three steps are necessary in testing a clock synchronization algorithm.

OpenClock also has the capability of testing and comparing algorithms under attacks. It consists of a network attack simulator that can be used to inject different kinds of attacks. Users leverage the attack simulator to test when their algorithms fail. Users can also design new algorithms and test their algorithms' resilience to attacks. The attack simulator imitates an adversary sitting on a network element that can arbitrarily delay victims packets in the network. To present different ways in which OpenClock can be used, we test three use cases in this paper. However, OpenClock is not limited to these use cases. First, we present an attack on NTP packets, then compare performance of a simple NTP versus a relatively secure NTP in the presence of attacks. We also show two use cases that compare synchronization performance of two protocols running different clock disciplining mechanisms in different intervals.

In this work, we lay the foundations of a clock synchronization testbed. Its usage can be extended in terms of synchronization algorithms and their parameters, and the kinds of hardware and network attacks. In the rest of the paper, we first go over the categorization of clocks in OpenClock in Section II. Then we explain the complete OpenClock architecture in Section III followed by an evaluation in Section IV.

II. Hierarchy of Clocks

Clocks are used for timekeeping, timestamping, and scheduling. A clock is represented by a timing stack. This stack consists of an oscillator that oscillates at a particular frequency. The frequency of oscillation corresponds to the clock resolution. A counter counts the oscillations, and a software converts those counts to a human readable time in seconds. The ability of a clock to measure small time intervals is limited by its resolution, and a clock can be no more accurate to some reference time than its resolution.

A clock could either be a software abstraction in the operating system or a logical mapping in the application. Our proposed clock synchronization testbed OpenClock consists of a hierarchy of clocks. The clocks at the bottom of the hierarchy are *(A) platform clocks*; they represent the timing characteristics of a particular hardware. The middle layer in the clock hierarchy is comprised of *(B) synchronization clocks*; they are derived from the platform clocks and assist time synchronization protocols. The clocks at the top of the hierarchy are *(C) application clocks*; they are derived from synchronization clocks, and exposed to applications for timekeeping, and timestamping.

A. Platform Clocks

Platform clocks define the timing capabilities of a given platform. There are three types of platform clocks; a 1) system clock, a 2) precise hardware clock, and a 3) peripheral clock. Every device has at least one of these clocks. Below, we explain the types of platform clocks in reference to the Linux operating system.

1. System Clock provides a local sense of time to the operating system and user processes. System time is the number of time units passed since an epoch e.g, POSIX-compliant systems such as Linux count seconds since 1st January 1970. It's timing stack is shown in Figure 2 (middle vertical path). The system clock is typically implemented as a programmable timer that periodically interrupts the CPU.

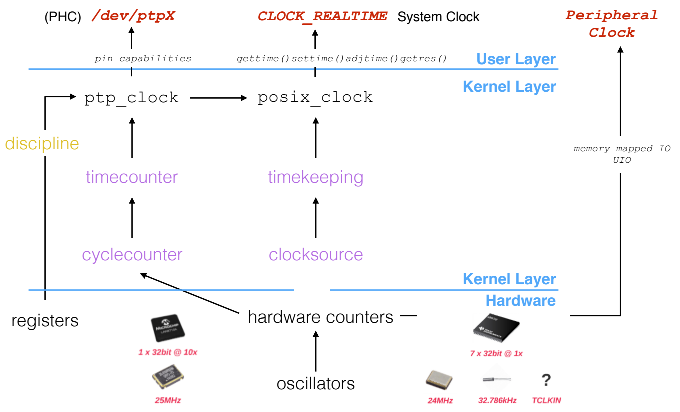


Fig. 2: The timing stack of three different Platform Clocks in Linux OS

Linux uses timers as the basis for higher-level clock abstractions: the `clocksource` encapsulates a non-wrapping hardware counter. The clocksource contains a member-function for reading the hardware counter, and `mult` and `shift` parameters that convert the counter value into nanoseconds for timekeeping. `CLOCK_REALTIME` is the Linux system clock that is exposed to userspace via the standardized POSIX clock interface. This interface allows the system clock to be disciplined using synchronization algorithms such as NTP [5] and PTP [6]. We refer to the system clock as `SYS` from now onward.

2. Precise Hardware Clock (PHC) is capable of timestamping events and scheduling tasks in hardware [7]. Unlike system clock, a PHC provides hardware pins for precise timestamping and accurate interrupt generation. In Figure 2 (left vertical path), an oscillator source is used to drive a hardware counter, from which an overflow-safe logical PTP clock `ptp_clock` is derived using the `cyclecounter` and `timecounter` abstractions. This clock is exposed to userspace as a PTP character device. It extends a POSIX clock interface and implements hardware pin functionalities. A PTP clock is also called a PHC. A userspace daemon can synchronize this clock to other PHCs in the network. PHC provides high synchronization accuracy and low synchronization jitter because it timestamps network packets in hardware. We refer to a PHC clock as `PHC` from now onward.

3. Peripheral Clock is a bare-metal peripheral clock on a platform such as a co-processor clock. The applications running on an operating system can access this clock via the Userspace I/O (UIO) Linux kernel subsystem as shown in Figure 2 (right vertical path). UIO maps regions of the peripheral clock memory and registers directly into userspace, with a small amount of kernel-space code to handle interrupts. This allows most of the driver logic to run in userspace instead of kernel-space, reducing the need for debugging kernel modules. UIO is used because it has low latency and is supported by both old and new Linux kernels. Thus a peripheral's clock is accessed at the userspace via UIO. We refer to this clock as `PPHL` from now onward.

B. Synchronization Clocks

For the working of a time synchronization protocol, two clocks are needed. A local clock that is used for timekeeping, and a network interface clock that timestamps the incoming and outgoing packets at the interface. For some platforms and synchronization protocols, a single clock is used both for timekeeping and packets timestamping (NTP [5]). While in others, different clocks are used for timekeeping and timestamping (PTP [6], FTSP [8]). OpenClock presents two synchronization clocks that are derived from platform clocks.

1. Core Clock is the first synchronization clock. It does timekeeping by providing a core sense of time to the entire platform. For a clock to qualify as a core clock, it must provide (i) the ability to read a strictly monotonic counter that cannot be altered. Optionally, a core clock could have (ii) the ability to generate interrupts at precise time, and (iii) provide the hardware resolution and uncertainty associated with reading the clock. Core clocks may also provide interfaces to expose platform-specific functionality such as, timestamping and generating external events.

2. Network Interface Card Clock (NICC) is the second synchronization clock. It timestamps the network packets in a synchronization protocol. Only those network interfaces that can accurately timestamp packet transmission and reception – at the physical or MAC layers – are exposed as NICC. This enables clock synchronization protocols to precisely estimate the offset between two clocks, and the propagation delay associated with a medium. Like core clocks, NICC provide the ability to read time, and optionally provide I/O functionality for precisely timestamping an event, or generating a very deterministic pulse in the future. However, a NICC is not necessarily monotonic and it can be disciplined. In addition, NICC do not provide the ability to generate interrupts, and cannot be used to accurately schedule user-level application threads.

While every node must contain a core clock, the presence of a NICC is optional. However, we believe that both categories of clocks are required for precise clock synchronization. Depending upon the hardware capabilities of a platform, and synchronization requirements of an application, a core clock can be derived from any platform clock, whereas, NICC can only be derived from a `PHC` or a `PPHL`.

C. Application Clocks

Multiple clocks are needed to compare different synchronization protocols on a single platform. We leverage a timing abstraction called a `timeline` that derives its time using a projection from core clock. Hence we have the capability to maintain multiple core clock projections in the form of multiple timelines. The timeline abstraction was first proposed by Anwar et al. [3] where they used it to synchronize distributed clocks with desired accuracy. Note that in OpenClock, the synchronization protocols discipline application clocks i.e. the timelines. The synchronization clocks are not disciplined, they are only used to assist the synchronization protocol.

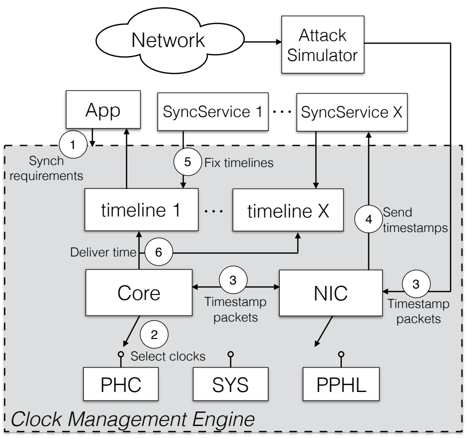


Fig. 3: Clock synchronization steps in OpenClock architecture

Timeline maintains a virtual time base with respect to an epoch. The timeline abstraction enables the OS to provide as many disciplinable clocks as needed by the applications. Timelines enforce isolation among different synchronization algorithms and their respective clock adjustment routines, by providing a unique disciplinable virtual clock to each synchronization protocol. An application creates a timeline with a unique uuid and specifies its synchronization requirements and protocols. These protocols synchronize their respective timelines under the given requirements. Hence the timeline abstraction provides a natural isolation among multiple protocols running on the same platform.

III. OpenClock Architecture

We present OpenClock, a clock synchronization testbed that is used to compare different synchronization algorithms on a single platform. Figure 3 shows the OpenClock architecture. It consists of a *clock management engine* that interacts with *synchronization services* to manage and discipline timelines. The detailed procedure is: (1) an application registers its synchronization requirements with the clock management engine. (2) This engine assigns a platform clock to core clock and NICC according to the requirements. (3) If a different platform clock is assigned to core and NICC, they both should synchronize to each other and present the same time. NICC timestamps network packets to synchronize to a reference time, and (4) send those timestamps to all the synchronization services. (5) These services utilize the timestamps in their algorithms to calculate clock disciplining parameters, and fix their respective timelines. Note that each synchronization service disciplines its corresponding timeline. (6) The applications can then retrieve synchronized time from their timelines, and compare the performance of different synchronization algorithms. We now cover the key components of OpenClock in detail.

A. Clock Management Engine

The clock management engine initializes and manages the hierarchy of clocks (explained in Section III). This engine assess the hardware timing capabilities of a platform and expose them as platform clocks. `SYS` is present on every

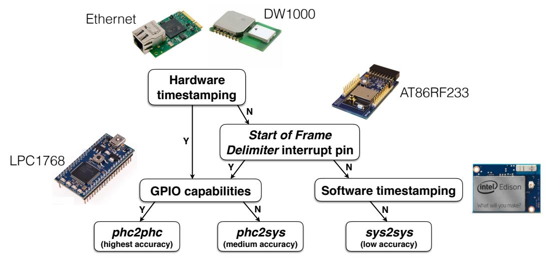


Fig. 4: Hardware capabilities of clocks influence synchronization performance (adapted from [3])

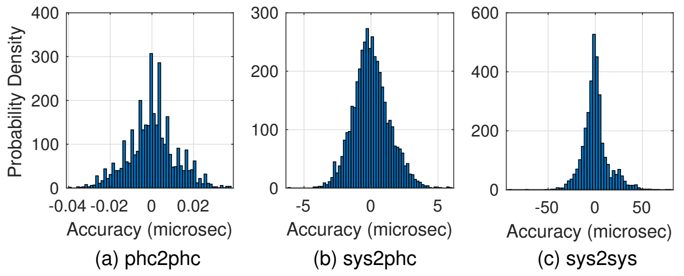


Fig. 5: Comparison of time accuracies for different synchronization clocks

platform and operating system. However, this clock does not have the hardware capabilities to measure precise intervals or schedule tasks at precise time instants. `PHC` have these hardware capabilities but it's been tied to ethernet interfaces on certain platforms. In our previous work [2], we show if a network interface support certain hardware features, we can expose it as a `PHC` by writing drivers for it. We propose that besides network interfaces, a processor clock or a co-processor clock can also be converted to a PHC, given that they have necessary hardware functionalities. In this work, we expose a processor (AM335X) as a `PHC` because it is capable of timestamping and scheduling hardware events. We use the processor timer and wrote a kernel module to transform it into a `PHC`. The clock management engine also initializes any `PPHL` on a platform. For example, on the beaglebone black platform, there is a Programmable Realtime Unit (PRU) that has good timing capabilities. Our previous work cyclops [1] exposes PRU as a timing device and synchronizes it to the processor clock. We utilize the PRU clock as `PPHL` in OpenClock, and we believe similar peripherals on other platforms can provide a `PPHL` platform clock. Hence a `SYS`, `PHC`, and `PPHL` are initialized and managed by the clock management engine.

After initializing the platform clocks, the clock management engine assigns the platform clocks to synchronization clocks based on the synchronization requirements. It also dynamically switches core clocks and NICC to different platform clocks. For example, if a user wants to run NTP, the engine selects core clock as `SYS` and leave the NICC empty. If the user wants to run PTP, core clock is selected as `SYS` and NICC as `PHC`. To run the QoT Stack [3] with highest accuracy, both core clock and NICC are chosen as `PHC`.
We know that the hardware capabilities influence the accuracy of a synchronization protocol. In Figure 4, the flow chart and the accompanying hardware platforms provide a guideline; given certain capabilities, how would you traverse down the flowchart to determine which accuracy the protocol will achieve. For example, given an Intel edison platform, the clock management engine cannot select either the core or the NICC as a `PHC` because edison is incapable of hardware timestamping. On the other hand, for a DW1000 platform interfaced with Beaglebone Black, the engine selects both the core and NICC to be a `PHC` [3]. For LPC1768, only core clock is a `PHC`, while NICC is empty.

To test the affect of platform clock on the synchronization accuracy, we run an experiment to synchronize the core clock with the NICC on one device. We use three different combinations of `SYS` and `PHC`. In the first experiment, both the core and NICC are `PHC`, and the synchronization accuracy achieved is in the order of nanoseconds as shown in Figure 5a. In the second experiment, we choose core as `SYS` and NICC as `PHC`, and the accuracy reduced to microseconds in Figure 5b. In the last experiment, we choose both the core and NICC as `SYS`. The accuracy reduced even further to 10s of microseconds in Figure 5c.

In OpenClock, besides using the traditional protocols, users can write their own synchronization protocols and choose any platform clock to act as core or NICC. The user also has the advantage of testing established time synchronization algorithms such as NTP and PTP with different clock settings. In OpenClock, a user specifies its own clock of choice, if however a user doesn't specify the synchronization clocks, the engine choose `SYS` to be the default core clock as it is available on every platform. The engine also maintains default clock settings for known synchronization protocols but they can be overridden if desired.

After assigning desired platform clocks to synchronization clocks, the engine creates timelines as application clocks. Recalling from Figure 3, a timeline is a projection of core clock's time. The engine maintains the projection parameters for all timelines, and provides an interface to all synchronization services to change the projection parameters of their respective timelines.

B. Synchronization Service

As shown in Figure 3, multiple synchronization services work with the clock management engine in OpenClock. A synchronization service utilizes the engine's interface to discipline its timeline. OpenClock also provides multiple parameters that tune the performance of a synchronization algorithm. The two tunable parameters are, the 1) synchronization interval, and the 2) clock discipline mechanism. These tunable parameters can be changed during initialization or at runtime. The performance is enhanced by reducing the first tunable parameter i.e. the synchronization interval. To tune the other parameter, one has to choose between two different mechanisms to discipline a clock. A `feedback` mechanism timestamps packets and

calculates synchronization parameters from the disciplined clock. On the other hand, a `feedforward` mechanism is based on a clock that is never disciplined. The calculated synchronization parameters reflect the local clock's relative drift with respect to global time.
Using OpenClock, users can write their own synchronization algorithms and specify their own tunable parameters.

C. Network Attacks Simulator

OpenClock supports comparison of multiple algorithms under fair conditions. It also provides an opportunity to compare algorithmic resilience to attacks on network packets. With the system and network attacks on the rise, there is a need to design algorithms that are both resilient to faults and attacks. OpenClock lets the user test its algorithms under various kinds of attacks by providing a network attack simulator. This simulator injects delays in packets transmission and reception as shown in Figure 3. These attacks compromise the accuracy of time by delaying the packets to be timestamped. The attacks can be injected in both forward and reverse paths in the network. The attacked packets are fed to the synchronization algorithms that need to be compared.

To use the OpenClock testbed, a user provides a configuration file to the system. This file specifies the name of timelines, types of clocks, protocols for timelines along with their tunable parameters, and an attack indicator. One example configuration file is shown below,

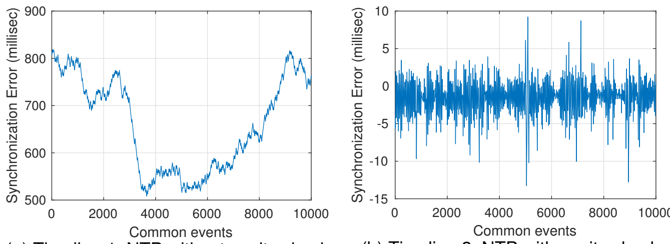
```
timeline1{
                Core: SYS
                NICC: PHC
                SYNC:{
                        protocol: PTP
                        servo: feedforward
                        interval: 1
                }
                attack: true
}

timeline2{
  ...
}
...
```

Here, the user has defined two timelines. On one timeline, she configures core clock as `SYS` and NICC as `PHC` running PTP algorithm that synchronizes to a master every second using a feedforward clock disciplining mechanism.

IV. Evaluation

The purpose of providing a testbed for clock synchronization protocols is to compare multiple algorithms under same hardware and network conditions. OpenClock leverages timeline to provide multiple disciplinable clocks on a single platform and run multiple protocols. Below, we provide three use cases that show case different ways in which OpenClock could be used. Nonetheless, the usage of this testbed is not limited to these test cases.

Use Case 1: Effect of network attacks on synchronization error: When a network packet moves from client to server, the adversary in a network router can delay the packet. We refer to it as the *forward path attack*. The adversary can also delay the packet in the opposite direction i.e. when the packet moves from server to client in the network. We term it as *reverse path attack*. Multiple protocols are subjected to these attacks. In NTP, if an attacker is able to attack sufficient number of packets, it can manipulate the Marzullo's algorithm [9] to converge to a time desired by the attacker. We simulate both forward and reverse path attacks in OpenClock. We use these attacks to understand how NTP algorithms can be fooled. We run unmodified NTP on one timeline, and assume that the attacker is compromising more than half the packets from the NTP servers. By adding 2 second forward path delay to 4 out of 6 packets coming from 6 different NTP servers in the network, we slowly shift the timeline's notion of time away from the global time. We then modify NTP algorithm by adding a sanity check. This check sorts the timestamps from multiple servers based on their offsets, then carefully discards first half or the last half of NTP timestamps. We run the unmodified NTP on timeline1 and modified NTP on timeline2 at the same time processing the same attacked network packets. Our results in Figure 6 show that the attacks led to an increase in unmodified NTP's synchronization error, while the modified NTP experience less degradation in synchronization accuracy. So far in this work, we simulate network attacks and test for resilience against those attacks. We can also test for hardware attacks (temperature variations etc.) by simulating their effect in the testbed.

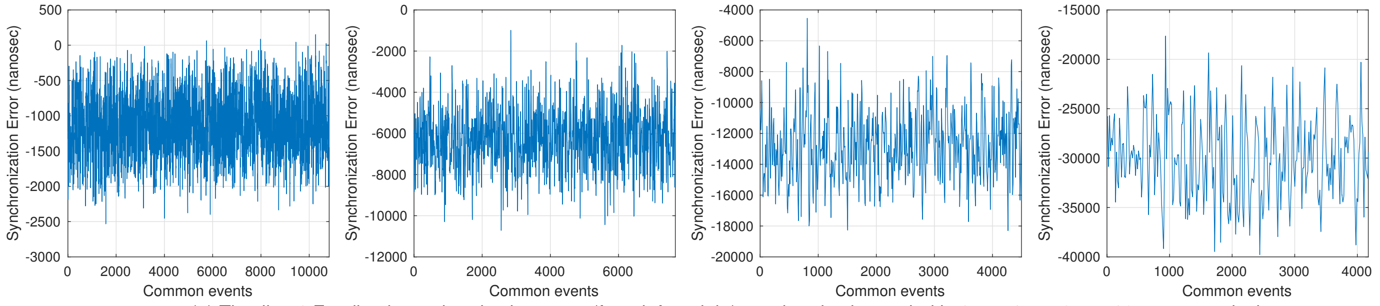


(a) Timeline 1: NTP without sanity check (b) Timeline 2: NTP with sanity check

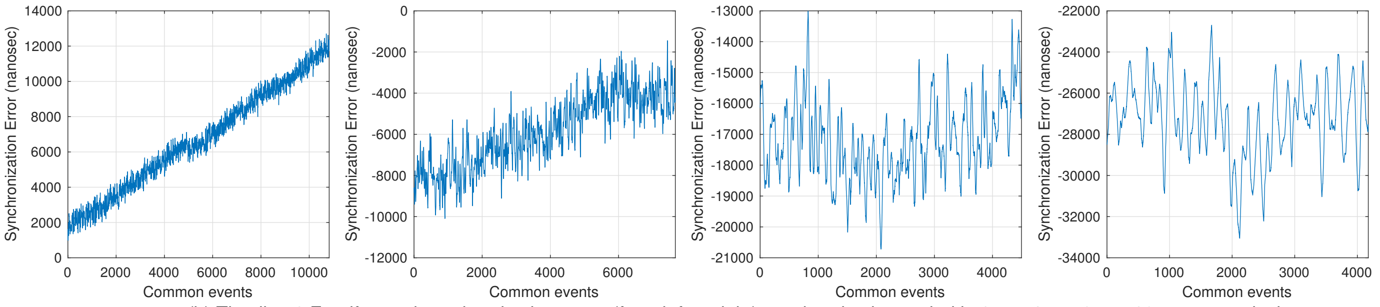
Fig. 6: Comparison of synchronization accuracies for unmodified NTP running in Timeline 1, and modified NTP running on Timeline 2 under network attacks. (a) Timeline 1 fluctuates and accumulates large error over time, while (b) Timeline 2 accumulates less error

Use Case 2: Effect of clock discipline mechanism on synchronization error: To understand which clock discipline mechanism enhances system performance, we run PTP on two timelines. Timeline 1 employs feedback discipline using PTP PI servo while Timeline 2 runs feedforward discipline using linear regression. Both timelines choose `SYS` for core clock and `PHC` for NICC. The results for both the timelines is shown in Figure 7. We explain the results with the next use case.

Use Case 3: Effect of synchronization parameters on error: There exists multiple tunable parameters in a synchronization protocol that can affect the performance. For example, to study the affect of synchronization period on the error, we run PTP with a feedback PI servo on two timelines. Both timelines use `SYS` for core clock and `PHC` for NICC. The only difference



(a) Timeline 1 Feedback synchronization error: (from left to right) synchronization period is 1sec, 4sec, 8sec, 16sec respectively



(b) Timeline 2 Feedforward synchronization error: (from left to right) synchronization period is 1sec, 4sec, 8sec, 16sec respectively

Fig. 7: Visualizing effect of two tunable parameters in this figure, the first parameter is clock discipline mechanism that changes from top to bottom row. The second parameter is synchronization period that changes from left to right. Synchronization error increases with an increase in synchronization period. This is evident in both (a) and (b) going from left to right. Comparison of feedforward and feedback for 1 second shows that feedback performs well for small synchronization periods. As we move to higher periods, the feedforward error tends to converge

between the two timelines is that Timeline 1 has a synchronization period of 1 second while Timeline 2 has a period of 4 seconds. The results in Figure 7 show that both feedback and feedforward have different impact on synchronization error with increase in period. Feedforward performs better at higher periods because it disciplines clock by measuring relative drift that can only be measured accurately over long durations.

V. Conclusion

When designing new time synchronization algorithms, they are compared with the established algorithms on distributed platforms. We argue that such comparisons are not fair as distributed platforms are subject to hardware and network variabilities. Moreover, many algorithms are not tested for attacks as it is hard to simulate same attacks on multiple platforms. We design and implement OpenClock, a real testbed that overcomes these limitations: (1) it provides multiple disciplinable clocks on a single platform to circumvent hardware variability, (2) it provides adjustable parameters for timelines to tune the synchronization performance. (3) OpenClock also presents an attack simulator that injects same attack to all algorithms under test. The benefit of an attack simulator is two folds: it can help find vulnerabilities in an algorithm, and help test resilience of algorithms under those attacks. OpenClock can be extended in multiple ways. Developers can write their own synchronization algorithms, present new dynamically tunable parameters, and define new hardware and network attacks. Our modular design based on clocks hierarchy and timelines provides programming and porting flexibility.

References

[1] A. Alanwar, F. M. Anwar, Y.-F. Zhang, J. Pearson, J. Hespanha, and M. B. Srivastava, "Cyclops: Pru programming framework for precise timing applications," in *Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 1–6.

[2] F. M. Anwar and M. B. Srivastava, "Precision time protocol over lr-wpan and 6lowpan," in *Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 1–6.

[3] F. Anwar, S. D'souza, A. Symington, A. Dongare, R. Rajkumar, A. Rowe, and M. Srivastava, "Timeline: An operating system abstraction for time-aware applications," in *Real-Time Systems Symposium (RTSS), 2016 IEEE*. IEEE, 2016, pp. 191–202.

[4] K.-T. Cho and K. G. Shin, "Fingerprinting electronic control units for vehicle intrusion detection." in *USENIX Security Symposium*, 2016, pp. 911–927.

[5] D. L. Mills, "Internet time synchronization: the network time protocol," *Communications, IEEE Transactions on*, vol. 39, no. 10, 1991.

[6] K. Lee, J. C. Eidson, H. Weibel, and D. Mohl, "Ieee 1588-standard for a precision clock synchronization protocol for networked measurement and control systems," in *Conference on IEEE*, vol. 1588, 2005, p. 2.

[7] R. Cochran and C. Marinescu, "Design and implementation of a ptp clock infrastructure for the linux kernel," in *Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2010 International IEEE Symposium on*. IEEE, 2010, pp. 116–121.

[8] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi, "The flooding time synchronization protocol," in *SenSys, Proceedings of the 2nd international conference on Embedded networked sensor systems*, 2004.

[9] K. Marzullo and S. Owicki, "Maintaining the time in a distributed system," *ACM SIGOPS Operating Systems Review*, vol. 19, no. 3, pp. 44–54, 1985.