Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization

Yilong Geng, Shiyu Liu, and Zi Yin, ***Stanford University;*** **Ashish Naik,** ***Google Inc.;***
Balaji Prabhakar and Mendel Rosenblum, ***Stanford University;*** **Amin Vahdat,** ***Google Inc.***

https://www.usenix.org/conference/nsdi18/presentation/geng

This paper is included in the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18).

April 9–11, 2018 • Renton, WA, USA

ISBN 978-1-931971-43-0

Open access to the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation is sponsored by USENIX.

Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization

Yilong Geng[1], Shiyu Liu[1], Zi Yin[1], Ashish Naik[2],
Balaji Prabhakar[1], Mendel Rosunblum[1], and Amin Vahdat[2]

[1]Stanford University, [2]Google Inc.

Abstract

Nanosecond-level clock synchronization can be an enabler of a new spectrum of timing- and delay-critical applications in data centers. However, the popular clock synchronization algorithm, NTP, can only achieve millisecond-level accuracy. Current solutions for achieving a synchronization accuracy of 10s-100s of nanoseconds require specially designed hardware throughout the network for combatting random network delays and component noise or to exploit clock synchronization inherent in Ethernet standards for the PHY.

In this paper, we present HUYGENS, a software clock synchronization system that uses a *synchronization network* and leverages three key ideas. First, coded probes identify and reject impure probe data—data captured by probes which suffer queuing delays, random jitter, and NIC timestamp noise. Next, HUYGENS processes the purified data with Support Vector Machines, a widely-used and powerful classifier, to accurately estimate one-way propagation times and achieve clock synchronization to within 100 nanoseconds. Finally, HUYGENS exploits a natural network effect—the idea that a group of pair-wise synchronized clocks must be transitively synchronized—to detect and correct synchronization errors even further.

Through evaluation of two hardware testbeds, we quantify the imprecision of existing clock synchronization across server-pairs, and the effect of temperature on clock speeds. We find the discrepancy between clock frequencies is typically 5-10μs/sec, but it can be as much as 30μs/sec. We show that HUYGENS achieves synchronization to within a few 10s of nanoseconds under varying loads, with a negligible overhead upon link bandwidth due to probes. Because HUYGENS is implemented in software running on standard hardware, it can be readily deployed in current data centers.

1 Introduction

Synchronizing clocks in a distributed system has been a long-standing important problem. Accurate clocks enable applications to operate on a common time axis across the different nodes, which, in turn, enables key functions like consistency, event ordering, causality and the scheduling of tasks and resources with precise timing. An early paper by Lamport [13] frames the question of ordering events in distributed systems and proposes a solution known for obtaining partial orders using "virtual clocks," and Liskov [16] describes many fundamental uses of synchronized clocks in distributed systems.

Our work is motivated by several compelling new applications and the possibility of obtaining very fine-grained clock sychronization at an accuracy and cost that is much less than provided by current solutions. For example, in finance and e-commerce, clock synchronization is crucial for determining transaction order: a trading platform needs to match bids and offers in the order in which they were placed, *even if they entered* the trading platform from different gateways. In distributed databases, accurate clock synchronization allows a database to enforce external consistency [8] and improves the throughput and latency of the database. In software-defined networks, the ability to schedule tasks with precise timing would enforce an ordering of forwarding rule updates so that routing loops can be avoided [18]. In network congestion control, the ability to send traffic during time slots assigned by a central arbiter helps achieve high bandwidth and near-zero queueing delays [28]. Indeed, precisely synchronized clocks can help to revise the "clockless" assumption underlying the design of distributed systems and change the way such systems are built.

Consider distributed databases as an example. Spanner [8] provides external consistency[1] at a global scale

[1]A database is said to be *externally consistent* if it can ensure for each transaction A that commits before another transaction B starts, A is serialized before B.

using clocks synchronized to within T units of time, typically a few milliseconds. In order to achieve external consistency, a write-transaction in Spanner has to wait out the clock uncertainty period, T, before releasing locks on the relevant records and committing. Spanner can afford this wait time because T is comparable to the delay of the two-phase-commit protocol across globally distributed data centers. However, for databases used by real-time, single data center applications, the millisecond-level clock uncertainty would fundamentally limit the database's write latency, throughput and performance. Thus, if a low latency database, for example, RAMCloud [24], were to provide external consistency by relying on clock synchronization, it would be critical for T to be in the order of 10s of nanoseconds so as not degrade the performance.

This relationship between clock synchronization and database consistency can been seen in CockroachDB[1], an open source scalable database. In CockroachDB, uncertainty about clocks in the system can cause performance degradation with read requests having to be retried. That is, a read issued by server A with timestamp t for a record at server B will be successful if the last update of the record at B has a timestamp s, where $s \leq t$ or $s > t + T$. Else, clock uncertainty necessitates that A retry the read with timestamp s. For example, in an experimental CockroachDB cluster of 32 servers we read 128 records, each updated every 25 ms. We found that as the clock uncertainty T was reduced from 1 ms to 10 us and then to 100 ns, the retry rate fell from 99.30% to 4.74% and to 0.08% in an experiment with 10,000 reads for each value of T.

Thus, while it is very desirable to have accurately synchronized clocks in distributed systems, the following reasons make it hard to achieve in practice. First, transaction and network speeds have shortened inter-event times to a degree which severely exposes clock synchronization inaccuracies. The most commonly used clocks have a quartz crystal oscillator, whose resonant frequency is accurate to a few parts per million at its ideal operating temperature of 25-28°C [34]. When the temperature at a clock varies (in either direction), the resonant frequency *decreases quadratically* with the temperature (see [34] for details). Thus, a quartz clock may drift from true time at the rate of 6-10 microseconds/sec. But the one-way delay (OWD), defined as the raw propagation (zero-queuing) time between sender and receiver, in high-performance data centers is under 10 μs. So, if the clocks at the sender and receiver are not frequently and finely synchronized, packet timestamps are rendered meaningless! Second, "path noise" has made the nanosecond-level estimation of the OWD, a critical step in synchronizing clocks, exceedingly difficult. Whereas large queuing delays can be determined and removed from the OWD calculation, path noise—due to small fluctuations in switching times, path asymmetries (e.g., due to cables of different length) and clock timestamp noise, which is in the order of 10s–100s of ns is not easy to estimate and remove from the OWD.

The most commonly used methods of estimating the OWD are the Network Time Protocol (NTP) [21], the Precision Time Protocol (PTP) [4], Pulse Per Second (PPS) [25]—a GPS-based system, and the recently proposed Data center Time Protocol (DTP) [14]. We review these methods is more detail later; for now, we note that they are either cheap and easy to deploy but perform poorly (NTP) or provide clock synchronization to an accuracy of 10s–100s of nanoseconds in data center settings but require hardware upgrades (PTP, DTP and PPS) which impose significant capital and operational costs that scale with the size of the network.

The algorithm we propose here, HUYGENS, achieves clock synchronization to an accuracy of 10s of nanoseconds at scale, and works with current generation network interface cards (NICs) and switches in data centers without the need for any additional hardware. A crucial feature of HUYGENS is that it processes the transmit (Tx) and receive (Rx) timestamps of probe packets exchanged by a pair of clocks *in bulk*: over a 2 second interval and simultaneously from multiple servers. This contrasts with PTP, PPS and DTP which look at the Tx and Rx timestamps of a single probe–ack pair individually (i.e., 4 timestamps at a time). By processing the timestamps in bulk, HUYGENS is able to fully exploit the power of inference techniques like Support Vector Machines and estimate both the "instantaneous time offset" between a pair of clocks and their "relative frequency offset". These estimates enable HUYGENS to be not bound by rounding errors arising from clock periods.

Contributions of the paper. The goal of our work is to precisely synchronize clocks in data centers, thereby making "timestamping guarantees" available to divers applications as a fundamental primitive alongside bandwidth, latency, privacy and security guarantees. We have chosen to synchronize clocks (e.g. the PTP Hardware Clocks, or PHCs [2]) in the NICs attached to servers. By accurately synchronizing NIC clocks, we obtain globally accurate timestamps for data, protocol messages and other transactions between different servers. NIC-to-NIC probes encounter the minimum amount of noise in the path propagation time as compared to server-to-server probes which also suffer highly variable stack latencies. Our main contributions are:

(1) A comprehensive and large-scale study of clock discrepancies in real-world networks. The major findings are: (i) pairwise clock rates can differ by as much as 30μs/sec; (ii) clock frequencies vary at time scales of minutes due to temperature effects, but are fairly constant

over 2–4 second intervals; and (iii) a quantification of the effect of queueing delays, path noise and path asymmetry on clock synchronization.
(2) The HUYGENS algorithm and its real-time extension HUYGENS-R, which can respectively be used by applications for aligning timestamps offline or for real-time clock synchronization.
(3) A NetFPGA-based verification in a 128-server, 2-stage Clos data center network shows that HUYGENS and HUYGENS-R achieve less than 12–15ns average error and under 30–45ns 99^{th} percentile error at 40% network load. At 90% load, the numbers increase to 16–23ns and 45–58ns, respectively.
(4) We propose a lightweight implementation of HUYGENS that runs "in-place" with the probe data. In a 40 Gbps data center testbed, HUYGENS only takes around 0.05% of the link bandwidth and less than 0.44% of a server's CPU time.

2 Literature survey

As mentioned in the Introduction, all methods of determining clock offsets involve estimating the OWD. In order to estimate the OWD between clocks A and B, A sends a probe packet to B containing the transmission time of the probe. The OWD can either be estimated directly by determining the time spent by the probe at each element en route from A to B (e.g., as in PTP), or by estimating the RTT (where B sends a probe back to A). In the latter case, assuming the OWD is equal in both directions, halving the estimated RTT gives the OWD. Using the estimate of the OWD and the probe's transmit time, B can work out the time at A and synchronize with it. We survey the four methods mentioned previously for estimating the OWD between a pair of clocks.
NTP. NTP [21] is a widely-used clock synchronization protocol. It estimates the offset between two clocks by considering multiple probe-echo pairs, picking the three with the smallest RTTs, and taking half their average to get the OWD. It achieves an accuracy of tens of milliseconds [23] to 10s of microseconds [26], depending on the network type (e.g., wide-area vs data center).

NTP uses simple methods to process the probe data, hence it only achieves a coarse-grained clock synchronization. HUYGENS does stronger processing of the same probe data to extract a much more refined estimate of the offset between a pair of clocks. It then uses the network effect to obtain a further 3x reduction in the estimation error.
PTP. PTP [4] uses hardware timestamps to counter stack delays. It uses "transparent" switches which are able to record the ingress and egress time of a packet to accurately obtain packet dwell times at switches. With more extensive hardware support at switches and a dedicated network for carrying PTP packets the White Rabbit system [22] can achieve sub-nanosecond precision. However, the accuracy in a conventional fully "PTP-enabled network" ranges from a few tens to hundreds of nanoseconds [32]. If the network is not fully PTP-enabled, synchronization accuracy can degrade by 1000x even when the two clocks are only a few hops apart [32]. Detailed tests conducted in [14] show that PTP performs poorly under high load, corroborating similar findings in [32].
DTP. The DTP protocol [14] sidesteps the issue of estimating time-varying queue sizes, stack times, and most noise variables by making a clever observation: The IEEE 802.3 Ethernet standards provide a natural clock synchronization mechanism between the transmitter and receiver PHYs at either end of a wire. Therefore, DTP can achieve a very fine-grained clock synchronization without increasing network traffic and its performance is not load-dependent. It is limited by the clock-granularity of the standard: for a 10Gbps network link the granularity is 6.4ns, and since four timestamps are involved in calculating OWD, a single hop synchronization accuracy of 25.6ns can be achieved. DTP requires special extra hardware at every PHY in the data center, necessitating a fully "DTP-enabled network" for its deployment.
PPS. PPS obtains accurate (atomic) time using a GPS receiver antenna mounted on the roof of the data center. It brings this signal to a multi-terminal distribution box using cables with precisely measured lengths. The multi-terminal box amplifies and relays the clock over cables (also with precisely known lengths) to NICs which are capable of receiving PPS signals. This makes PPS prohibitively expensive to deploy at scale, most installations have a designated "stratum 1" zone with just a few (typically tens of) servers that have access to PPS.

In summary, current methods of synchronizing clocks in a data center are either not accurate enough or require hardware modifications to almost every element of a data center, making them very expensive to deploy.

3 Our approach

The HUYGENS algorithm exploits some key aspects of modern data centers[2] and uses novel estimation algorithms and signal processing techniques. We look at these in turn.
Data center features. Most data centers employ a symmetric, multi-level, fat-tree switching fabric [30, 29]. By symmetry we mean that the number of hops between any pair of servers, A and B, is the same in both directions. We *do not require the paths to involve identically the*

[2]Even though this paper is focused on clock synchronization in data centers, we believe the principles extend to wide area networks, possibly with a loss in synchronization accuracy.

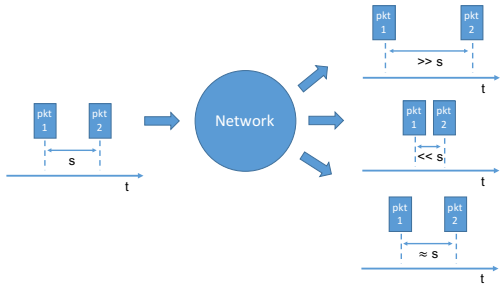


Figure 1: Coded probes

same switches in both directions.[3] Symmetry essentially equalizes the OWD between a pair of NICs in each direction, except for a small amount of "path noise" (quantified in Section 4). Furthermore, these propagation times are small, well-bounded by 25–30μs. Abundant bisection bandwidth and multiple paths between any pair of servers ensure that, even under 40-90% load, there is a reasonably good chance probes can traverse a network without encountering queueing delays. Finally, there are many servers (and NICs), making it possible to synchronize them in concert.

Algorithms and techniques. HUYGENS sets up a synchronization network of probes between servers; each server probes 10–20 others, regardless of the total number of servers in the network.

Coded probes. Naturally, probes which encounter no queueing delays and no noise on the path convey the most accurate OWDs. To automatically identify such probes, we introduce *coded probes*: a pair of probe packets going from server i to j with a small inter-probe transmission time spacing of s. If the spacing between the probe-pair when they are received at server j is very close to s, we deem them as "pure" and keep them both. Else, they are impure and we reject them. In Figure 1 the first two possibilities on the receiver side show impure probes and the third shows pure probes. Coded probes are very effective in weeding out bad probe data and they improve synchronization accuracy by a factor of 4 or 5.[4]

Support Vector Machines. The filtered probe data is processed by an SVM [9], a powerful and widely-used classifier in supervised learning. SVMs provide much more accurate estimates of propagation times between a pair of NICs than possible by the simpler processing methods employed by NTP and PTP. In Section 4 we shall see that "path noise" is small in magnitude and pathological in the sense that it has "negative-delay" components. Thus, simple techniques such as estimating the min-RTT or linear regression to process the probe data do not work. They can filter out large delays but cannot cope with small-magnitude path noise and are adversely affected by the negative-delay components which artifically shrink the OWD. The combination of coded probes and SVMs copes well with these problems.

Network effect.[5] Even though a data center network increases the path noise between clocks A and B because of multiple hops, it can simultaneously *increase the signal* by providing other clocks and new, potentially non-overlapping, paths for A and B to synchronize with them. Therefore, it is better—more accurate and scalable—to synchronize many clocks simultaneously than a pair of them at a time, as explained below. A significant by-product of using the network effect is that it is particularly good at detecting and correcting path asymmetries.

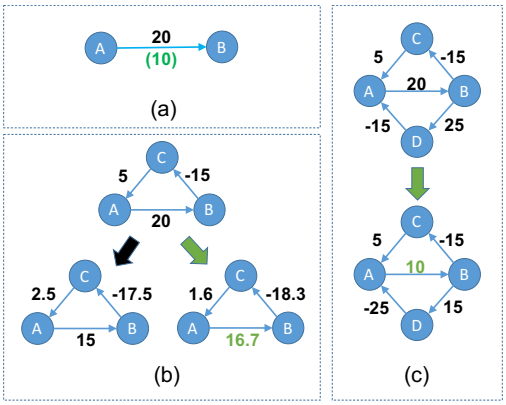


Figure 2: The Network Effect. How more clocks can help identify and reduce synchronization errors

Consider Figure 2. In (a), after pair-wise synchronization, clocks A and B believe that B is ahead of A by 20 units of time ($A \xrightarrow{20} B$). However, the truth (shown in green) is that B is ahead of A by only 10 units of time. A and B can never discover this error by themselves. In (b), a third clock C has undergone pairwise synchronization with A and B, and the resulting pairwise offsets are shown on the directed edges. Going around the loop A→B→C→A, we see that there is a *loop offset surplus* of 10 units! This immediately tells all three clocks there are errors in the pairwise synchronization. The bottom of Figure 2 (b) shows two possible corrections to the pairwise estimates to remove the loop surplus. Of these two choices, the HUYGENS algorithm will pick the one on the right, $A \xrightarrow{16.7} B \xrightarrow{-18.3} C \xrightarrow{1.6} A$. This is the *minimum-norm* solution and evenly distributes the loop surplus of 10 onto the 3 edges. Of course, if a fourth clock, D, joins the network, the two loop surpluses, equal to 10 and 30, are redistributed according to the minimum-norm solution by

[3] In Section 4 we show that a real-world 40 Gbps network has highly symmetric paths, for symmetry as defined here.

[4] The idea of using a pair of closely-spaced packets to determine the *available bandwidth* on a path was introduced in [12], see also [10]. Whereas that use case needs the separation between probes to increase in order to determine available bandwidth, we require no separation.

[5] Synchronization must be reflexive, symmetric and transitive; network effect is a deliberate exploitation of the transitivity property.

HUYGENS and this ends up correcting the discrepancy on edge A→B to its correct value of 10 units.

In general, the minimum-norm solution does not completely correct synchronization errors, nor does it evenly distribute loop surpluses. Its effect is to significantly pull in outlier errors (see Section 5).

The network effect is related to co-operative clock synchronization algorithms in the wireless literature [31, 15, 17, 19]. In the wireless setting, synchronization is viewed from the point of view of achieving consensus; that is, nodes try to achieve a global consensus on time by combining local information, and they use "gossip algorithms" to do so. Thus, a node exchanges information on time with neighboring nodes and this information gradually flows to the whole network with time synchronization as the outcome. This approach is appropriate in ad hoc wireless networks where nodes only know their neighbors. However, convergence times can be long and it can be hard to guarantee synchronization accuracy. Like [31], HUYGENS uses the network effect to directly verify and impose the transitivity property required of synchronization. In the wired setting in which HUYGENS operates, far away nodes are connected via the synchronization network, typically over a data center fabric. Thus, the probes have to contend with potentially severe network congestion, but the synchronization network can be chosen and not be restricted by network connectivity as in the ad hoc wireless case. The synchronization algorithm can be central and hence very fast (one-shot matrix multiplication—see Section 5) and yield provable synchronization error reduction guarantees.

4 Clocks in the real-world

In this section we use two testbeds and empirically study the degree to which pairs of clocks differ and drift with respect to one another, the effect of temperature in altering clock frequencies, and a characterization of queueing delays and path noise affecting the accurate measurement of the end-to-end propagation time of probes.

Testbed T-40. This is a 3-stage Clos network, all links running at 40Gbps. T-40 has 20 racks each with a top-of-the-rack (TOR) switch, and a total of 237 servers with roughly 12 servers per rack. There are 32 switches at spine layer 1. Each TOR switch is connected to 8 of these switches while each spine layer 1 switch is connected to 5 TOR switches. Thus, there is a 3:2 (12:8) oversubscription at the TOR switches. Each spine layer 1 switch is connected to 8 spine layer 2 switches and vice versa (there are 32 spine layer 2 switches). T-40 represents a state-of-the-art data center.

Testbed T-1. T-1 is a 2-stage Clos network with all links running at 1Gbps. It consists of 8 racks, each rack has a 1Gbps TOR switch and 16 *logical* servers. The 16 logical servers are built out of 4 Jetway network appliances (JNA) [11], 4 logical servers per JNA, as explained below. Each TOR switch has 16 downlinks and 8 uplinks, each uplink connecting it to one of 8 logically distinct spine switches. Thus, there is a 2:1 oversubscription at the TOR switches. The 8 logically distinct spine switches are built from 4 48-port 1Gbps physical switches using VLAN configurations [33]. T-1 respresents a low-end commodity data center.

For reasons of economy—in monetary, space and heat dissipation terms—we use JNAs to build servers in T-1. Each JNA has a 4-core Intel Celeron J1900 CPU, 8GB RAM, 250GB of disk storage, and ten 1Gbps Ethernet ports. Each Ethernet port has an Intel I211 NIC. The logical servers in a single JNA share the CPU, the RAM and the PCIe buses. Even though there can be 10 logical servers per JNA (one per NIC), to avoid overwhelming the CPU we build 4 logical servers per JNA. Each logical server is built inside a Docker container [20], giving them complete independence of operation. The servers implement a probing/responding application as well as various workload traffic generation applications. The different capabilities of T-40 and T-1 necessitated different probing and timestamping mechanisms, as explained below.

Probing. Recall that a probe is actually a pair of packets, called *coded probes*. We use 64-byte UDP packets for probing. Each server probes K other randomly chosen servers once every T seconds. Probing is bidirectional: servers which are probed send back probes every T seconds. In T-40, $K = 20$ and $T = 500\mu$s, and in T-1, $K = 10$ and $T = 4$ms.

Timestamping. The receive timestamps in T-40 and T-1 are recorded upon the receipt of the probes. In T-40, the transmit timestamp is equal to the time the probe's TX completion descriptor is written back into the host memory. The write-back time is often nearly equal to the transmission time, but, occasionally, it can be a few 10s or 100s of nanoseconds *after* the probe transmit time. This gives rise to a "negative-delay" timestamp noise; i.e., noise which can lead to probe propagation times *strictly smaller* than the absolute minimum possible. In T-1, the JNA's architecture makes the write-back approach perform much worse. Instead, the Intel I211 NIC in the JNA places the transmit start time of a probe in its payload and forwards it to the receiving NIC, where it is extracted.[6]

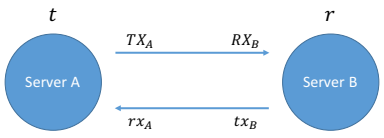


Figure 3: Signaling between clocks.

[6]This feature is not supported by the NICs in T-40.

Signaling. Consider Figure 3. Let T and R be the absolute times at which a probe was transmitted at NIC A and received at NIC B. Let TX_A and RX_B be the corresponding transmit and receive timestamps taken by the NICs. Define $\Delta_A = TX_A - T$ and $\Delta_B = RX_B - R$. Since $R > T$, we get $RX_B - \Delta_B > TX_A - \Delta_A$. Rearranging terms, we get

$$\Delta_B - \Delta_A < RX_B - TX_A. \quad (1)$$

From a probe (or echo) in the reverse direction, we get

$$tx_B - rx_A < \Delta_B - \Delta_A < RX_B - TX_A. \quad (2)$$

Thus, each probe gives either an upper bound or a lower bound on the discrepancy between two clocks depending on its direction; the tightest bounds come from probes encountering zero queueing delay and negligible noise. The discrepancy is time-varying due to different clock frequencies.

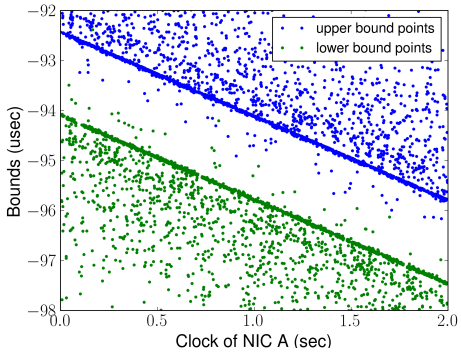


Figure 4: Bounds on the discrepancy between clocks in T-40

4.1 Clock frequencies across servers

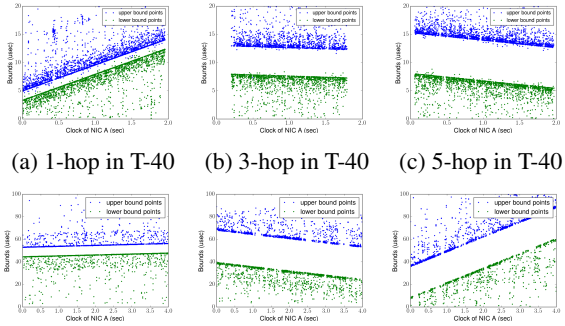
Figure 4 shows upper and lower bounds on $\Delta_B - \Delta_A$ in T-40 plotted against the time at NIC clock A: each blue dot is an upper bound and each green dot is a lower bound. The following observations can be made:

1. The set of blue points delineating the "least upper bound" of the data points lie on a straight line over short timescales, 2 seconds in this case. This line is parallel to the straight line on which the green dots delineating the "greatest lower bound" lie. We refer to the region between the lines as the "forbidden zone." There are a number of blue dots and green dots in the forbidden zone. These are due to the "negative-delay" NIC timestamp noise mentioned previously. It is crucial to filter out these points.

2. Since the dots on the lines bounding the forbidden zone capture the smallest one-way propagation time of a probe (equal only to wire and switching times), the spacing between them (roughly equal to 1700 ns in Figure 4) is the smallest RTT between the two NICs. Assuming symmetric paths in both directions, half of the spacing will equal the smallest one-way propagation time (roughly 850 ns).

3. The upper and lower bound lines have a non-zero slope and intercept. The slope in Figure 4 is close to -1.6μs/sec and it measures the "drift" in the frequencies of the clocks at A and B. That is, when clock A measures out one second of time, clock B would have measured out $1-(1.6\times10^{-6})$ second. The average of the two intercepts is the offset between the two clocks: when clock A's time is 0, clock B's time is roughly -93.3μs.

Remark. The slope captures the discrepancy in the clock frequencies and represents the intrinsic pull away from synchronism between the two clocks. *When the clocks are completely synchronized, the slope and the average of the two intercepts should both be 0.*

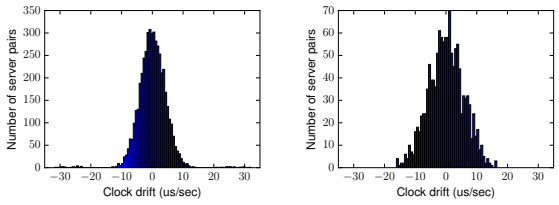


(a) 1-hop in T-40 (b) 3-hop in T-40 (c) 5-hop in T-40

(d) 1-hop in T-1 (e) 3-hop in T-1 (f) 3-hop in T-1

Figure 5: Examples of clock drifts

Figure 5 shows more examples of relative clock drifts in T-40 and T-1. Figures 6 (a) and (b) show the histograms of the drifts between pairs of clocks in T-40 and T-1, respectively. The number of pairs considered in each testbed and a numerical quantification of the data in Figure 6 is in Table 1. While most pair-wise clock drifts are around 6-10μs/sec, the maximum can get as high as 30μs/sec.



(a) T-40 (b) T-1

Figure 6: Distribution of the relative frequency difference between pairs of clocks

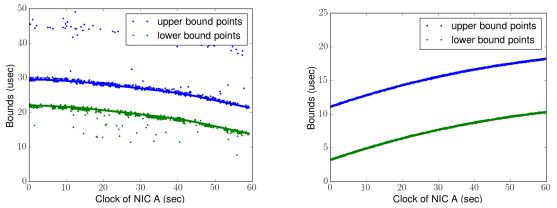
Variation in clock frequencies due to temperature

When considering longer timescales, in the order of minutes, one can sometimes observe nonlinearities in the upper and lower bound curves, see Figure 7. This is due

Testbed	#Servers	#Clock pairs	St.dev. of slope (drift)	Max abs. of slope (drift)
T-40	237	4740	4.5μs/sec	32.1μs/sec
T-1	128	1280	5.7μs/sec	16.5μs/sec

Table 1: Summary statistics of clock drift rates

to temperature changes which affect the resonance frequency of the clocks. The temperature can change when a NIC sends a lot of data or otherwise dissipates a lot of power. The temperature of a NIC varies slowly with time. Therefore, even though there is nonlinearity in the order of minutes or longer, at timescales of 2-4 seconds the upper and lower bound curves are essentially linear. We shall approximate the nonlinear curves by piecewise linear functions over 2-4 seconds.



(a) T-40 (b) T-1

Figure 7: Nonlinear drifts in clock frequencies

4.2 Queueing delays and path noise

1. **Queueing delay.** This is the total queueing delay on the path and can be 100s of microseconds to a few milliseconds, depending on the load. However, we shall see that even at 90% loads there are enough points with zero queueing delay (hence, on the boundary of the forbidden zone) to accurately synchronize clocks.
2. **Path noise.** We distinguish two types of path noise. *Switching noise.* Even under 0 load, a probe's switch-traversal time can be jittered due to random delays from passing through multiple clock domains, input arbitration, or other hardware implementation reasons specific to the switch (e.g., store-and-forward versus cut-through switching). "Dark matter packets" are another source of switch noise. These are protocol packets (e.g., spanning tree protocol (STP) [5], SNMP [3], link-layer discovery protocol (LLDP) [6]) not emitted by the end-hosts, hence invisible to them. When probes queue behind them at switches, small and random switching delays are added. *NIC timestamp noise.* This is the discrepancy between the timestamp reported by the NIC and the true time that the probe was transmitted or received. This can be caused by faulty hardware, or the way timestamping is implemented in the NIC. As described in Section 4, the latter can cause *negative-delay*, giving rise to points in the forbidden zone.

Empirically, path noise larger than 50ns degrades performance in T-40. In T-1 that number is 2μs. Packet transmission times in switches, in the PHYs, etc, are 40 times longer on T-1 than T-40, and this corresponds with an increase in noise magnitude.

4.3 Path symmetry

T-40 is a fat-tree network with 1, 3 and 5 hops between server pairs. By investigating the statistics of the RTTs between servers at different hop distances, we can test the veracity of our path symmetry assumption. Recall that path symmetry means that servers separated by a certain number of hops have more or less the same OWD, *regardless* of the particular paths taken in the forward and reverse direction to go between them.

	1 hop	3 hops	5 hops
# server-pairs	390	1779	6867
Ave. ZD-RTT	1570 ns	4881 ns	7130 ns
Min. ZD-RTT	1512 ns	4569 ns	6740 ns
Max. ZD-RTT	1650 ns	4993 ns	7253 ns

Table 2: Zero-delay-RTTs (ZD-RTTs) in T-40

In Table 2, we consider the zero-delay-RTT (ZD-RTT) between server pairs at different hops from one another. The forward and reverse routes between a pair of servers in T-40 are *arbitrary and not identical.* Nevertheless, we see from the minimum, average and maximum values of the ZD-RTT that it is tightly clustered around the mean. Since this was taken over a large number of server-pairs, we see empirical evidence supporting path symmetry.

If asymmetry exists in a network, it will degrade the overall synchronization accuracy by half the difference in the forward and reverse path delays. Fortunately, the network effect can be used to identify asymmetric paths and potentially replace them with symmetric paths (or paths that are less asymmetric).

5 The Huygens Algorithm

The Huygens algorithm synchronizes clocks in a network every 2 seconds in a "progressive-batch-delayed" fashion. Probe data gathered over the interval [0, 2) seconds will be processed during the interval [2, 4) seconds (hence batch and delayed). The algorithm completes the processing before 4 seconds and corrections can be applied to the timestamps at 1 sec, the midpoint of the interval [0, 2) seconds. Then we consider probe data in the interval [2, 4) seconds, and so on. By joining the times at the midpoints of all the intervals with straight lines, we obtain the corrections at all times. Thus, the corrected time is available as of a few seconds before the present

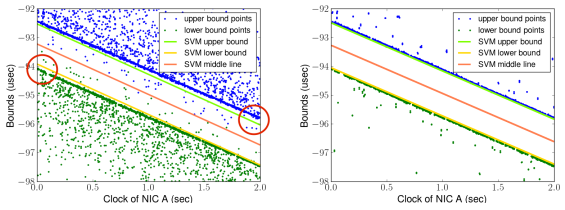
time (hence progressive).[7]

Coded probes. Presented informally previously, coded probes are a pair of packets, P_1 and P_2, with transmit timestamps t_1 and t_2 and receive timestamps r_1 and r_2, respectively. A coded probe is "pure" if $r_2 > r_1$ and $|(r_2 - r_1) - (t_2 - t_1)| < \varepsilon$, where $\varepsilon > 0$ is a prescribed guard band. Else, it is "impure". The timestamps of both of the pure probes are retained, and those of both impure probes are discarded. If either P_1 and P_2 is dropped, we discard the coded pair.

Support Vector Machines. SVMs are a widely used and powerful tool for linear and nonlinear classification in supervised learning settings [9]. A *linear SVM* is supplied with a collection of points (x_i, l_i) for $1 \le i \le N$, where x_i is a point in $\mathbb{R}^2$ and l_i is a binary label such as "upper bound point" or "lower bound point". It classifies points of like label, separating them with a hyperplane of maximal margin; that is, a hyperplane which is at a maximum distance from the closest point of either label in the data.

When used in our context, the SVM is given the upper and lower bound points derived from the probe data between clocks A and B over a 2-second interval. If the data is clean, i.e., if there are no points in the forbidden zone (defined in Section 4.1) and there are enough points with noise- and delay-free propagation time in both probing directions, the SVM will return a straight line with slope α_{AB} and intercept β_{AB}.

We use *soft-margin SVMs* which can tolerate points in the forbidden zone and other noise and delays. However, the performance of the SVM is sensitive to these artifacts, and *especially* to points in the forbidden zone. Therefore, even if the SVM returns a line in the noisy case, it will not be an accurate estimate of the discrepancy $\Delta_B - \Delta_A$. For this reason we first treat the data with the coded probe filter and extract pure probes to which we apply the SVM. For example, in Figure 8 (a) the SVM processes *all probe data* between a pair of clocks in T-40 and it is clear that its estimation of the upper and lower bound lines (the support vectors) are inaccurate. In (b) we see the significant improvement from using coded probes to filter out bad probe data.



(a) All coded probes (b) "Pure" coded probes

Figure 8: Effectiveness of coded probes. In T-40, coded probes reduce synchronization error by 4-5x.

[7] In Section 8 we obtain a real-time extension of the HUYGENS.

5.1 Network Effect

Suppose there are n clocks, $C_1, \ldots, C_n$, connected through a data center fabric. WLOG let C_1 be the "reference" clock with which all clocks will be synchronized. At time 0 a probe mesh is set up between the n clocks, each one probing K others. This is done using messages between the nodes and results in the "probing graph," $G = (V, E)$. The "owner" of edge (i, j) is the node who initiated the probing, ties broken at random. Once the probe mesh has been set up, we construct the Reference Spanning Tree (RST) on G with C_1 as the root in a breadth-first fashion.

In every 2-second interval, we synchronize C_i to C_1 at the midpoint of the interval.[8] The midpoints of adjacent intervals are then connected with straight lines to obtain a piecewise linear synchronization of C_i with C_1. Accordingly, consider the interval $[2j, 2(j+1))$ seconds for some $j \ge 0$. For ease of notation, denote $2j$ by L, $2(j+1)$ by R, and the midpoint $2j+1$ by M.

Iteration

1. *Coded probes and SVMs.* Timestamp data collected during $[L, R)$ is processed using the coded probes filter and bad probes are discarded. For each edge $(i, j) \in G$, where i is the owner of the edge, the filtered data is processed by an SVM to yield the slope, α_{ij}, and the intercept, β_{ij} of the hyperplane determined by the SVM. Let $(\vec{\alpha}, \vec{\beta})$ be the vectors of the $\alpha_{i,j}$ and $\beta_{i,j}$ for $(i, j) \in G$. The equations

$$\alpha_{ji} = \frac{-\alpha_{ij}}{1+\alpha_{ij}} \quad \text{and} \quad \beta_{ji} = \frac{-\beta_{ij}}{1+\alpha_{ij}} \tag{3}$$

relate the slopes and intercepts in one direction of (i, j) to the other.

2. *Preliminary estimates at time M.* Use the RST and the $(\vec{\alpha}, \vec{\beta})$ to obtain the preliminary, group-synchronized time at clock C_i with respect to the reference clock's time of M_1 sec. This is done as follows. First consider C_i to be a neighbor of C_1 on the RST. Then,

$$M_i^P = M_1 + \alpha_{1i} M_1 + \beta_{1i}.$$

Proceed inductively down the tree to obtain M_j^P at each clock C_j when C_1 equals M_1.

3. Obtain $\Delta_{ij}^P \stackrel{\triangle}{=} \alpha_{ij} M_i^P + \beta_{ij}$ for every i and j, with the convention $M_1^P = M_1$. Gather the Δ_{ij}^P into a vector Δ^P, the "preliminary pair-wise clock discrepancy vector."

4. *Network effect: loop correction.* Apply loop correction to Δ^P to determine the degree of inconsistency in the pair-wise clock estimates, and obtain Δ^F, the "final pair-wise clock discrepancy vector."

$$\Delta^F = \left[I - A^T \left(A A^T \right)^{-1} A \right] \Delta^P, \text{where the} \tag{4}$$

"loop-composition matrix", A, is defined below.

5. Obtain the final estimates M_i^F at C_i when the time at

[8] For concreteness, time instances are taken with reference to C_1.

C_1 equals M_1. For a neighbor C_i of C_1 on the RST:

$$M_i^F = M_1 + \Delta_{1i}^F.$$

Proceed inductively down the RST to obtain M_j^F at each clock C_j when C_1 equals M_1.

End Iteration

Elaboration of Steps 2–4

Steps 2 and 3. In Step 2 preliminary midpoint times are obtained using only the probe data on the edges of the RST. These midpoint estimates are used in Step 3 to get preliminary discrepancies, Δ^P, between the clock-pairs across all edges of the probing graph, G.

Step 4. Using equation (4), loop-wise correction is applied to Δ^P to obtain the final pair-wise clock discrepancy vector, Δ^F. Given $G = (V, E)$, the matrix A is obtained as follows. The number of columns of A equals $|E|$, each column corresponds with a *directed edge* $(i \to j) \in E$. Each row of A represents a loop of G, traversed in a particular direction. For example, suppose loop L is traversed along edges $(i \to j)$, $(j \to k)$, $(k \to l)$ and $(l \to i)$. Then, the row in A corresponding to loop L will be a 1 at columns corresponding to edges $(i \to j)$, $(j \to k)$ and $(k \to l)$, a -1 corresponding to column $(i \to l)$, and 0 elsewhere. The number of rows of A equals the largest set of *linearly independent* loops (represented in the row-vector form described) in G.

Derivation of equation (4). The quantity $A\Delta^P$ gives the total surplus in the preliminary pair-wise clock discrepancy in each loop of A. For example, for loop L defined above, this would equal:

$$\Delta_{ij}^P + \Delta_{jk}^P + \Delta_{kl}^P - \Delta_{il}^P \stackrel{\triangle}{=} y_L.$$

Let $Y = A\Delta^P$ represent the vector of loop-wise surpluses. In order to apply the loop-wise correction, we look for a vector N which also solves $Y = AN$ and posit the correction to be $\Delta^F = \Delta^P - N$. Now, A has full row rank, since the loops are all linearly independent. Further, since the number of linearly independent loops in G equals $|E| - |V| + 1$ which is less than $|E|$, the equation $Y = AN$ is under-determined and has multiple solutions. We look for the *minimum-norm solution* since this is most likely the best explanation of the errors in the loop-wise surpluses.[9] Since the pseudo-inverse, $N = A^T(AA^T)^{-1}Y = A^T(AA^T)^{-1}A\Delta^P$, is well-known to be the minimum-norm solution [27], we get

$$\Delta^F = \Delta^P - N = [I - A^T(AA^T)^{-1}A]\Delta^P,$$

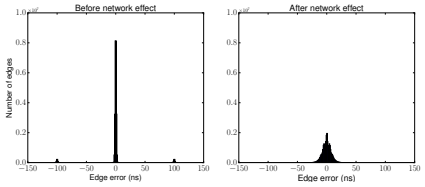
which is equation (4).[10]

[9] It is most likely that the loop-wise surpluses are due to a lot of small errors on the edges rather than a few large ones.

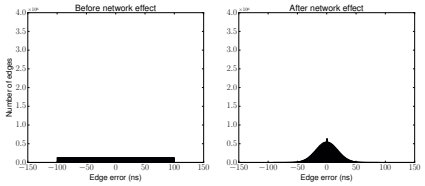
[10] Under a Gaussian assumption on the noise in the discrepancy vector Δ^P, we've shown that the standard deviation of the noise in Δ^F is a factor $\frac{1}{\sqrt{K}}$ of the noise in Δ^P. Numerically, this means a reduction of the errors by 68.4% and 77.6% when $K = 10$ and 20, respectively. Due to a shortage of space, we omit the proof here.

Remark. The minimum-norm solution gives the optimal (Maximum Likelihood) estimate of edge errors if they were distributed as independent and identically distributed (IID) Gaussians *before* applying the network effect. Even when edge errors are not IID Gaussians, the minimum-norm solution ensures that the edge errors after the network effect are closer to Gaussian with a much smaller variance ($\frac{1}{\sqrt{K}}$ of the pre-network-effect variance). Most importantly, this is true whether the initial edge errors occured due to significant path asymmetries (which can be large and systematic) or due to path noise (which is typically small and nearly zero mean).

Figure 9 illustrates the point. In a network of 256 nodes, each probing 10 other nodes, we see the network effect reduces the standard deviation of edge errors (after coded probes and SVMs) in two cases: (a) the edge errors are typically small but some times can be as large as 100 ns, and (b) distribted uniformly between -100 ns and 100 ns. In both cases the errors after appying the minimum-norm solution are clustered around 0 in a bell-shaped distribution.



(a) Network effect eliminates large edge errors



(b) Network effect compresses uniformly distributed edge errors

Figure 9: Examples of network effect: N=256, K = 10, collective results across 10000 runs

6 Implementation and Evaluation

The probing phase of HUYGENS is obviously distributed: it must occur along the edges of G. Many of the other basic operations can also by implemented distributedly at the sensing nodes, hence significantly reducing data movement overheads. Or, they can be implemented on a dedicated computing system separate from the sensing system. We describe the distributed implementation.

6.1 A lightweight, scalable implementation

We implement HUYGENS as an "in-place" distributed system, making it lightweight and scalable. There are

three main components—the master, the slaves and the sensors. A global master node initiates the probing phase and runs loop-wise correction (Steps 2–5). There is a sensor and a slave node at each server for conducting probing, data collection, coded probe filtering and running SVMs. The sensor sends the probes and collects timestamps, the slave runs all the computations. This implementation eliminates overheads due to data transfer and makes HUYGENS scalable because each sensor–slave combination only needs to send the α- and β-values to the master node and *not the probe data*.

Probing bandwidth and CPU overhead. Probing at the rates we have implemented (see Probing in Section 4), HUYGENS takes 0.05% of the bandwidth at each node in T-40 and 0.25% of the bandwidth at each node in T-1. The fact that HUYGENS takes a much smaller percentage of the bandwidth in T-40 than in T-1 is due to an important property: it sends roughly the same number of probes per unit time regardless of the line rate.

The CPU overhead imposed by HUYGENS is negligible. On an 2.8 GHz Intel i5 (5575R) CPU, using only *one core* and running SVM for 2 seconds probe data from one edge takes less than 7ms. Therefore, the in-place distributed implementation would take less than 0.44% of CPU time in a modern 32-core server even when $K = 20$.

6.2 Evaluation

We run experiments on T-1 to evaluate: (i) the accuracy of HUYGENS using NetFPGAs, (ii) the efficacy of the network effect, and (iii) HUYGENS' performance under very high network load.

Network load. We use the traffic generator in [7]: each server requests files simultaneously from a random number (30% 1, 50% 2 and 20% 4) of other servers, called the "fanout" of the request. The gap between adjacent requests are independent exponentials with load-dependent rate. The file sizes are distributed according to a heavy-tailed distribution in the range [10KB, 30MB] with an average file size of 2.4MB. This traffic pattern can mimic a combination of single flow file transfers (fanout = 1) and RPC style incast traffic (fanout > 1).

Evaluation Methodology

NetFPGA-CML boards provide two natural ways to verify HUYGENS' accuracy in synchronizing clocks:

(i) **Single FPGA.** Each NetFPGA has four 1GE ports connected to a *common clock*. We take two of these ports, say P_1 and P_2, and make them independent servers by attaching a separate Docker container to each. P_1 and P_2 then become two additional servers in the 128-server T-1 testbed. Using HUYGENS we obtain the "discrepancy" in the clocks at P_1 and P_2, whereas the ground truth is that there is 0 discrepancy since P_1 and P_2 have the same clock.

Remark. To make it more difficult for HUYGENS to synchronize P_1 and P_2, we do not allow them to directly probe each other. They are at least 2 or 3 hops away on the RST.

(ii) **Different FPGAs.** This time P_1 and P_2 are ports on different FPGAs, with clocks C_1 and C_2, say. They can be synchronized using HUYGENS on the T-1 testbed. They can also be synchronized using a *direct channel* by connecting the GPIO pins on the two NetFPGAs using copper jumper wires. The direct channel provides us with an essentially *noise- and delay-free, short RTT*[11] signaling method between C_1 and C_2. The pin-to-pin signals are sent and echoed between the NetFPGAs every 10ms. We process the TX and RX timestamps of the pin-to-pin signals using a linear regression and obtain the discrepancy between C_1 and C_2 using the direct channel.

In both (i) and (ii), the HUYGENS probe data is processed using the algorithm described in Section 5. We take C_1 (the clock at P_1) as the root of the RST. C_2 is then another node on the RST and HUYGENS gives a *preliminary* and a *final estimate* of its midpoint in a 2 second probing interval with respect to C_1's midpoint (as described in Steps 2–5). This is compared to the ground truth or direct channel discrepancy between C_1 and C_2. Even though (i) gives us a ground truth comparison, we use (ii) because it gives a sense of the degree of synchonization possible between two different clocks that are connected directly.

7 Verification

We consider a 10-minute experiment.[12] For the single FPGA comparison, we obtain HUYGENS' preliminary and final estimates of the midpoint times at C_2 in successive 2-second intervals with respect to C_1. We compare these estimates with the ground truth discrepancy, which is 0. There are 300 2-second intervals in total; we report the average and the 99^{th} percentile discrepancies. In the case of different FPGAs, we compare the estimates from HUYGENS with the estimate from the direct channel. Table 3 contains the results.

	Single NetFPGA		Different NetFPGAs	
	Prelim	Final (net effect)	Prelim	Final (net effect)
Mean of abs. error (ns)	41.4	11.2	47.8	13.4
99^{th} percentile of abs. error (ns)	91.0	22.0	92.1	30.2

Table 3: HUYGENS synchronization accuracy: 16-hour experiment in T-1 at 40% load with $K = 10$

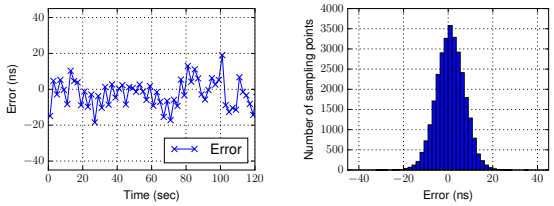
We see that the network effect (loop-wise correction) is quite powerful, yielding a 3-4x improvement in

[11] The RTT between the FPGAs is 8 clock cycles (64ns) or smaller.

[12] Section 8 presents results from a 16 hour run.

the accuracy of clock synchronization. The most compelling demonstration is the single FPGA comparison in which the ground truth is unambiguously 0. The different FPGA comparison shows HUYGENS with loop correction obtains the same performance as an almost error-free, direct connection between the two clocks even though in HUYGENS the clocks are connected through a network of intermediaries.

Figure 10 shows a 2 minute sample trajectory of the errors as well as the distribution of the errors over 10 minutes in the same FPGA comparison.

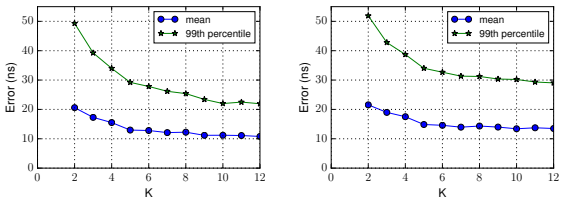


(a) A 2 min zoom-in (b) Distribution of errors

Figure 10: HUYGENS vs ground truth for 2 ports on the same NetFPGA: 16-hour experiment at 40% load with $K = 10$

7.1 The network effect

We consider the benefit of the network effect by increasing K from 2 to 12. Under an assumed theoretical model, it was stated in Section 5 that this would reduce the standard deviation of the clock synchronization error by a factor $\frac{1}{\sqrt{K}}$. Figure 11 quantifies the benefit of the network effect for the mean of the absolute error and the 99^{th} percentile errors as K gets larger.

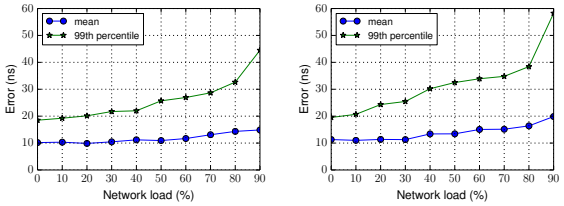


(a) Same NetFPGA (b) Different NetFPGAs

Figure 11: Mean and 99^{th} percentile synchronization error in T-1 as K varies with 40% load

7.2 Performance under high load

Figure 12 shows the accuracy of HUYGENS under different network loads. As can be seen, even at 90% load HUYGENS is still able to synchronize clocks with 99^{th} percentile error smaller than 60 nanoseconds. HUYGENS is robust to very high network load for the following reasons: (i) It applies intensive statistical procedures on each 2 seconds of probe data; the 2 second interval happens to be long enough that, even at very high load, a small number of probes go though empty queues, allowing HUYGENS to obtain accurate estimates. (ii) HUYGENS takes advantage of the redundant connectivity in the network: even if one probing pair is blocked due to congestion, the two clocks in this probing pair will still quite likely be able to reach other clocks and synchronize with them. (iii) Loop-wise correction is able to identify and compensate probing pairs which are badly affected by high load. We couldn't load T-1 at more than 90% because the switches in T-1 have shallow buffers and drop large numbers of packets, leading to excessive TCP timeouts.



(a) Same NetFPGA (b) Different NetFPGAs

Figure 12: HUYGENS error at different loads in T-1, $K = 10$.

7.3 Comparision with NTP

Since NTP is a widely-used algorithm and does not require specialized hardware, it is worth comparing it to HUYGENS on T-1. For fairness, we let NTP use the same hardware timestamps as HUYGENS, although almost all real-world implementations of NTP use CPU timestamps.

Load	Method	Single NetFPGA		Different NetFPGAs	
		Mean of abs. error (ns)	99^{th} percentile of abs. error (ns)	Mean of abs. error (ns)	99^{th} percentile of abs. error (ns)
0%	HUYGENS	10.2	18.5	11.3	19.5
	NTP	177.7	558.8	207.8	643.6
40%	HUYGENS	11.2	22.0	13.4	30.2
	NTP	77975.2	347638.4	93394.0	538329.9
80%	HUYGENS	14.3	32.7	16.4	38.4
	NTP	211011.7	778070.4	194353.5	688229.1

Table 4: A comparison of HUYGENS and NTP.

Table 4 shows that, even with hardware timestamps, NTP's error is 4 orders of magnitude larger than HUYGENS's under medium to high network loads. Note that although some implementations of NTP use a DAG-like graph to synchronize a clock with clocks which are upstream on the DAG, this operation is local when compared to the more global loop-wise correction step in the network effect.

8 Real-time Huygens

We now extend HUYGENS to get a real-time version, HUYGENS-R. Underlying this extension is the empiri-

cal observation in Section 4 that clock frequencies are slowly varying. Therefore, a linear extrapolation of the estimates of HUYGENS over a few seconds will yield a reasonably accurate real-time clock synchronization algorithm.

The extension is best described using Figure 13. Consider clocks C_1 and C_i. Let I_k be the time interval $[2(k-1), 2k)$. Step 5 of the HUYGENS algorithm yields the final offset of the midpoint of each I_k at C_i from the midpoint of the same interval at C_1. For example, T_2+3 is the time at C_i when the time at C_1 is 3 seconds; in general, $T_k+(2k-1)$ is the time at C_i when the time at C_1 is $2k-1$ seconds. By connecting the T_k with straight lines, we get the HUYGENS offsets between C_i and C_1 at all times.

The green points, O_l, are the offsets between clocks C_i and C_1 for the real-time version. They are obtained as shown in the figure: O_l lies on the line from T_l to T_{l+1} when the time at C_1 equals $2l+6$. Thus, O_1 is on the line from T_1 to T_2 when C_1's time equals 8, etc. Connect the successive points O_l and O_{l+1} using straight lines to obtain the green curve. This curve gives the HUYGENS-R offsets between clocks C_i and C_1 at all times after 8 seconds. Since HUYGENS-R is an extrapolation of HUYGENS, it is not defined until some time after HUYGENS has been operational. As defined above, the earliest time at which HUYGENS-R can provide synchronized clocks is when C_1's time equals 8 seconds.

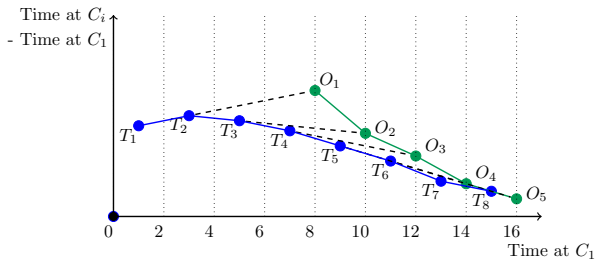
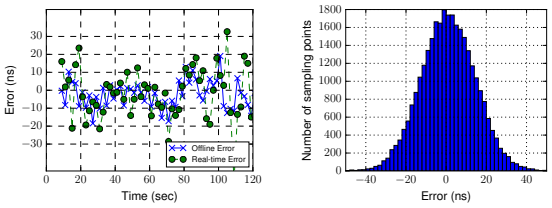


Figure 13: From HUYGENS to HUYGENS-R

Figure 14 quantifies the accuracy of HUYGENS-R using a single NetFPGA. In part (a) of the figure we see that while HUYGENS-R's errors are worse than HUYGENS' errors, they are not much worse. Part (b) of the figure gives a distribution of the errors. Under 40% load, the average value and the 99^{th} percentile of the absolute error are 14.1ns and 43.5ns for HUYGENS-R, slightly larger than the corresponding numbers of 11.0ns and 22.7ns for HUYGENS. These numbers increase to 22.1ns and 55.0ns respectively, versus 14.3ns and 32.7ns for HUYGENS under 80% network load.

Deployment. Even though HUYGENS and HUYGENS-R work effectively at 90% load, the desire for "high availability" in practice may require the use of a dedicated IEEE 802.1 QoS priority for the probes. This not only isolates probe traffic (thereby ensuring that HUYGENS and HUYGENS-R run at all loads), but by giving probe traffic the highest priority, we can also reduce the effect of queueing delays.



(a) A 2 min zoom-in (b) Distribution of errors

Figure 14: HUYGENS-R vs HUYGENS: 16-hour experiment at 40% load with $K = 10$

9 Conclusion

In this paper, we investigated the practicality of deploying accurate timestamping as a primitive service in data center networks, in support of a number of higher-level services such as consistency in replicated databases and congestion control. We set out to achieve synchronization accuracy at the granularity of tens of nanoseconds for all servers in a data center with low overhead among a number of dimensions, including host CPU, network bandwidth, and deployment complexity. When compared with existing approaches to clock synchronization, we aimed to support currently functioning data center deployments with no specialized hardware, except NICs that support hardware timestamping (e.g., PHC). With these goals, we introduced HUYGENS, a probe-based, end-to-end clock synchronization algorithm. By using coded probes, Support Vector Machines and the network effect, HUYGENS achieves an accuracy of 10s of nanoseconds even at high network load. HUYGENS can scale to the whole data center since each server only needs to probe a constant number (10–20) of other servers and the resulting data can be processed in-place. In particular, the parameters and the 2-second update times HUYGENS uses remain the same regardless of the number of servers. In a 40 Gbps data center testbed HUYGENS only consumes around 0.05% of the servers bandwidth and less than 0.44% of its CPU time. Since it only requires hardware timestamping capability which is widely-available in modern NICs, HUYGENS is ready for deployment in current data centers. We are currently exploring the performance of HUYGENS in wide area settings and are seeing promising results.

References

[1] CockroachDB. `https://github.com/cockroachdb/cockroach`. Accessed: 2017-9-22.

[2] PTP hardware clock infrastructure for Linux. `https://www.kernel.org/doc/Documentation/ptp/ptp.txt`. Accessed: 2017-9-22.

[3] Simple Network Management Protocol. `https://en.wikipedia.org/wiki/Simple_Network_Management_Protocol`. Accessed: 2017-9-22.

[4] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Standard 1588* (2008).

[5] IEEE Standard for Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks–Amendment 20: Shortest Path Bridging. *IEEE Standard 802.1AQ* (2012).

[6] IEEE Standard for Local and metropolitan area networks - Station and Media Access Control Connectivity Discovery. *IEEE Standard 802.1AB* (2016).

[7] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center TCP (DCTCP). In *ACM SIGCOMM computer communication review* (2010), vol. 40, ACM, pp. 63–74.

[8] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W. C., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems 31*, 3 (2013), 8.

[9] CORTES, C., AND VAPNIK, V. Support-vector networks. *Machine learning 20*, 3 (1995), 273–297.

[10] JAIN, M., AND DOVROLIS, C. Ten fallacies and pitfalls on end-to-end available bandwidth estimation. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement* (2004), ACM, pp. 272–277.

[11] JETWAY COMPUTER CORPORATION. Cableless & Fanless Embedded Barebone Celeron J1900 / 10 Intel Gigabit LAN . `http://www.jetwaycomputer.com/spec/JBC390F541AA.pdf`, 11 2016. Accessed: 2017-9-22.

[12] KESHAV, S. *The packet pair flow control protocol*. International Computer Science Institute, 1991.

[13] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[14] LEE, K. S., WANG, H., SHRIVASTAV, V., AND WEATHERSPOON, H. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference* (2016), ACM, pp. 454–467.

[15] LENG, M., AND WU, Y.-C. Distributed clock synchronization for wireless sensor networks using belief propagation. *IEEE Transactions on Signal Processing 59*, 11 (2011), 5404–5414.

[16] LISKOV, B. Practical uses of synchronized clocks in distributed systems. *Distributed Computing 6*, 4 (1993), 211–219.

[17] MAGGS, M. K., O'KEEFE, S. G., AND THIEL, D. V. Consensus clock synchronization for wireless sensor networks. *IEEE sensors Journal 12*, 6 (2012), 2269–2277.

[18] MAHAJAN, R., AND WATTENHOFER, R. On consistent updates in software defined networks. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks* (2013), ACM, p. 20.

[19] MALLADA, E., MENG, X., HACK, M., ZHANG, L., AND TANG, A. Skewless network clock synchronization without discontinuity: Convergence and performance. *IEEE/ACM Transactions on Networking (TON) 23*, 5 (2015), 1619–1633.

[20] MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal 2014*, 239 (2014), 2.

[21] MILLS, D. L. Internet time synchronization: the network time protocol. *IEEE Transactions on communications 39*, 10 (1991), 1482–1493.

[22] MOREIRA, P., SERRANO, J., WLOSTOWSKI, T., LOSCHMIDT, P., AND GADERER, G. White rabbit: Sub-nanosecond timing distribution over ethernet. In *2009 International Symposium on Precision Clock Synchronization for Measurement, Control and Communication* (2009), IEEE, pp. 1–5.

[23] MURTA, C. D., TORRES JR, P. R., AND MOHAPATRA, P. QRPp1-4: Characterizing Quality of Time and Topology in a Time Synchronization Network. In *IEEE Globecom 2006* (2006), IEEE, pp. 1–5.

[24] OUSTERHOUT, J. K., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G. M., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for ramclouds: scalable high-performance storage entirely in dram. *Operating Systems Review 43*, 4 (2010), 92–105.

[25] PARKINSON, B. W. *Progress in astronautics and aeronautics: Global positioning system: Theory and applications*, vol. 2. AIAA, 1996.

[26] PÁSZTOR, A., AND VEITCH, D. Pc based precision timing without gps. In *ACM SIGMETRICS Performance Evaluation Review* (2002), vol. 30, ACM, pp. 1–10.

[27] PENROSE, R. A generalized inverse for matrices. In *Mathematical proceedings of the Cambridge philosophical society* (1955), vol. 51, Cambridge Univ Press, pp. 406–413.

[28] PERRY, J., OUSTERHOUT, A., BALAKRISHNAN, H., SHAH, D., AND FUGAL, H. Fastpass: A centralized zero-queue datacenter network. In *ACM SIGCOMM Computer Communication Review* (2014), vol. 44, ACM, pp. 307–318.

[29] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review* (2015), vol. 45, ACM, pp. 123–137.

[30] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., KANAGALA, A., PROVOST, J., SIMMONS, J., TANDA, E., WANDERER, J., HLZLE, U., STUART, S., AND VAHDAT, A. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), vol. 45, pp. 183–197.

[31] SOLIS, R., BORKAR, V. S., AND KUMAR, P. A new distributed time synchronization protocol for multihop wireless networks. In *Decision and Control, 2006 45th IEEE Conference on* (2006), IEEE, pp. 2734–2739.

[32] WATT, S. T., ACHANTA, S., ABUBAKARI, H., SAGEN, E., KORKMAZ, Z., AND AHMED, H. Understanding and applying precision time protocol. In *2015 Saudi Arabia Smart Grid (SASG)* (2015), IEEE, pp. 1–7.

[33] YUASA, H., SATAKE, T., CARDONA, M. J., FUJII, H., YASUDA, A., YAMASHITA, K., SUZAKI, S., IKEZAWA, H., OHNO, M., MATSUZAKI, A., ET AL. Virtual LAN system, July 4 2000. US Patent 6,085,238.

[34] ZHOU, H., NICHOLLS, C., KUNZ, T., AND SCHWARTZ, H. Frequency accuracy & stability dependencies of crystal oscillators. *Carleton University, Systems and Computer Engineering, Technical Report SCE-08-12* (2008).