Sundial: Fault-tolerant Clock Synchronization for Datacenters

Yuliang Li[*†], Gautam Kumar*[*], Hema Hariharan*[*], Hassan Wassel*[*], Peter Hochschild*[*], Dave Platt*[*],*
Simon Sabato[‡], Minlan Yu*[†], Nandita Dukkipati*[*], Prashant Chandra*[*], Amin Vahdat*[*]*
Google Inc.[*], Harvard University*[†], Lilac Cloud*[‡]

Abstract

Clock synchronization is critical for many datacenter applications such as distributed transactional databases, consistent snapshots, and network telemetry. As applications have increasing performance requirements and datacenter networks get into ultra-low latency, we need submicrosecond-level bound on time-uncertainty to reduce transaction delay and enable new network management applications (e.g., measuring one-way delay for congestion control). The state-of-the-art clock synchronization solutions focus on improving clock precision but may incur significant time-uncertainty bound due to the presence of failures. This significantly affects applications because in large-scale datacenters, temperature-related, link, device, and domain failures are common. We present Sundial, a fault-tolerant clock synchronization system for datacenters that achieves ∼100ns time-uncertainty bound under various types of failures. Sundial provides fast failure detection based on frequent synchronization messages in hardware. Sundial enables fast failure recovery using a novel graph-based algorithm to precompute a backup plan that is generic to failures. Through experiments in a >500-machine testbed and large-scale simulations, we show that Sundial can achieve ∼100ns time-uncertainty bound under different types of failures, which is more than two orders of magnitude lower than the state-of-the-art solutions. We also demonstrate the benefit of Sundial on applications such as Spanner and Swift congestion control.

1 Introduction

Clock synchronization is increasingly important for datacenter applications such as distributed transactional databases [12, 32], consistent snapshots [11, 16], network telemetry, congestion control, and distributed logging.

One key metric for clock synchronization is the *time-uncertainty bound* for each node, denoted as ε in this paper, which bounds the difference between local clock and other clocks. This concept is used by TrueTime in Spanner [12]. Spanner leverages TrueTime to guarantee the correctness properties around concurrency control and provide consistency in distributed databases. Another example is consistent snapshots, which are commonly used for debugging or handling failures in distributed systems. To ensure consistency among snapshots, each node needs to wait for its time-uncertainty bound (ε) before recording the states.

Traditional clock synchronization techniques provide ε at the millisecond level (e.g., <10ms in TrueTime [12]), which is no longer effective for modern datacenter applications with increasing performance requirements and ultra low latency datacenter networks (e.g., with latency around 5μs [25]). Today's applications can benefit significantly from submicrosecond-level ε. For example, FaRMv2 [32], an RDMA-based transactional system, observes the median transaction delay can drop by 25% if we improve ε from ∼20μs to 100ns. CockroachDB [3] can significantly reduce the retry rate when ε drops from 1ms to 100ns based on an experiment in [13].

Providing submicrosecond-level ε can also enable new network management applications. For example, with submicrosecond-level clock differences across devices, we can measure one-way delay, locate packet losses, and identify per-hop latency bursts [23, 24]. It also enables synchronized network snapshots [37] which are useful for identifying RTT-scale network imbalance and collect global forwarding state. Accurate one-way delay provides a better congestion signal to delay-based congestion control [17, 29] to differentiate between forward and reverse path congestion.

There are several systems that achieve submicrosecond-level clock precision. The state-of-the-art commercial solution on precise clock synchronization is Precision Time Protocol (PTP) [4]. PTP is widely available in switches and NICs [6,8,9]. Each switch or NIC has a hardware clock driven by an oscillator, generates timestamped synchronization messages *in software*, and sends them over a spanning tree to synchronize with other nodes. Normally, oscillator drifts stay within ±100μs per second and the devices synchronize every 15ms to 2 seconds [4, 8]. A recent proposal DTP [21] sends messages in the physical layer every few microseconds and can also achieve ∼100ns precision. Huygens [13] is a clock-synchronization system built in software that achieves <100ns

precision by using Support Vector Machines to accurately estimate one-way propagation delays.

While these works provide high clock precision under normal cases, the time-uncertainty bound ε grows to 10-100s of μs as datacenters are subject to a variety of failures. In large-scale datacenters, there are common temperature-related failures which affect oscillator drifts. There are also frequent link, device, and domain failures (i.e., a domain of links and devices that fail together) that affect the synchronization across nodes (see §3).

In this paper, we present Sundial, which provides ∼100ns time-uncertainty bound (ε) under failures including temperature-related, link, device and domain failures and reports ε to applications – two orders of magnitude better than current designs. Even in cases of simultaneous failures across domains, Sundial provides correct ε to applications. Sundial achieves this with a hardware-software codesign that enables fast failure detection and recovery:

Fast failure detection based on frequent synchronous messaging on commodity hardware: Sundial exchanges messages every ∼100μs in hardware without changing the physical layer. The frequent message exchange enables fast failure detection and recovery, and frequent reduction of ε. To ensure fast failure detection for remote nodes in the spanning tree, Sundial introduces *synchronous* messaging which ensures that each node sends a new message only when it receives a message from the upstream.

Fast failure recovery with precomputed backup plan that is generic to all types of failures: To enable fast failure recovery, Sundial controller precomputes a backup plan consisting of one backup parent for each node and a backup root, so that each device can recover locally. The backup plan is generic to different types of failures (i.e., link, device failures, root failures, and domain failures) and ensures that after failure recovery, the devices remain connected without loops. We introduce a new search algorithm for the backup plan that extends a variant of edge-disjoint spanning tree algorithm [35] but with additional constraints such as no-ancestor condition (the edge in the current tree cannot be a forward edge in the backup tree) and disjoint-failure-domain condition (no domain failure can take down both the parent and the backup parent for any device). Our algorithm only takes 148ms on average to run on an example Jupiter [33] topology with 88K nodes.

We evaluate Sundial with experiments in a >500 machine prototype implementation and via large-scale simulations. Sundial achieves ∼100ns time-uncertainty bound both under normal time and under different types of failures, which is more than two orders of magnitude lower than the state-of-the-art solutions such as PTP [4], Huygens [13], and DTP [21]. Sundial reduces the commit-wait latency of Spanner [12] running inside a datacenter by 3-4x, and improves the throughput of Swift congestion control [17] by 1.6x under reverse-path congestion.

2 Need for *Tight* Time-uncertainty Bound

A clock synchronization system for datacenters need not only a current value of time but also time-uncertainty bound that is used by applications for correctness as well as performance. We describe several datacenter applications and how tight time-uncertainty bound benefits them below.

Distributed Transactional Databases: Spanner [12], FaRMv2 [32] and CockroachDB [3] are some examples of distributed databases deployed at scale in production that directly use time-uncertainty bound to guarantee consistency – transactions wait out time-uncertainty bound before committing a transaction. Spanner is the first to use ε in production transactional systems. While it is globally distributed, its idea of using ε is adopted in many intra-datacenter systems such as FaRMv2 [32]. However, inside datacenters, with recent software and hardware improvements such as RDMA, NVMe, and in-memory storage, transaction latencies are going towards microsecond level. For example, FaRMv2 is built atop RDMA for datacenters and has ε of ∼20μs which already accounts for 25% of median transaction latency! Tight ε improves the performance of these systems both in terms of latency and throughput.

Consistent snapshots: Consistent snapshots [11, 16] is another important application for datacenters for debugging, failure handling, and recovery for cloud VMs. The consistency across servers can be guaranteed by waiting out ε to ensure the scheduled snapshot time is passed. With recent software and hardware improvements, ε becomes a performance bottleneck at a similar level as in distributed databases, limiting the frequency of taking snapshots.

Network telemetry: As network latency reduces to the order of a few microseconds, millisecond-level ε is too coarse-grained. Tight ε enables a wide range of fine-grained network telemetry. For example, per-link latency or packet losses can be measured by comparing the timestamps or counters at both ends of a link read at the same time [23, 24, 40]. Synchronized network snapshots at RTT scale can be enabled with tight time-uncertainty bound, and can be used for various telemetry needs such as measuring traffic imbalance across different links/paths in the dataceter [37].To achieve these, switch clocks also need to be synchronized.

One-way delay (OWD): Synchronized clocks enable the measurement of one-way delays. Small ε provides a tighter bound on the error in the measurement especially under failures. Measurement of OWD is useful for many applications including telemetry and congestion control. For example, OWD differentiates between forward and reverse-path congestion improving performance of delay-based congestion control algorithms such as Swift [17] (§6.3).

Distributed logging: A key challenge for debugging large-scale distributed systems is to analyze logs collected from different devices with clock differences. Tighter ε enables more useful analysis and opens up more distributed debug-

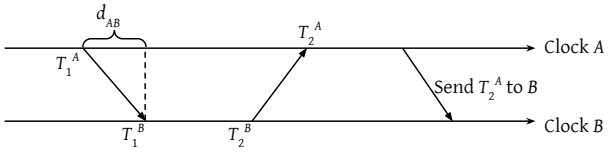


Figure 1: Message exchanges to synchronize B to A.

ging opportunities. Our ∼100ns ε is about the same as L3 cache miss time, so it can help order all log messages in a datacenter. We note that this class of applications has an additional requirement in that the synchronized clocks follow a master clock that reflects the physical time of day (§4.5).

3 Failures in Clock Synchronization System

In this section, we discuss the different failure scenarios affecting a clock synchronization system and their respective impacts. We start with a brief background on clock synchronization to aid the discussion.

3.1 Background on Clock Synchronization

The clock is driven by a crystal oscillator. Every device has a clock, whose value is incremented on every tick of a hardware oscillator. Different oscillators, even of the same type, have slightly different frequencies. The frequency of an oscillator may change over time, due to factors such as temperature changes, voltage changes, or aging resulting in clocks to drift away over time. As an example, oscillators in production networks can have a frequency variation of ±100 ppm (parts per million) [7], meaning that the oscillator can drift within the range of ±100μs per second compared to running at the nominal frequency. More stable oscillators (e.g., atomic clocks based on Cesium, Hydrogen or Rubidium particles or oven-controlled oscillators) are too expensive to deploy on every device in production.

Clocks exchange messages with each other for synchronization. To ensure that clocks remain close to each other, we need to periodically adjust the clocks to account for potential drift. Figure 1 shows an example where clock B synchronizes to A. A sends a synchronization message (abbreviated as sync-message in this paper) with a timestamp T_1^A based on A's clock, and B records the receiving time (timestamped by B) of the sync-message T_1^B. Now, if B knows the message delay d_{AB} from A to B, B can compute the *offset* between A and B as $T_1^A + d_{AB} - T_1^B$. To know d_{AB}, B sends another message to A to measure RTT, and use half of RTT to estimate: $d_{AB} = (T_2^A - T_1^A - (T_2^B - T_1^B))/2$. B uses *offset* to adjust its clock. A periodically sends out these sync-messages at an interval denoted by sync-interval. The accuracy of d_{AB} depends on multiple factors and we discuss them below.

A network of clocks synchronize using a synchronization structure. A common way to do this is to construct a spanning tree over which sync-messages are sent, e.g., PTP which is the most widely available system for datacenter clock synchronization uses a spanning tree with one device serving as the root (called master or grandmaster). The model for best case synchronization is that each device's parent is one of its direct neighbors in the physical network and sync-messages flow periodically from the root across the spanning tree.[1] This has two advantages. First, it allows switch clocks to also be synchronized enabling additional telemetry applications (§2). Second, it significantly improves the measurement of d_{AB} as shown in Figure 2. Noises in estimation of d_{AB} by halving the RTT can arise due to (1) asymmetric propagation delays of the forward path and the reverse path, and (2) queuing delays. For direct neighbors in the physical network, propagation delay asymmetry is near zero, and there is no queuing delay[2]. There are proposals that do not use a spanning tree as the synchronization structure but either they don't reflect the physical time [21] (§4.5) or they cannot provide submicrosecond-level precision [12, 27, 28] (§7).

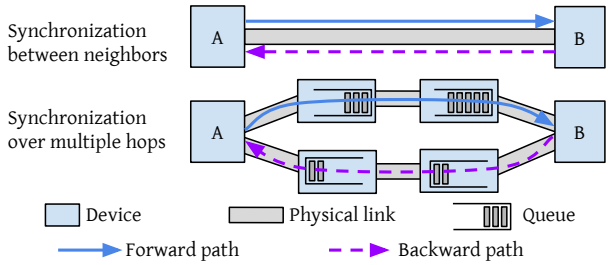


Figure 2: Benefit of synchronization between neighbors: symmetric forward and backward paths, and no noises from queuing delay.

Time-uncertainty bound. As clocks can drift apart over time, time-uncertainty bound (ε) can be calculated as:

$$\varepsilon = \varepsilon_{base} + (now - T_{last\_sync}) \times max\_drift\_rate \quad (1)$$

ε of a clock exhibits a sawtooth function. T_{last_sync} is the last time when the clock is synchronized to the *root* (not just its direct parent), $now - T_{last\_sync}$ increases with time and goes back to zero after synchronization to the root, and max_drift_rate is a constant representing the maximum possible drift rate between the local clock and the root's clock. The ε_{base} is a small constant (a few nanoseconds) that accounts for other noises (e.g., timestamping errors, bidirectional delay asymmetry of physical links, etc.).

We will show that in the face of failures in production environments, max_drift_rate should be conservatively derived (§3.2.1), and $now - T_{last\_sync}$ can be large (§3.2.2).

3.2 Impact of Failures on ε

We classify failures affecting clock synchronization into three categories and study their impact on ε – failures that induce large frequency variations and need a conservative setting of max_drift_rate, connectivity failures that affect T_{last_sync}, and incorrect behaviors due to broken clocks and message corruption that need to be detected and addressed.

3.2.1 Failures that Induce Large Frequency Variations

An oscillator's frequency can incur a large variation in the event of sudden temperature or voltage fluctuation. Cooling failures are common and can affect thousands of machines.

[1] Note that PTP doesn't require this to be the case.

[2] While the devices may have local queues, the timestamp is marked at dequeue/egress time and is not subject to local queuing delay.

In an cooling incident that occurred in production recently, it resulted in errors related to clock synchronization in a large fraction of machines (and not just the ones affected by the failure). The temperature variation resulted in oscillator frequency variation to exceed *max_drift_rate* and the operator had to shut down many machines.[3] Thus, the *max_drift_rate* needs to be set very conservatively (e.g., 200ppm in TrueTime [12]) to tolerate frequency variations under a wide range of temperature (e.g., up to 85 °C) even though in normal cases, temperature variations occur slowly [13]. This entails that in order to keep ε small, we need to reduce $now - T_{last\_sync}$ through frequent messaging – ε of 100ns with *max_drift_rate* of 200ppm needs sync-interval to be <500μs. Software cost of reducing sync-interval to such low values is high – PTP takes one core to process thousands of sync-messages and associated computations per second [1], and Huygens consumes 0.44% CPU for a sync-interval of 2s (which grows proportionally as the interval is reduced). We need hardware support for efficiency (§4.1).

3.2.2 Connectivity Failures

Failures that break the connectivity of the spanning tree also affect ε. For example, if a device or a link in the spanning tree fails, the whole subtree under this device or link loses connectivity to the root[4], until a new spanning tree is reconfigured by the SDN controller. ε grows proportionally to the time it takes for recovery – if it takes 100 ms, ε grows to more than 20μs. Even a distributed spanning tree protocol supported by PTP (best master clock algorithm) is slow to converge.

What is worse, is that the inflation of ε is not only for a device affected by the failure at a given time; instead, **almost all devices have to report high ε, all the time** and not only during the failure duration. This is because a device cannot distinguish whether it is affected by a failure or not. Consider a 3-node setup as depicted in Figure 3 with *A* as the root of the spanning tree and *B* and *C* as *A*'s child and grandchild respectively. When *A* fails, *B* detects the failure but *C* continues synchronizing to *B* without noticing the failure. This means at any time, there is no way for *C* to tell if it is in-sync or not, no matter if there is an actual failure or not and thus, it has to **always** report large ε (i.e., > 20μs) even during normal periods.[5] Another way to look at this is in the context of Equation 1, *C* cannot set T_{last_sync} to the time it receives the last sync-message from its parent T_{last_msg}; instead, for correctness, *C* has to **always** set $T_{last\_sync} = T_{last\_msg} - T_{recovery}$, where $T_{recovery}$ is the maximum time to recover from any failure that may break its connectivity to the root. All non-direct descendants of the root are affected by this.

[3] Normally, after a cooling system failure, operators let machines continue running for 10s of minutes before the recovery of cooling system or a gradual shutdown of machines, because this is usually safe and a sudden total shutdown should be avoided as much as possible.

[4] PTP is configured on a per-port basis (not per-device), so sync-message cannot bypass the failed link or the link associated with the failed device.

[5] Without changing the PTP standard, *B* cannot explicitly communicate to *C* about the failure.

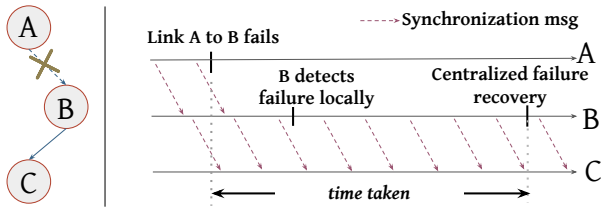


Figure 3: Challenge of determining T_{last_sync}. Node C cannot determine if it is synchronized to the root or not, so C has to always set T_{last_sync} conservatively early to account for possible down time.

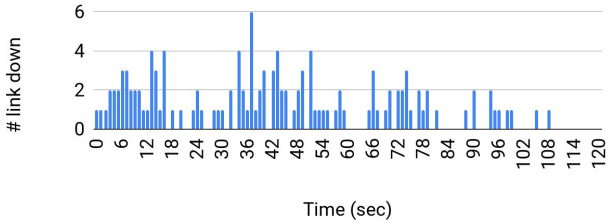


Figure 4: Number of link down events per second in a 1000-machine cluster during a near two-minute window of a failure incident.

There are many possible causes of connectivity failures: besides the common link or switch down, there are incidents that can take down massive (10s to 100s) devices or links, such as failures related to patch panels, link bundles, power domains, or human operations [38, 39]. Figure 4 shows the time series of link down events in a 1000-machine cluster during a failure incident. The suspected cause was a software bug related to a patch panel but its impact on device/link failures lasted across nearly two minutes – a total of 133 links go down. Thus, in order to provide small ε, the system must recover from connectivity failures quickly.

3.2.3 Broken Clocks and Message Corruption

Clocks may break and stop functioning well resulting in actual drift rate to exceed *max_drift_rate*. While this is rare relative to more severe hardware problems – statistics from production show that broken CPUs are 6 times more likely than broken clocks [12] – they need to be taken care of to provide correct ε to applications. Similarly, sync-message corruption may garble the associated timestamp and affect correctness of reported ε. A fault-tolerant clock synchronization system must detect and address such anomalies.

4 Sundial Design and Implementation

Motivated by the discussion above, we identify two key requirements to build a fault-tolerant clock synchronization system for datacenters that achieves performant time-uncertainty bounds. First is a small sync-interval (§3.2.1) – this is well served with a hardware implementation to avoid high CPU overhead of receiving and transmitting synchronization messages in software. Second is fast failure recovery so that ε continues to be small even when failures happen (§3.2.2). The challenge here is that recovering solely via a centralized controller is slow for our target ε requirements. Instead, as we show later, we can recover from most failures locally by adding redundancy to the synchronization graph, where in

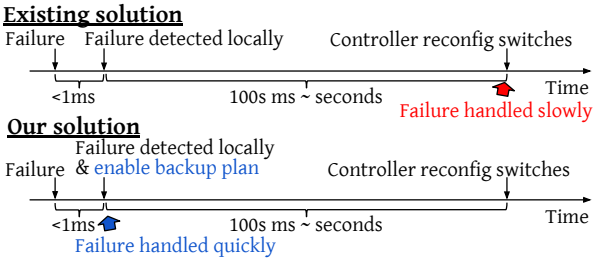


Figure 5: Fast failure recovery using precomputed backup plan.

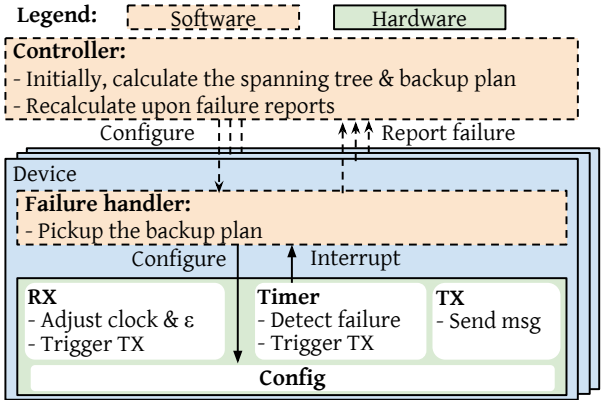


Figure 6: Sundial Framework. Solid arrows are the fast local recovery. Dashed arrows are slower but non-critical paths of recovery.

addition to the primary spanning tree, each device maintains a backup parent, such that it can transition to the backup parent locally upon detecting a failure. As shown in Figure 5, this takes the round trip time to the controller and the computation time out of the critical path of failure handling.

Thus, Sundial uses a hardware-software codesign. Figure 6 depicts Sundial's framework, which has three main components. Sundial implement the most essential functions of exchanging synchronization messages and detecting failures in hardware such that it can synchronize frequently and quickly detect failures. Sundial relies on software components to take action once a failure is detected, by invoking a failure handler in software which reconfigures the hardware to transition to the backup parent pre-programmed by a centralized controller (also in software). We use the topology in Figure 7(a) as a toy example to aid with the discussion in this section with Figure 7(b) as an example spanning tree.

4.1 Sundial Hardware Design

Sundial's hardware has three main components. It implements *frequent* transmission of sync-messages in a *synchronous* fashion, i.e., sync-messages are sent downstream only upon their receipt. The hardware is also responsible for detecting failures and triggering software handlers for quick recovery. Finally, the hardware maintains the current value of ε. We detail out these components below.

4.1.1 Frequent Synchronous Messaging

Sundial's hardware supports frequent message sending to prevent clocks from drifting apart significantly. On the root, this is done via a hardware timer maintaining a counter that

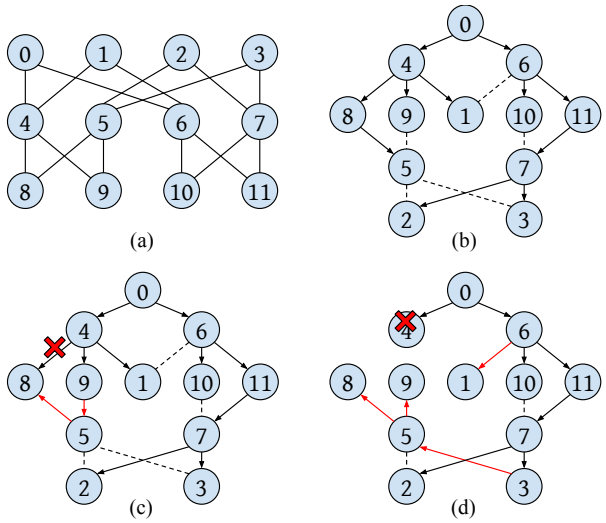


Figure 7: Failure cases in a k=4 FatTree. (a) is the raw FatTree. To show the spanning tree clearer, we draw an equivalent topology in (b) and a spanning tree in it. An arrow is from a parent to its child, and a dashed line indicates an edge not used in the spanning tree. (c) shows one way of adjusting the spanning tree when the link between 4 and 8 fails; not only the directly impacted nodes (node 8), but also other nodes (node 5) have to change parent. (d) shows one way of adjustment when node 4 fails; the way node 5 changes its parent (to node 3) is different from the case in (c) (change to node 9).

increments on every oscillator cycle, and triggers message transmission when the time since last transmission exceeds the configured sync-interval. We configure sync-interval on the root device to be around 100μs. The sync-messages are sent at the highest priority, but the network overhead remains extremely small – a 100-byte packet every 100μs only consumes less than 0.01% bandwidth and adds at most 10ns queuing delay for other traffic.

For non-root devices, a challenge is that an upstream failure can affect all devices in that subtree. Consider the case in Figure 7(c), if link 4-to-8 goes down, 8 needs to switch to 5 as its parent, which needs 5 to change its parent as well. A potential solution is explicit notification of failures to other devices, but this has two issues – not only can this be unreliable (since the notification messages may get dropped), it also adds complexity to the hardware. Instead, we solve this via *synchronous messaging* where message transmission is triggered only upon receipt of a message from upstream. In this way, an upstream failure implies that messages stop propagating downstream, and devices can take corrective actions.

4.1.2 Fast Failure Detection

Sundial's hardware uses a timeout to detect if it stops receiving messages indicating an upstream failure. The timeout is set to span multiple sync-intervals, such that occasional message drop or corruption doesn't trigger it. It's implemented using a counter that is incremented on every oscillator cycle, and cleared on receiving a sync-message – once it's exceeded, the hardware issues an interrupt to the software.

To detect broken clocks and message corruption, each de-

vice verifies the incoming timestamp (adjusted for link delay). If the adjusted value lies outside the local ε, the message is marked invalid and doesn't trigger an update and message transmissions. A broken clock can cause continuous invalid messages and thus, we don't reset the timeout counter on their receipt. Once a broken clock is detected, the failure handler in device software is triggered to handle it (§4.2.2).

4.1.3 Time-uncertainty Bound Calculation

The hardware maintains ε according to Equation 1. In our implementation, we configure *max_drift_rate* = 200ppm and ε_{base} = 5ns×*depth* where *depth* is the distance of the device from the root in the tree.

T_{last_sync} is updated when receiving a sync-message. In PTP, T_{last_sync} should be set to earlier than T_{last_msg}. Thanks to synchronous messaging, Sundial sets it to T_{last_msg} since a device stops receiving messages on an upstream failure. This lowers $now - T_{last\_sync}$ which in turn lowers ε.

4.2 Sundial Software Design

There are two main components to round out the fault-tolerant design of Sundial – a centralized SDN controller that precalculates backup plans and programs them on the devices and a failure handler in device software that quickly moves to the backup when a failure is detected by the hardware.

4.2.1 Centralized Controller

The centralized controller in Sundial is responsible for computing the primary spanning tree along with the backup plan based on the current topology and configures the devices accordingly. Comparing Figure 7(c) and 7(d), we see that not all neighbors of a node (e.g., node 5 in the figure) can be the backup parent under different failures. Sundial uses a search algorithm (detailed below) to compute a fault-tolerant backup plan that is *generic* to link, non-root node, root node, and domain failures (which can take down multiple links or devices). We break down this requirement into 5 properties.

Properties of a fault-tolerant backup plan. We briefly introduce the terminology used. The **primary spanning tree** is one that is currently being used to propagate sync-messages. The **backup plan** consists of a backup-parent for each node/device and a backup root. Terms like parent, edges, paths, and ancestors apply separately to the primary and the backup graph (graph formed by the edges in the backup plan).

(1) No-loop condition: *For any primary subtree, the backup edges of nodes in the subtree do not form a loop.* This is a necessary and sufficient condition to be generic to any single link failure. The necessity is obvious: if there is a loop, the nodes in the loop do not synchronize to the root after a failure. We prove the sufficiency by induction as follows. Suppose a k-node subtree is affected by a link failure, and the k backup edges do not form a loop (Figure 8); the nodes other than the k nodes are unaffected and still form a tree (called the *main tree*). At least one of the k nodes' (say, C) parent is in the main tree; otherwise, all k nodes' parents are in the k nodes, which must form a loop, contradicting the no-loop condition.

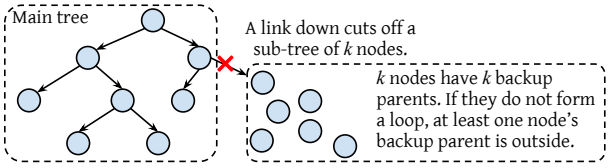


Figure 8: No-loop condition. It is sufficient to guarantee connectivity after any link failure.

We can now expand the main tree to include C since C is connected to the main tree via its backup edge. We can then iteratively add the remaining $k-1$ nodes to the main tree.

(2) No-ancestor condition: *The backup parent of a node is not its ancestor.* This and property (1) together ensure that the backup plan is generic to any non-root node failure. Otherwise, if that ancestor fails, that node has no backup parent.

(3) Reachability condition: *The backup root must be able to reach all nodes through backup paths.* This is necessary and sufficient to be generic to the root failure. When the root fails, all nodes change to their backup parents, and the backup root will become the new root. To synchronize all nodes, they must be reachable from the backup root.

(4) Disjoint-failure-domain condition: Domain failures present a unique challenge, because they may take down multiple devices or links. If a domain failure breaks the connectivity of a device s to the root, s will turn to its backup parent; but if the domain failure also takes down its backup parent, then s cannot recover its connectivity.

The following property solves this problem: *for any node s, there shouldn't be a single domain failure that both breaks s's connectivity to the root and takes down the backup parent or backup edge, unless that failure also takes down the node s.*

Formally, if the set of failure domains that can break s's connectivity to the root[6] is D_p, the set of failure domains that can take down s's backup parent or backup edge is D_b, and the set of failure domains that s belongs to is D_s, we should have $D_p \cap D_b \subseteq D_s$.

The necessity is obvious. We present the intuition behind the proof of the sufficiency. If a domain failure happens, s has two possibilities: either s's connectivity is unaffected, or s connects to its backup parent b. If it is the latter, then the questions is whether b is connected to the root, which also has two possibilities. Doing this recursively, s keeps connecting to more nodes along a backup path. The backup path will not go indefinitely due to the no-loop condition, so it finally reaches either an unaffected node or the root.

(5) Root failure detection: Upon root failure, the backup root needs to collect sufficient information to elect itself. Figure 9 describes the approach – the backup root is chosen amongst root's children so it has one source of information by itself.

To get information from additional sources, we set up the backup graph to have a backup path from the subtree of another child of the primary root (i.e., the backup path from node

[6] Any device or link failure along the primary path from the root to s can break s's connectivity.

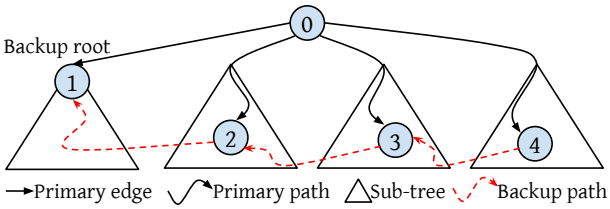


Figure 9: Root failure detection. Under any non-root failure, the backup root continues receiving messages, which can be used to distinguish other failures.

2 to 1 in Figure 9). In this way, if the link between the primary root and the backup root fails (link from 0 to 1), the backup root knows the primary root is still alive because it continues receiving sync-messages that come through the backup path. We can continue this backup path to cross more subtrees of children of the primary root to get additional sources of information (e.g., crossing node 3 and 4 in Figure 9).

In this way, as long as the root is alive, the backup root continues receiving sync-messages. Only when the root fails, the backup root stops receiving messages. So the backup root can detect the primary root failure using a second timeout of not receiving messages after it first turns to its backup parent, and it elects itself as the new root after the second timeout.

Putting all 5 properties together. Only non-root nodes have backup parents, so there are N-1 nodes and N-1 edges in the backup graph (N is the total number of nodes), so there must be exactly one loop[7] in the backup graph, and each node in the loop has a backup subtree (can be a single node) under it (Figure 10). With property (3), the backup root must be in the loop, so that the backup root can reach all nodes. The loop should cross multiple primary subtrees of root's children, so it meets both property (1) and property (5) (it delivers multiple sources of primary root's information to the backup root). Lastly, the backup graph should meet properties (2) and (4).

Figure 11(a) shows an example of primary tree and backup graph for the topology in Figure 7. Note that the computed primary tree is different to support a backup graph. The backup graph has a loop (between node 4 and 8) with the backup root 4 on it; the loop crosses the two primary subtrees of root's children (node 8 is under node 6's primary subtree). To show how property (4) handles domain failures, we add a failure domain that includes both node 11 and 3 (primary and backup parents of node 7 in Figure 11(a)). Now in the new backup graph (Figure 11(b)), to meet property (4), node 7's backup parent becomes node 2, so that even if both node 3 and 11 go down, node 7 (and other nodes) is still connected.

We want to highlight how the system recovers when the root fails. All backup edges get enabled forming a loop, but no sync-messages flow at this time. At the second timeout, the backup root elects itself and ignores incoming messages, effectively disabling the edge towards it (Figure 10). In this way, sync-messages do not loop.

[7] A graph with equal numbers of nodes and edges has at least one loop. In addition, if there is more than one loop, then the graph is not fully connected.

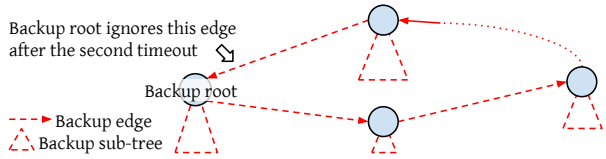


Figure 10: Backup Graph. There is exactly one loop with the backup root in it. Each node in the loop is the root of a subtree.

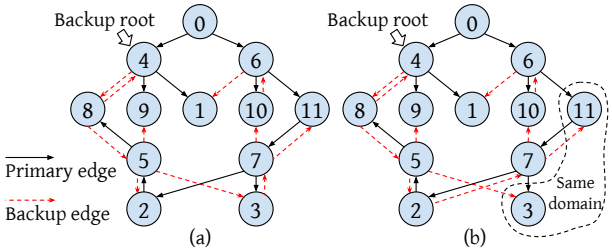


Figure 11: (a) A primary tree and a backup graph that meet all properties in Figure 7. But if node 3 and 11 are in the same domain, node 7 cannot have them as its primary and backup parents, so its backup parent becomes node 2 in (b).

Algorithm for computing backup plan. Sundial uses a search algorithm to calculate the backup plan which includes a primary tree and the backup graph. Note that not every primary tree has a valid backup graph. Thus, the goal is to search for a primary tree and its backup graph together. The search heuristic is based on the *score* of a primary tree – the maximum number of edges in the backup graphs it supports. The corresponding backup graphs are called the largest backup graphs (of the primary tree).

Algorithm 1 describes the algorithm. *pending* is the set of primary trees that are pending to be checked, initialized with a simple BFS (Line 1). After initialization (Line 2), we start the SEARCH function (Line 3) that will return a pair of primary tree and backup graph. In SEARCH, each time, we pick the primary tree p with the highest score (Line 6) – the most promising one – and find the largest backup graphs for it (Line 7). If some backup graph is complete, i.e., every device (including the backup root) has a backup parent, the search successfully returns (Line 8 - 9). Otherwise, we update the best score so far (Line 10), and mutate p (Line 11) to get a new set of primary trees in *pending* and iterate.

In MUTATE, for each backup graph b (Line 14), we try to expand b to include edge <x, y> (Line 15). Since <x, y> is not usable in backup graphs of p[8] (i.e., USABLEINBACKUP(<x, y>, p) is false), we IMPROVE p to make <x, y> usable (Line 16). We then add each improved version p' to *pending* if not already tested (Line 19). After all the mutations, p is removed from *pending* (Line 22). We will discuss the optimizations in Line 20 - 21 later.

FINDLARGESTBACKUP and IMPROVE are the key functions. FINDLARGESTBACKUP conforms to the 5 properties. Properties (2) and (4) decide what edges can be used in backup graphs given a primary tree p, as expressed in function

[8] <x, y> is not usable for sure; otherwise b is not the largest because it can readily include <x, y>.

Algorithm 1 Searching for a primary tree and a backup graph.

```
1:  pending = {BFS(prim_root)};
2:  tested = ∅; best_score = 0;
3:  return SEARCH();
4:  function SEARCH
5:      while pending is not empty do
6:          p = pending.get_best(); tested∪={p};
7:          backup_set = FINDLARGESTBACKUP(p);
8:          if ∃b ∈ backup_set|b is complete then
9:              return p, b;
10:         best_score=max{best_score,calc_score(p)};
11:         MUTATE(p, backup_set);
12:     return NotFound;
13: procedure MUTATE(p, backup_set)
14:     for b in backup_set do
15:         for each <x, y> | x ∈ b, y ∉ b do
16:             new_prim_set=IMPROVE(p, <x, y>, b);
17:             for p′ in new_prim_set do
18:                 if p′ ∉ tested then
19:                     pending∪={p′};
20:                     if calc_score(p′)>best_score then
21:                         return ;
22:     pending-=p;
```

Algorithm 2 Check if <x, y> is usable in backup graphs of p.

```
1: function USABLEINBACKUP(<x, y>, p)
2:     return (x is not y's ancestor in p) && (y's ancestor in p and
   x meet disjoint-failure-domain condition);
```

USABLEINBACKUP (Algorithm 2). Properties (1), (3), and (5) decide how the backup graph should look like. We can simply use BFS starting from the backup root (property (3)) to find the tree (property (1)) that is largest, and then enumerate the backup parent for the backup root and see if it forms a loop that meets property (5). IMPROVE's goal is to change p to p' so that <x, y> becomes usable (i.e., USABLEINBACKUP(<x, y>, p') is true). It finds the set of p' that meets this goal.

As long as FINDLARGESTBACKUP and IMPROVE are exhaustive, the search is exhaustive – it will find a solution if one exists. The search process is similar to an algorithm that finds two edge-disjoint spanning trees [35], because our backup graph is composed of a more restricted spanning tree that is edge-disjoint with the primary tree, and an extra edge towards the backup root. The problem seems to be NP-hard although we don't have a proof yet.

In practice, our implementation of SEARCH is extremely fast – it only takes 148ms on average in a simulated Jupiter topology with 88,064 nodes [33] leveraging the following three strategies. (i) In Line 20 - 21 of Algorithm 1, we prune enumerations as per Line 14 - 15 as long as we find a p' that is heuristically better than any primary trees so far (including p). This significantly speeds up the search, as we can immediately make progress – after return (Line 21), the search immediately starts a new iteration at Line 6 based on p', which is heuristically better than continuing mutating p. Note this strategy does not miss any primary trees, as the original p remains in *pending*. (ii) FINDLARGESTBACKUP only returns one of the largest backup graphs, rather than all of them. This is sufficient as all largest backup graphs for a primary tree connect the same set of nodes and we use this strategy in Algorithm 3. (iii) IMPROVE just returns the set of p' that *keeps the largest number of b's edges usable*. This tries to keep as many useful fruits of past iterations as possible, so it speeds up the search. Algorithm 4 is based on this strategy.

Algorithm 3 Finding the largest backup graph of p.

```
1: function FINDLARGESTBACKUP(p)
2:     b=BFS_ForBackup(backup_root, p);          ▷ BFS uses
   USABLEINBACKUP to avoid unusable edges.
3:     Find <y, backup_root> where y ∈ b && USABLEIN-
   BACKUP(<y, backup_root>, p) && the loop crosses multiple
   subtrees of prim_root in p; Add <y, backup_root> to b;
4:     return {b};
```

Algorithm 4 Changing p to make <x, y> usable and keep as many b's edges usable as possible.

```
1:  function IMPROVE(p, <x, y>, b)
2:      if x is y's ancestor in p then
3:          for each edge <u, v> on the path x ⇝ y in p do
4:              new_prim_set∪ = RECONNECT(v, x, p, b);
5:      if <x, y> fails disjoint-failure-domain condition then
6:          new_prim_set∪ = RECONNECT(y, x, p, b);
7:      return new_prim_set;
8:  function RECONNECT(v, x, p, b)
9:      BFS from v along reverse edges, and stops at nodes outside
   x-subtree in p, while keeping as many b's edges usable as pos-
   sible. It gives a set of paths S={w ⇝ v|w is outside x-subtree in
   p}
10:     for path in S do
11:         For each <i, j> ∈ path set j's parent to i in p to get p′;
12:         new_prim_set∪ = {p′};
13:     return new_prim_set;
```

These three strategies significantly reduce the computation time per iteration (Line 6 - 11). While the latter two strategies make the search non-exhaustive, all practical datacenter topologies have high redundancy such that in our experiments, we quickly found a backup-plan even after injecting 50 successive failures. Also owing to the high redundancy in practical topologies, the number of iterations is small since the initial p already has a very high score, only a few hundreds below the total number of nodes. Finally, another consequence of high redundancy is that in practice, the search iterates with almost monotonically increasing scores [9], sometimes with jumps of tens or hundreds, reaching the final backup plan in tens of iterations on average.

Mutation for meeting property (5) follows a similar process as MUTATE. We omit the details due to limited space.

Calculating $\varepsilon_{base,backup}$. When a node turns to its backup parent, its *depth* may change, so we also precompute $\varepsilon_{base,backup}$

[9] IMPROVE can easily find paths while keeping all b's edges usable, because of the high redundancy.

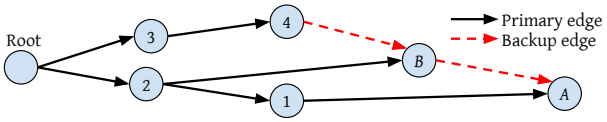


Figure 12: Node A's *depth* is dependent on the failure. If node 1 fails, A's *depth* is 3 (A, B, 2, root). But if node 2 fails, A's *depth* is 4 (A, B, 4, 3, root).

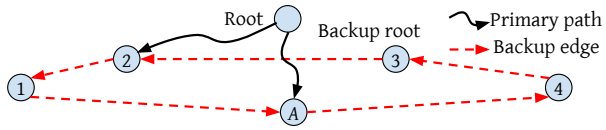


Figure 13: A has 6 possible paths to the root, of 3 types. **(1) Backup path**: if the root is down, the backup path (A, 1, 2, 3) is in effect. **(2) Primary path**: when A is unaffected by failures. **(3) Mixed path**: when failures affect A and some other nodes on the loop, A connects to the root first along the loop for one or more hops, and then along the primary path (e.g., A, 1, 2, ..., root). There are 4 possible mixed paths, starting the primary paths from respective node 1, 2, 3, and 4.

to which a device set ε_{base} upon timeout. The exact *depth* is failure-dependent as shown in Figure 12.

So we calculate the maximum possible *depth* for each node after *any* failures. A naive approach is to enumerate all possible combinations of failures, which can be slow. Instead, Sundial uses a simple dynamic-programming (DP) based scheme. If a node s turns to its backup parent b, we calculate s's maximum possible depth $s.depth_{backup}$:

$$s.depth_{backup} = 1 + \max(b.depth_{primary}, b.depth_{backup})$$

where the max function considers two possible cases: b is unaffected by failures ($b.depth_{primary}$ denotes b's depth in the primary tree, a deterministic value), or affected by failures. $depth_{backup}$ can be calculated top-down.

DP works for all nodes except the nodes on the loop in the backup graph, whose DP calculations inter-depend. But we can easily calculate their maximum possible depths. On an L-node loop, for each node we enumerate all $L+1$ possible ways it connects to the root (Figure 13). So the overall time complexity is $O((N-L)+L(L+1))$ for a total of N nodes.

4.2.2 Failure Handler in the Device Software

A daemon running in firmware serves as the failure handler and responds to interrupts generated by the hardware once it detects a failure – the hardware is reconfigured to move to the backup parent based on the backup plan and set ε_{base} to $\varepsilon_{base,backup}$. For the backup root, if an interrupt is triggered, the failure handler also continues to monitor incoming sync-messages for the second timeout. At the second timeout, the device sets itself as the primary root.

Handling broken clocks. If a clock is broken [12], it can drift away faster than *max_drift_rate*. In Sundial, we detect such clocks in two steps: (1) detect the existence of a broken clock when receiving an invalid message, and (2) confirm which one is broken. Figure 14 illustrates the process. As such, a broken clock is isolated without affecting other clocks.

The failure handler is triggered by a hardware interrupt upon receiving an invalid message to handle broken clocks.

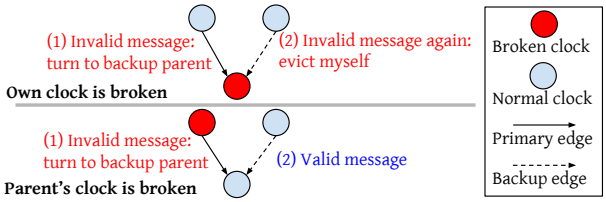


Figure 14: Handling a broken clock in two steps. If a node's own clock is broken, the messages from both its primary and backup parents are marked invalid by itself (the timestamp is outside local ε), so it evicts itself. If a node's parent's clock is broken, after receiving an invalid message it turns to its backup parent, and continues synchronization thereafter.

For the node with a broken clock, it evicts itself (no longer participates in synchronization). For the node whose parent has a broken clock, it turns to its backup parent.

4.3 Implementation

Controller. We implement a module in the network controller. The module registers a function to be called by the controller framework for failure notifications. When notified, this module reads the current device/link/port states, and computes a new backup plan. For each device, it compares the existing configuration and the new configuration, and only reconfigures the devices whose configuration changes, through RPC. It also configures the TX side of both primary and backup edges to send sync-messages.

RPC Interface between the Controller and Device Firmware. The controller and the device firmware communicates through RPCs. These RPCs have the following parameters: backup parent, first timeout, and second timeout which are used to configure the device hardware.

Firmware. The RPC handler configures the backup parent, the first timeout, and the second timeout accordingly. The backup parent and the second timeout are maintained in the firmware, and the first timeout is maintained in the hardware registers to enable failure detection in hardware. Only the backup-root has a non-zero value for the second timeout.

The firmware also registers a handler function for the interrupt triggered by the first-timeout. This function first reconfigures the hardware to accept sync-messages from the backup parent; then, if the second-timeout is non-zero, it waits for the timeout to see if it receives any new sync-messages; if not, it configures the hardware to become the root.

We cannot reveal hardware details due to confidentiality.

4.4 Practical Considerations

Concurrent connectivity failures may happen in practice, and may not be recovered by the backup plan, which needs to involve the controller. Sundial maintains the correctness of ε in this case. The only impact is that ε grows larger before being recovered by the controller: if it takes 100ms to recover, ε grows up to 20μs during this time (still $\sim$100ns during normal time). The impact is negligible, because compared to single failures, concurrent failures are already rare, and only a very small subset of them cannot be recovered by the backup

plan, as discussed below.

The most commonly seen concurrent failures are caused by a domain failure, which is not an issue because of the disjoint-failure-domain condition of the backup plan (§4.2.1).

If cross-domain failures happen, whether they impact Sundial depends on their locations and time proximity. For the nodes whose connectivity is affected, the backup plan is ineffective only if these failures also take down their backup parents/edges (special locations) within a short period of time before the controller recomputes a new backup plan (time proximity). The chance is very small, because cross-domain failures are random in locations and time proximity.

Small window of error before evicting a broken clock. The broken clock detection only happens when messages arrive. There is a small time window between when the failure actually happens and when the next message arrives, during which errors could arise. This can be solved via hardware redundancy – each node physically keeps two clocks, and each clock query reads the two clocks and checks if they match (their time-uncertainty ranges overlap). Once a clock is broken, the next read immediately detects it. Additionally, Sundial prevents this failure to affect other clocks, because its children ignore the invalid messages.

False positives. If a device timeouts without a failure, it will turn to the backup parent. Such false positives are harmless, except extra controller processing. We do not observe false positives in our experiments.

4.5 Sundial's Position in the Design Space

4.5.1 Design Space of Clock Synchronization

At the submicrosecond level, Sundial is the first to support time-uncertainty bound. We identify three key aspects of the design that a clock synchronization system must answer.

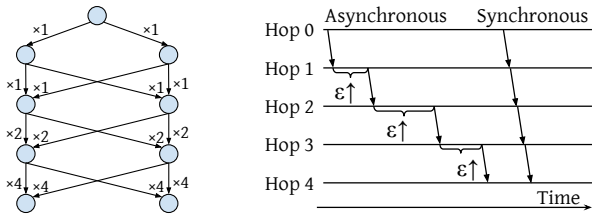
1. Type of message: There are multiple options, synchronization messages can either be sent directly with specialized physical layer (PHY) with zero-overhead messages, or at higher layers (L2, L3, L4) with increasing bandwidth overhead and increasing ease of deployment.

2. Noise due to message delay between a pair of clocks. The message delay in the forward and reverse directions may be unequal due to queuing or asymmetric paths. There are three options to deal with such noise: (1) Only synchronize between neighboring devices, such that there is no noise (§3.1). (2) Use multiple messages to filter out noise; (3) Tolerate the noise. Option (1) is the best if all devices (switches and hosts) can participate. Otherwise, option (2) and (3) face a tradeoff between noise and overhead.

3. Network-wide synchronization structure: three options.

(1) Master clock distributed through a tree. A master clock distributes its time to other clocks through a tree. The master clock can synchronize to the physical time (e.g., via GPS), so that all clocks reflect the physical time.

(2) Master clock distributed through a mesh. Similar to (1), but instead of a tree, each clock receives sync-messages from multiple other clocks, forming a mesh.



(a) Synchronous messaging: exponential inflation of sync-messages. **(b)** Asynchronous messaging has much larger ε.

Figure 15: Mesh structure: higher ε due to asynchronous messaging.

(3) No master clock (no physical time). Clocks synchronize independently with each other without regards to a master clock. For example, in DTP [21] each clock follows the fastest of its neighbors. In this option, all clocks converge to a function (e.g., max() in DTP) of all clocks, which has nothing to do with the physical time. This option is worse than (1) and (2) because access to physical time is important for many datacenter applications.

Tradeoff between (1) and (2). While (2) is clearly more fault-tolerant, it cannot get ε as low as (1). The reason is that mesh-based solutions cannot use synchronous messaging. As shown in Figure 15a, if a clock receives sync-messages from k other clocks, synchronous messaging inflates the number of messages by k per hop, causing exponential inflation. So mesh-based solutions have to use asynchronous messaging, which has much larger ε – as shown in Figure 15b, ε increases per hop from the master to the participant clocks. On the other hand, tree-based solutions can use synchronous messaging, achieving much lower ε. §6.2 evaluates this effect.

4.5.2 Sundial's Design Choices

Sundial's key contribution is in the third design choice, which exhibits fundamental tradeoff between small ε and fault-tolerance. Sundial aims to achieve the best of both worlds, by combining tree and mesh topologies: Sundial sends messages through a mesh, such that it still has available edges upon failures; but the effective synchronization only happens over a primary tree, enabling it to use synchronous messaging.

The first two design choices have clear best options, and they are mainly determined by hardware availability. In our implementation, Sundial synchronizes neighboring devices at the L2 level as the specialized PHY layer is not available. That said, Sundial can benefit from such a layer if it's available. Comparison with other schemes is in §7.

5 Application Access to Synchronized Clocks

In Sundial, the primary mechanism to access synchronized clocks is via hardware Rx/Tx timestamps. Additionally, for applications that want to access host clock directly, Sundial provides local host to NIC clock synchronization.

Access via hardware timestamps. NIC and switch hardware timestamps marked on the packets [29] are the primary access mechanism, for which it provides ∼100ns time-uncertainty bound. Applications such as distributed databases

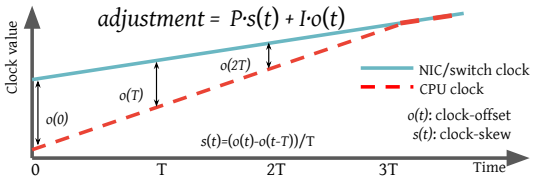


Figure 16: PI controller based on clock-skew; offset and skew are measured periodically and an adjustment is computed using suitable P and I constants.

that have strict ε requirements rely directly on NIC-Rx-timestamps marked on the last packet in a message to order them to provide consistency properties. Networking stacks such as Snap [26] provide op-stream interface to applications (preventing out-of-order delivery) and export the NIC timestamps. Telemetry and congestion control applications also rely directly on NIC timestamps to measure one-way delays.

Host clock synchronization. We also synchronize the local host clock to the NIC clock for applications that want to directly read the host clock (and don't require strict guarantees on ε). We use a Proportional-Integral controller based on clock-skew between the host and NIC clocks as depicted in Figure 16. We measure the offset, $o(t)$ and skew, $s(t)$, every T time-units (we use T=10ms) and apply the rate adjustment to the host clock to tick faster or slower. The constants P and I need to be tuned in production. One challenge is that the two clock-measurements are subject to local delays such as PCIe jitter and we use linear regression to filter the noise out.

6 Evaluation

Through experiments in a >500-machine testbed-prototype (§6.1) and through large-scale simulations (§6.2), we show that Sundial's time-uncertainty bound is ∼100ns under different types of failures, and discuss application improvements enabled by Sundial in §6.3.

6.1 Time-uncertainty Bound (ε) in Testbed

6.1.1 Methodology

Testbed. The testbed consists of 23 pods, 276 switches and 552 servers. A pod including 12 switches and 24 servers acts as a failure domain. The oscillators used in the hardware have a frequency specification of ±100ppm. The depth of the base spanning tree in the topology is 5.

Schemes for comparison. We compare Sundial with recent submicrosecond-level clock synchronization schemes: PTP [4], Huygens [13], and DTP [21]. While they do not consider time-uncertainty bound (ε) and how it is reported to applications, we augment the designs to provide ε, according to Equation 1 in §3.1 and describe them below.

***Sundial*:** We set the sync-interval to 90μs.[10] The timeout is 185μs (>2×sync-interval). The second timeout for the backup root to elect itself is set to 180μs (185+180>4×sync-interval). The backup plan has a maximum $depth_{backup}$ of 6.

***PTP+ε*:** PTP is the most common submicrosecond-level synchronization protocol with a default sync-interval of two seconds. To add ε, we set ε_{base} to 5ns×$depth$, and max_drift_rate to 200ppm. T_{last_sync} is updated as follows – for root's children, we set $T_{last\_sync} = T_{last\_msg}$; but for other descendants, we set $T_{last\_sync} = T_{last\_msg} - T_{recovery}$ to account for possible out-of-sync duration caused by remote connectivity failures that are oblivious to them (§3.2.2). We set $T_{recovery}$ to 2s, since it takes 2s to recover from failure.[11]

***PTP+DTP+ε*:** What if we could set lower sync-interval in PTP+ε? We evaluate another scheme that leverages DTP – DTP allows very small sync-interval (a few microseconds) with low bandwidth overhead by modifying the physical layer protocol. Since DTP requires hardware support, we emulate it in our testbed by setting 5μs sync-interval (much smaller than 90μs).[12] All devices that are not direct children of the root set $T_{recovery}$=100ms, where 100ms is the typical connectivity failure recovery time measured from datacenters.[13]

***Huygens+ε*:** Huygens gathers network-wide sync-messages during each 2-second sync-interval, and uses machine learning to decide the best adjustment for each device at the beginning of the next sync-interval. While we do not have its implementation, we report the best possible ε it can achieve. Specifically, we assume it is not affected by connectivity failures because of its use of network-wide information, so T_{last_sync} is set to the beginning time of each sync-interval (without minus $T_{recovery}$). We also assume it can filter out delay noises entirely and optimistically set ε_{base} to 0.

Failure injection. We evaluate the impact of failures on ε in Sundial and above schemes by injecting link failures, non-root device failures, root failures, and domain failures (where multiple devices can go down).

Metrics and measurement approach. We measure ε on every device by running a daemon in the firmware to read ε every 10μs. After a failure, the controller sends an RPC to configure the devices for recovery. The frequent monitoring interferes with processing RPCs that are sent by the controller in the event of failures. As a workaround, we set a stop time which allows the controller RPC to execute after the monitoring stops. In this way, the monitoring tells us which devices are affected by failures and their ε. But it also inflates the controller delay, which is unfair to other schemes as they heavily rely on the controller for failure recovery. With knowledge of the expected controller delay, we can easily restore the expected ε based on the measured ε (Figure 17), because ε's behavior is deterministic during failures recovery: ε keeps increasing, and goes back to normal when the failure is recov-

[10] 90μs is just enough for ∼100ns ε, although lower ε is achievable.

[11] In favor of low ε, $T_{recovery}$ = 2 seconds is already a very optimistic setting for PTP+ε, because recovery may take longer if the next sync-message is also dropped by another failure that just happens at that time. Setting $T_{recovery}$ larger results in even higher ε. But we show that even with this optimistic setting, PTP+ε still has much higher ε than Sundial.

[12] This is sufficient to show the improvement of ε, even though we don't have the physical layer protocol to keep the bandwidth overhead low.

[13] This is already friendly to PTP+DTP+ε because to *guarantee* correct ε, $T_{recovery}$ should be the maximum recovery time, which is several seconds.

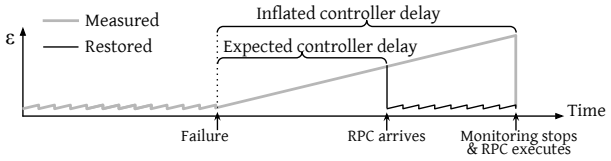


Figure 17: Restoration of ε under inflated controller delay.

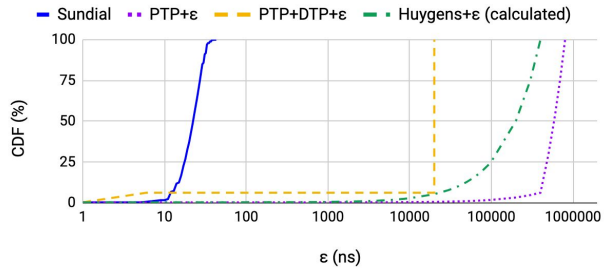


Figure 18: CDF of ε measured across devices without failures.

ered. To get the expected controller delay, we use its lower bound, the delay on the controller (without network delay), which is more friendly to schemes other than Sundial.

6.1.2 ε Distribution without Failures

Figure 18 shows the distribution of ε over all devices under different schemes. In Sundial, ε≤43ns, which matches the calculated value – the deepest device in the tree has ε_{base} of 25ns, and 90μs sync-interval leads to an additional 18ns.

In contrast, all other schemes have a much higher ε. In PTP+ε and PTP+DTP+ε, devices that do not directly synchronize to the root have to set T_{last_sync} earlier than T_{last_msg} by 2s and 100ms respectively, to account for possible failure-induced out-of-sync periods, so their ε can go up to 800μs and 20μs respectively during a sync-interval. Devices directly synchronizing to the root can set T_{last_sync} to T_{last_msg} and achieve lower ε. So their ε increases from 5ns (1-hop ε_{base}) to ∼400μs and 6ns respectively (2s and 5μs sync-intervals lead to 400μs and 1ns additional ε respectively at the end of each sync-interval). For these devices (∼6.3% of all), PTP+DTP+ε's low ε shows the benefit of extremely small sync-interval when failure is not a concern. Note that if available, Sundial can also benefit from DTP's physical layer design to futher reduce sync-interval. In Huygens+ε, during each 2s interval, ε increases from 0 to 400μs. Reducing sync-interval comes with CPU cost (Huygens already consumes 0.44% CPU of the whole cluster). However, even if the sync-interval was halved, ε is still 3 orders of magnitude higher than Sundial's.

6.1.3 ε Distribution during Failures

To understand the behavior under failures, we inject 50 random failures over a course of 6 minutes including 24 single link failures, 23 non-root single device failures, 2 domain failures and 1 root failure.

Figure 19 shows the time series of ε of a device affected by a link failure. In Sundial, ε is sawtooth between 15ns and 33ns during normal time, because this device has a depth of 3 in the tree.[14] When the link failure happens, ε increases

[14]Figure 21 shows the behavior at smaller timescales.

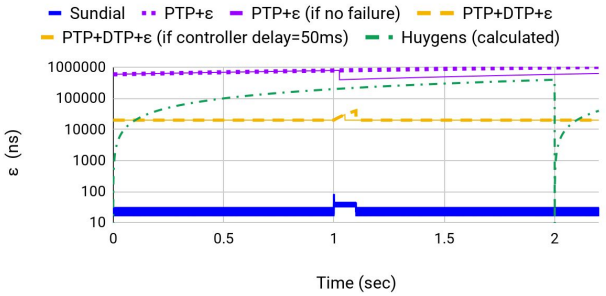


Figure 19: Time series of ε of a device affected by a link failure. The failure happens at 1s and the controller reacts to it near 1.1s.

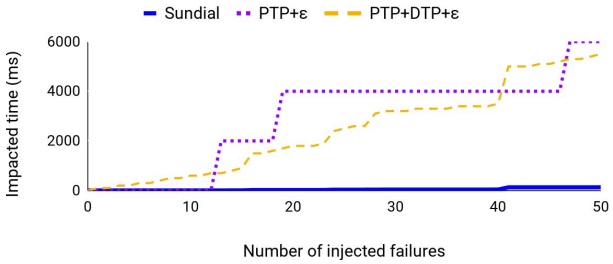


Figure 20: Blast radius of failures under different schemes. Impacted device time is the summation of per-device impacted time – duration when a device stops receiving sync-messages – over all devices .

to a maximum of 84ns and goes down in just 270μs (after the 185μs timeout, the next message is at 270μs). After that, ε is sawtooth between 30ns and 48ns, because its ε_{base} is set to $\varepsilon_{base,backup}$ by the local recovery, which is 30ns ($depth_{backup}$=6). Once the controller reconfigures the spanning tree, ε goes back to between 15ns and 33ns because its depth is 3. In PTP+ε, since the sync-message is dropped due to this failure, ε continues to increase for the next 2 seconds. Even if the sync-message was not dropped, ε for PTP+ε (w/o failure) remains high. PTP+DTP+ε's ε increases to 40μs and recovers to 20μs when the controller recovers the connectivity. However, even if the controller delay was lower (50 ms), it only reduces the peak ε to 30μs, but the normal ε is still around 20μs. Huygens+ε is not affected by failures, but its ε is normally very large (200μs at median and up to 400μs).

The behavior is similar under other failures – ε depends on the recovery time. For PTP+ε and PTP+DTP+ε, the recovery time depends on how long it takes for the controller to recover from it. For Sundial, the recovery time is much smaller as it's local. Any non-root failure recovery time is around 270μs, as is the case in Figure 19. The root failure takes slightly longer to recover from (365μs after the two timeouts) and ε increases to up to 103ns. The devices at different levels in the tree have slightly different ε (discussed in §6.1.4).

We now study the spatial and temporal impact range (blast radius) of failures. Figure 20 shows that Sundial's blast radius is very small. Even after 50 failures, the total impacted time summarized over all devices is only 131ms. The most significant jump happens when the root fails (40-th failure). PTP+ε and PTP+DTP+ε's blast radius is much higher owing to their longer recovery time. Note that more devices are affected by

failures under Sundial (401 in total) than under PTP+ε (3 in total) and PTP+DTP+ε (55 in total) as Sundial's backup-plan-based recovery can affect remote devices as well (those under the subtree of the failure). Even then the total impacted time for Sundial remains significantly smaller.

PTP+ε exhibits a step function because only failures occuring close to sync-interval boundaries affect it as the sync-interval of 2s is longer than the time to recover in most cases. The impact, however, is larger than in other schemes because it takes 2s for the next sync-message. PTP+DTP+ε's sync-interval is only 5μs and thus, every failure affects it. While Huygens+ε is not affected by connectivity failures, its ε remains high as shown before.

6.1.4 Microbenchmarks

How Sundial's different techniques improve ε. We zoom into details of how each technique improves ε. Specifically, starting with PTP+ε, we add (1) frequent sync-messages, (2) synchronous messaging, and (3) backup plan to it one by one, resulting in four schemes: PTP+ε, PTP+ε+freq_msg, PTP+ε+freq_msg+sync_msging, and Sundial itself.

Figure 21 shows the time series of ε during a link failure. Frequent sync-messages improve ε by an order of magnitude. Synchronous messaging further reduces ε during normal time as it helps each device detect connectivity failures: as long as a device receives a sync-message, it is connected to the root, so T_{last_sync} can be safely set to T_{last_msg}. Finally, adding the backup plan significantly speeds up the failure recovery – ε only increases for 270μs to a maximum of 84ns before the backup plan is activated, two orders of magnitude lower.

To show how Sundial's backup plan handles domain failures, we also run Sundial without considering domain failures (called Sundial w/o domain). We find that if a domain failure simultaneously takes down both the primary and backup parents of a device, the device's ε is like PTP+ε+freq_msg+sync_msging in Figure 21. This is expected because a down backup parent is equivalent to no backup parent. But if the failure domain is considered in the backup plan, ε is similar to Sundial in Figure 21, because the backup plan guarantees that no device loses both its primary and backup parents due to this domain failure. We also try another domain failure, which gradually takes down the primary and backup parents of a device, mimicing the domain failure that gradually takes down multiple devices or links (e.g., Figure 4). The result is similar.

Distribution of ε at different levels of the tree. We plot the maximum ε across devices at different depths, under different scenarios, shown in Figure 22. Root's ε is always 0. ε increases linearly with depth, which is expected as each level increments ε_{base} by 5ns.

6.2 Large-scale Simulations

We compare Sundial vs Marzullo's algorithm [27], an agreement algorithm for fault-tolerant clock-synchronization which is used by NTP [28] and TrueTime [12]. Marzullo's algorithm

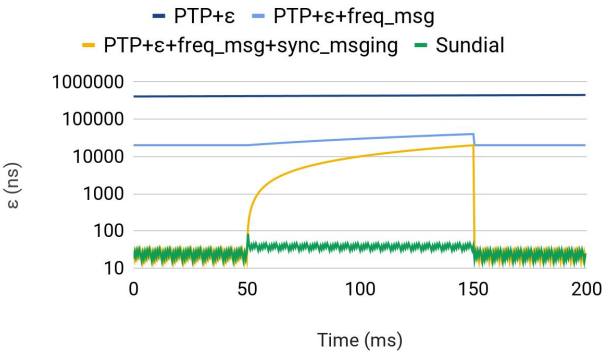


Figure 21: A link failure happens at 50 ms. The controller reacts to the failure at around 150 ms.

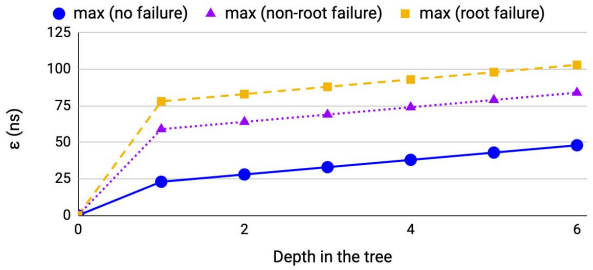


Figure 22: Distribution of ε at different levels in the tree.

also introduces time-uncertainty bound (ε) (called as error-bound in the original version). Since it is not supported in hardware due to its complexity, we use large scale simulations to demonstrate the performance characteristics.

Marzullo's algorithm synchronizes clocks through a mesh, so it can tolerate connectivity failures but has higher ε (§4.5). To reconcile the different time values and ε from multiple clocks, each node does intersection of time-uncertainty ranges of different clocks as the correct time should be within all ranges. A set of master clocks (1 or more clocks synchronized via GPS) serve as the source of synchronization, whose ε is always close to zero. Broken clocks can also be detected when the intersection result is empty. We simulate in a Jupiter topology [33] with 88,064 devices, where each node sends sync-messages to all its neighbors to maximize the tolerance to failures. We set 2 masters to tolerate master failures. The sync-interval is 90μs, same as Sundial. Figure 23 shows that during the normal time, Sundial has smaller ε than Marzullo's algorithm. Under failures, Marzullo's algorithm's ε is affected insignificantly. For Sundial, ε increases during failure recovery; the largest ε is 178ns, which is under the root failure.

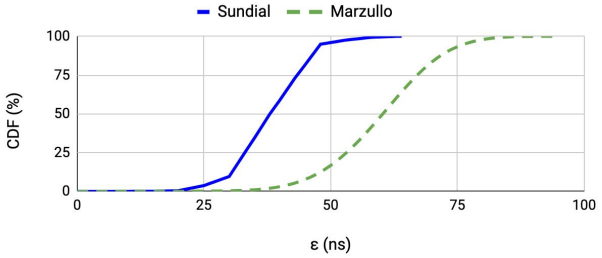


Figure 23: CDF of ε during normal time in Jupiter in simulation.

	DTP [21]	Huygens [13]	Marzullo [27]	PTP boundary clock [4]	Sundial
Message type	*Special PHY*	L3	Unspecified	L2	L2
Dealing with delay noises	*Neighbor*	Multi. msg	Unspecified	*Neighbor*	*Neighbor*
Synchronization structure	No master	Master, mesh	Master, mesh	Master, tree	*Master, mesh+tree*
Support time-uncertainty bound	No	No	*Yes*	No	*Yes*

Table 1: Design choices of state-of-the-art clock synchronization schemes. Italic options are the best.

6.3 Application Performance Improvement

Distributed transactional system. We evaluate the impact of smaller time-uncertainty bound using a load-test provided to us by Spanner team [12]. We run the load-test inside a datacenter. The load-test does 4KB transactions and we measure commit-wait gap – time to wait out time-uncertainty before committing the transaction. Results are in Table 2 where we show that our system improves performance by 3-4× not only in the median but also at the 99-th percentile.

	Baseline	With Sundial
Median	211μs	49μs
99-%ile	784μs	238μs

Table 2: Sundial improves commit-wait latency by 3-4× for Spanner running inside a datacenter.

Congestion Control. Delay-based congestion control such as Swift [17] is widely used in datacenters relying on end-to-end RTT measurements to control sending rate. A key challenge with such schemes is how to differentiate between forward and reverse-path congestion. As an example, congestion in the reverse path can also inflate RTT causing a sender to slow down even though there is no congestion in the forward path.[15] Synchronized clocks solve this problem as they enable the measurement of one-way delay (OWD) which can pinpoint the direction in which congestion is occurring.

We perform a microbenchmark with 3 servers – A, B and C with Swift congestion control. First, we only send traffic from A to B which achieves line-rate throughput. Next, we introduce reverse-path congestion by adding traffic from B and C to A. In Table 3, we observe A's throughput goes down to 50Gbps even though there was no congestion in the forward path. Replacing RTT with OWD as measured using Sundial resolves this completely and A continues to send at line rate.

	RTT	OWD
No reverse congestion	80.1 Gbps	80.5 Gbps
Reverse congestion	50.5 Gbps	80.9 Gbps

Table 3: Using one-way delay (OWD) improves throughput in the presence of reverse-path congestion.

7 Related Work

Other clock synchronization systems. Table 1 compares state-of-the-art solutions, in the design space outlined in §4.5.

DTP [21] introduces a specialized PHY layer to achieve zero bandwidth overhead of sync-messages. If this modified PHY can be standardized and productionized in the future, Sundial can readily benefit from it to have even lower sync-interval and ε. However, DTP does not reflect physical time since it doesn't have a master clock.

Huygens [13] does not synchronize switches, so it uses multiple messages between each pair to filter out noises. As a result, Huygens' sync-interval is limited, so it cannot achieve tight ε. Huygens' main advantage is that it is implemented completely in software and doesn't require hardware support (other than hardware timestamps) but it does not consider ε; and if incorporated, Huygens' ε is large primarily due to the large sync-interval. While it assumes clocks drift slowly during normal time, it cannot set a small *max_drift_rate* as the maximum drift is subject to failures (§3.2.1); otherwise it risks datacenter-wide application-level errors (e.g., inconsistent transactions), which is unacceptable.

Marzullo's algorithm [27] is the first to introduce ε but its ε is high because it sends messages through a mesh. PTP boundary clock [4] is based on a tree, and is not fault-tolerant.

Other solutions are too expensive (e.g., GPS [22]), too complex [18, 20, 31] or do not provide physical time [30, 34, 36].

Fault tolerance in other systems. In distributed systems and networking, fault tolerance is provided through redundancy [5, 10, 14, 19, 33, 38]. However, Sundial's backup plan cannot be chosen arbitrarily and needs to satisfy a set of properties (§4.2.1) to be generic to different types of failures.

Ethernet uses spanning tree protocols [2, 15] that can recompute a spanning tree in a distributed fashion after a failure, but they usually take up to a few seconds to converge [15].

8 Conclusion

Sundial is the first submicrosecond-level clock synchronization system that is resilient to failures. It uses hardware-software codesign to quickly detect failures and recover from them. Our evaluation shows that Sundial provides ∼100ns time-uncertainty bound under different types of failures, and improves performance in Spanner and in Swift.

9 Acknowledgements

We thank our shepherd Lorenzo Alvisi and OSDI reviewers for their helpful feedback. We also thank Arjun Singh, Jakov Seizovic, David Wetherall, David Dillow, Joe Zbiciak, and Xian Wu for the constructive feedback, Peter Cuy, Alex Iriza, and Bryant Chang for guidance on implementation, Shin Mao, Nanfang Li, and Ioannis Giannakopoulos for help on experimental evaluation.

[15] While prioritizing the ACK may solve the problem, it is impractical in production because of two reasons. (1) Network priorities are typically tied to business priorities; and we simply cannot send ACKs for lower business priority traffic on a higher network priority. (2) Sending ACKs on a higher network priority precludes ACK piggybacking on data packets, thereby increasing the packets-per-second to process. This is especially detrimental for CPU-efficient networking stacks such as PonyExpress in Snap [26].

References

[1] Broadcom: Timing over Packet (ToP) Processor for Precision Timing Applications. https://www.broadcom.com/products/embedded-and-networking-processors/communications/bcm53903.

[2] IEEE 802.1D Work Group, IEEE Standard for Local and Metropolitan Area Networks: Media Access Control (MAC) Bridges, 2004.

[3] CockroachDB, 2008. https://github.com/cockroachdb/.

[4] IEEE Standard 1588-2008, 2008. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4579757.

[5] Redundancy N+1, N+2 vs. 2N vs. 2N+1, 2014. https://www.datacenters.com/news/redundancy-n-1-n-2-vs-2n-vs-2n-1.

[6] IEEE 1588 PTP and Analytics on the Cisco Nexus 3548 Switch, 2017. https://www.cisco.com/c/en/us/products/collateral/switches/nexus-3000-series-switches/white-paper-c11-731501.html.

[7] Clock Oscillators Surface Mount Type KC3225L-P2/ KC3225L-P3 Series, 2018. https://global.kyocera.com/prdct/electro/pdf/kc3225l_p2p3_e.pdf.

[8] Juniper Precision Time Protocol Overview, 2020. https://www.juniper.net/documentation/en_US/junos/topics/concept/ptp-overview.html.

[9] Mellanox Precision Time Protocol, 2020. https://docs.mellanox.com/display/ONYXv381174/Precision+Time+Protocol.

[10] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 2008.

[11] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 1985.

[12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[13] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, 2018.

[14] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM*, 2009.

[15] New BPDU Handling. Understanding rapid spanning tree protocol (802.1 w). *Catalyst*, 2948(L3/4908G):L3.

[16] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on software Engineering*, 1987.

[17] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the ACM SIGCOMM*, 2020.

[18] Leslie Lamport. *Synchronizing time servers*. Digital, Systems Research Center, 1987.

[19] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[20] Leslie Lamport and Peter M Melliar-Smith. Byzantine clock synchronization. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 68–74, 1984.

[21] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, 2016.

[22] Wlodzimierz Lewandowski, Jacques Azoubib, and William J Klepczynski. Gps: Primary tool for time transfer. *Proceedings of the IEEE*, 87(1):163–172, 1999.

[23] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016.

[24] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center

networks. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, 2016.

[25] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and et al. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, 2019.

[26] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, and et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[27] Keith Marzullo and Susan Owicki. Maintaining the time in a distributed system. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 295–305, 1983.

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

[29] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *SIGCOMM Comput. Commun. Rev.*, 2015.

[30] Luca Schenato and Federico Fiorentin. Average timesynch: A consensus-based protocol for clock synchronization in wireless sensor networks. *Automatica*, 47(9):1878–1886, 2011.

[31] Ulrich Schmid. Synchronized utc for distributed real-time systems. *Annual Review in Automatic Programming*, 18:101–107, 1994.

[32] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojeviundefined, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. SIGMOD '19, 2019.

[33] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, 2015.

[34] Roberto Solis, Vivek S Borkar, and PR Kumar. A new distributed time synchronization protocol for multihop wireless networks. In *Proceedings of the 45th IEEE Conference on Decision and Control*, pages 2734–2739. IEEE, 2006.

[35] Robert Endre Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, 1976.

[36] Geoffrey Werner-Allen, Geetika Tewari, Ankit Patel, Matt Welsh, and Radhika Nagpal. Firefly-inspired sensor network synchronicity with realistic radio effects. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, SenSys '05, 2005.

[37] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, 2018.

[38] Mingyang Zhang, Radhika Niranjan Mysore, Sucha Supittayapornpong, and Ramesh Govindan. Understanding lifecycle management complexity of datacenter topologies. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[39] Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C. Mogul, and Amin Vahdat. Minimal rewiring: Efficient live expansion for clos data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[40] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015.