Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases

Sandeep Kulkarni*, Murat Demirbas**, Deepak Madeppa**, Bharadwaj Avva**, and Marcelo Leone*

*Michigan State University
**University at Buffalo, SUNY

Abstract

There is a gap between the theory and practice of distributed systems in terms of the use of time. The theory of distributed systems shunned the notion of time, and introduced "causality tracking" as a clean abstraction to reason about concurrency. The practical systems employed physical time (NTP) information but in a best effort manner due to the difficulty of achieving tight clock synchronization. In an effort to bridge this gap and reconcile the theory and practice of distributed systems on the topic of time, we propose a hybrid logical clock, HLC, that combines the best of logical clocks and physical clocks. HLC captures the causality relationship like logical clocks, and enables easy identification of consistent snapshots in distributed systems. Dually, HLC can be used in lieu of physical/NTP clocks since it maintains its logical clock to be always close to the NTP clock. Moreover HLC fits in to 64 bits NTP timestamp format, and is masking tolerant to NTP kinks and uncertainties. We show that HLC has many benefits for *wait-free* transaction ordering and performing snapshot reads in multiversion globally distributed databases.

1 Introduction

1.1 Brief history of time

> Time is an illusion.
>
> – Albert Einstein

Logical clock (LC). LC [12] was proposed in 1978 by Lamport as a way of timestamping and ordering events in a distributed system. LC is divorced from physical time (e.g., NTP clocks): the nodes do not have access to clocks, there is no bound on message delay and on the speed/rate of processing of nodes. The causality relationship captured, called happened-before (**hb**), is defined based on passing of information, rather than passing of time.[1] While being beneficial for the theory of distributed systems, LC is impractical for today's distributed systems: 1) Using LC, it is not possible to query events in relation to physical time. 2) For capturing **hb**, LC assumes that all communication occurs in the present system and there are no backchannels. This is obsolete for today's integrated, loosely-coupled system of systems.

In 1988, the vector clock (VC) [7, 19] was proposed to maintain a vectorized version of LC. VC maintains a vector at each node which tracks the knowledge this node has about the logical clocks of other nodes. While LC finds one consistent snapshot (that with same LC values at all nodes involved), VC finds all possible consistent snapshots, which is useful for debugging applications. In Figure 1, while LC would find (a,w) as a consistent cut, VC would also identify (b,w), (c,w) as consistent cuts. Unfortunately, the space requirement of VC is on the order of nodes in the system, and is prohibitive.

Physical Time (PT). PT leverages on physical clocks at nodes that are synchronized using the Network Time Protocol (NTP) [20]. Since perfect clock synchronization is infeasible for a distributed system, there are uncertainty intervals associated with PT. While PT avoids the disadvantages of LC by using physical time for timestamping, it introduces new disadvantages: 1) When the uncertainty intervals are overlapping, PT cannot order events. NTP can usually maintain time to within tens of milliseconds over the public Internet, and can achieve one millisecond accuracy in local area networks under ideal condi-

[1] Event e happened-before event f, if e and f are on the same node and e comes earlier than f, or e is a send event and f is the corresponding receive event, or is defined transitively based on the previous two.

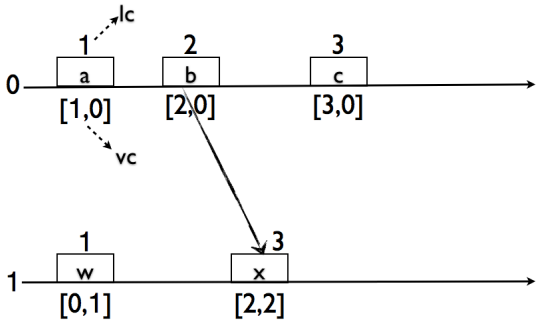

Figure 1: LC and VC timestamping

tions, however, asymmetric routes and network congestion can occasionally cause errors of 100 ms or more. 2) PT has several kinks such as leap seconds [13, 14] and non-monotonic updates to POSIX time [8] which may cause the timestamps to go backwards.

TrueTime (TT). TrueTime is proposed recently by Google for developing Spanner [2], a multiversion distributed database. TT relies on a well engineered tight clock synchronization available at all nodes thanks to GPS clocks and atomic clocks made available at each cluster. While TT avoids some of the disadvantages of LC/VC/PT, it introduces new disadvantages: 1) TT requires special hardware and a custom-build tight clock synchronization protocol, which is infeasible for many systems (e.g., using leased nodes from public cloud providers). 2) If TT is used for ordering events that respect causality then it is essential that if e hb f then $tt.e < tt.f$. Since TT is purely based on clock synchronization of physical clocks, to satisfy this constraint, Spanner delays event f when necessary. Such delays and reduced concurrency are prohibitive especially under looser clock synchronization.

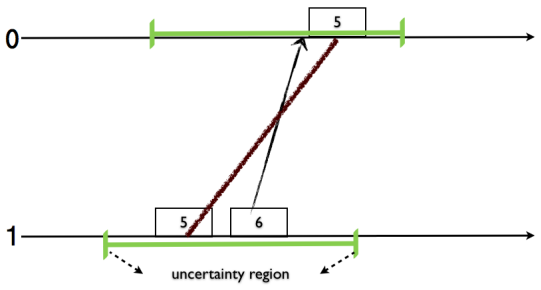

Figure 2: Not waiting out the uncertainty regions in TT may result in inconsistent snapshots

HybridTime (HT). HT, which combines VC and PT clocks, was proposed for solving the stabilizing causal deterministic merge problem [10]. HT maintains a VC at each node which includes knowledge this node has about the PT clocks of other nodes. HT exploits the clock synchronization assumption of PT clocks to trim entries from VC and reduces the overhead of causality tracking. In practice the size of HT at a node would only depend on the number of nodes that communicated with that node within the last ϵ time, where ϵ denotes the clock synchronization uncertainty. Recently, Demirbas and Kulkarni [3] explored how HT can be adopted to solve the consistent snapshot problem in Spanner [2].

1.2 Contributions of this work

In this paper we aim to bridge the gap between the theory (LC) and practice (PT) of timekeeping and timestamping in distributed systems and to provide guarantees that generalize and improve that of TT.

- We present a logical clock version of HT, which we name as Hybrid Logical Clocks (HLC). HLC refines both the physical clock (similar to PT and TT) and the logical clock (similar to LC). HLC maintains its logical clock to be always close to the NTP clock, and hence, HLC can be used in lieu of physical/NTP clock in several applications such as snapshot reads in distributed key value stores and databases. Most importantly, HLC preserves the property of logical clocks (e hb $f \Rightarrow hlc.e < hlc.f$) and as such HLC can identify and return consistent global snapshots without needing to wait out clock synchronization uncertainties and without needing prior coordination, in a posteriori fashion.
- HLC is backwards compatible with NTP, and fits in the 64 bits NTP timestamp format. Moreover, HLC works as a superposition on the NTP protocol (i.e., HLC only reads the physical clocks and does not update them) so HLC can run alongside applications using NTP without any interference. Furthermore HLC is general and does not require a server-client architecture. HLC works for a peer-to-peer node setup across WAN deployment, and allows nodes to use different NTP servers.[2] In Section 3, we present the HLC algorithm and prove a tight bound on the space requirements of HLC and show that the bound suffices for HLC to capture the LC property for causal reasoning.

[2]In fact HLC can work with ad hoc clock synchronization protocols [17] and is not bound to NTP.

- HLC provides masking tolerance to common NTP problems (including nonmonotonous time updates) and can make progress and capture causality information even when time synchronization has degraded. HLC is also self-stabilizing fault-tolerant and is resilient to arbitrary corruptions of the clock variables, as we discuss in Section 4.

- We implement HLC and provide experiment results of HLC deployments under various deployment scenarios. In Section 5, we show that even under stress-testing, HLC is bounded and the size of the clocks remain small. These practical bounds are much smaller than the theoretical bounds proved in our analysis. Our HLC implementation is made available in an anonymized manner at `https://github.com/AugmentedTimeProject`

- HLC has direct applications in identifying consistent snapshots in distributed databases [2, 11, 15, 16, 22, 24]. It is also useful in many distributed systems protocols including causal message logging in distributed systems [1], Byzantine fault-tolerance protocols [9], distributed debugging [21], distributed filesystems [18], and distributed transactions [25]. In Section 6, we showcase the benefits of HLC for snapshot reads in distributed databases.

2 Preliminaries

A distributed system consists of a set of nodes whose number may change over time. Each node can perform three types of actions, a send action, a receive action, and a local action. The goal of a timestamping algorithm is to assign a timestamp to each event. We denote a timestamping algorithm with an all capital letters name, and the timestamp assigned by this algorithm by the corresponding lower case name. E.g., we use LC to denote the logical clock algorithm by Lamport [12], and use $lc.e$ to denote the timestamp assigned to event e by this algorithm.

The notion of happened before hb captures the causal relation between events in the system. As defined in [12], event e happened before event f (denoted by $e \ \mathsf{hb} \ f$) is a transitive relation that respects the following: e and f are events on the same node and e occurred before f, or e is a send event and f is the corresponding receive event. We say that e and f are concurrent, denoted by $e||f$, iff $\neg(e \ \mathsf{hb} \ f) \wedge \neg(f \ \mathsf{hb} \ e)$.

Based on the existing results in the literature, the following are true:

$$e \ \mathsf{hb} \ f \ \Rightarrow \ lc.e < lc.f$$
$$lc.e = lc.f \ \Rightarrow \ e||f$$
$$e \ \mathsf{hb} \ f \ \Leftrightarrow \ vc.e < vc.f$$

However, the following claims are not true:

$$e \ \mathsf{hb} \ f \ \Leftarrow \ lc.e < lc.f$$
$$lc.e = lc.f \ \Leftarrow \ e||f$$
$$e \ \mathsf{hb} \ f \ \Rightarrow \ pt.e < pt.f$$

3 HLC: Hybrid Logical Clocks

In this section, we introduce our HLC algorithm starting with a naive solution first. We then prove correctness and tight bounds on HLC. We also elaborate on the useful features of the HLC for distributed systems.

3.1 Problem statement

The goal of HLC is to provide one-way causality detection similar to that provided by LC, while maintaining the clock value to be always close to the physical/NTP clock. The formal problem statement for HLC is as follows.

Given a distributed system, assign each event e a timestamp, $l.e$, such that

1. $e \ \mathsf{hb} \ f \Rightarrow l.e < l.f$,
2. Space requirement for $l.e$ is $O(1)$ integers,
3. $l.e$ is represented with bounded space,
4. $l.e$ is *close* to $pt.e$, i.e., $|l.e - pt.e|$ is bounded.

The first requirement captures one-way causality information provided by HLC. The second requirement captures that the space required for $l.e$ is $O(1)$ integers. To prevent encoding of several integers into one large integer, we require that any update of $l.e$ is achieved by $O(1)$ operations. The third requirement captures that the space required to represent $l.e$ is bounded, i.e., it does not grow in an unbounded fashion. In practice, we like $l.e$ to be the size of $pt.e$, which is 64 bits in the NTP protocol.

Finally, the last requirement states that $l.e$ should be close to $pt.e$. This enables us to utilize HLC in place of PT. To illustrate this consider the case where the designer wants to take a snapshot at (physical) time t. Given that physical clocks are not perfectly synchronized, it is not possible to get a consistent snapshot by just reading state at different nodes at time t as shown in Figure 2. On the other hand, using HLC we can obtain such a snapshot by taking the snapshot of every node at *logical time* t. Such a snapshot is guaranteed to be consistent, because from the HLC requirement 1 we have $l.e = l.f \ \Rightarrow \ e||f$. In Section 6, we discuss in more detail how HLC enables users

Initially $lc.j := 0$

Send or local event
$l.j := max(l.j + 1, pt.j)$
Timestamp with $l.j$

Receive event of message m
$l.j := max(l.j + 1, l.m + 1, pt.j)$
Timestamp with $l.j$

Figure 3: Naive HLC algorithm for node j

to take uncoordinated a-posteriori consistent snapshots of the distributed system state.

3.2 Description of the Naive Algorithm

Given the goal that $l.e$ should be close to $pt.e$, in the naive algorithm we begin with the rule: *for any event* e, $l.e \geq pt.e$. We design our algorithm as shown in Figure 3. This algorithm works similar to LC. Initially all l values are set to 0. When a send event, say f, is created on node j, we set $l.f$ to be $max(l.e+1, pt.j)$, where e is the previous event on node j. This ensures $l.e < l.f$. It also ensures that $l.f \geq pt.f$. Likewise, when a receive event f is created on node j, $l.f$ is set to $max(l.e+1, l.m+1, pt.j)$, where $l.e$ is the timestamp of the previous event on j, and $l.m$ is the timestamp of the message (and, hence, the send event). This ensures that $l.e < l.f$ and $l.m < l.f$.

It is easy to see that the algorithm in Figure 3 satisfies the first two requirements in the problem statement. However, this naive algorithm violates the fourth requirement, which also leads to a violation of the third requirement for bounded space representation. To show the violation of the fourth requirement, we point to the counterexample in Figure 4 which shows how $|l.e - pt.e|$ grows in an unbounded fashion. The messaging loop among nodes 1, 2, and 3 can be repeated forever, and at each turn of the loop the drift between logical clock and physical clock (the $l - pt$ difference) will keep growing.

The root of the unbounded drift problem is due to the naive algorithm using l to maintain both the maximum of pt values seen so far and the logical clock increments from new events (local, send, receive). This makes the clocks lose information: it becomes unclear if the new l value came from pt (as in the message from node 0 to node 1) or from causality (as is the case for the rest of messages). As such, there is no suitable place to reset l value to bound the $l-pt$ difference, because resetting l may lead to losing the hb relation, and, hence, a violation of requirement 1.

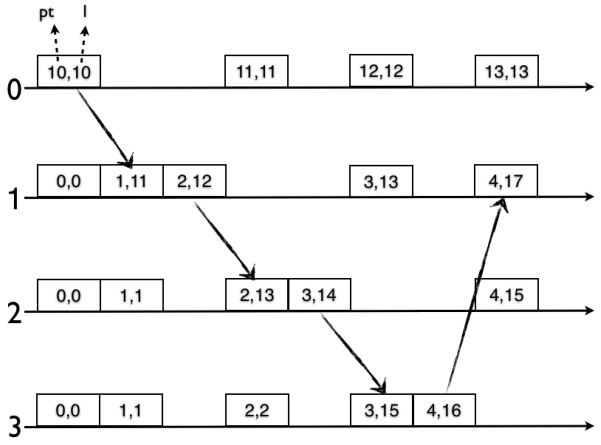

Figure 4: Counter example

Note that the counterexample holds even with the requirement that the physical clock of a node is incremented by at least one between any two events on that node. Figure 4 satisfies this constraint between pt and l, yet still $|l - pt|$ keeps growing unboundedly. However, there are conditions under which the counterexample does not work, and the naive algorithm suffices for solving the HLC problem. If we assume that the time for send event and receive event is long enough so that the physical clock of *every* node is incremented by at least one, then the counterexample on Figure 4 fails, and the naive algorithm would be able to maintain $|l - pt|$ bounded.

Instead of depending on assumptions on physical clock rate and event generation rate *across all nodes* in the system for proving the correctness and boundedness of HLC, we show how to properly implement HLC next.

3.3 HLC Algorithm

> All problems in computer science can be solved by another level of indirection. –David Wheeler

We use our observations from the counterexample to develop the correct HLC algorithm. In this algorithm, the $l.j$ in the naive algorithm is expanded to two parts: $l.j$ and $c.j$. The first part $l.j$ is introduced as a level of indirection to maintain the maximum of pt information learned so far, and c is used for capturing causality updates only when l values are equal.

In contrast to the naive algorithm where there was no suitable place to reset l without violating hb , in the HLC algorithm, we can reset c when the information

Initially $l.j := 0; c.j := 0$

Send or local event

$l'.j := l.j;$
$l.j := max(l'.j, pt.j);$
If $(l.j = l'.j)$ then $c.j := c.j + 1$
  Else $c.j := 0;$
Timestamp with $l.j, c.j$

Receive event of message m

$l'.j := l.j;$
$l.j := max(l'.j, l.m, pt.j);$
If $(l.j = l'j = l.m)$ then $c.j := max(c.j, c.m)+1$
  Elseif $(l.j = l'.j)$ then $c.j := c.j + 1$
  Elseif $(l.j = l.m)$ then $c.j := c.m + 1$
  Else $c.j := 0$
Timestamp with $l.j, c.j$

Figure 5: HLC algorithm for node j

heard about maximum pt catches up or goes ahead of l. Since l denotes the maximum pt heard among nodes and is not continually incremented with each event, within a bounded time, either one of the following is guaranteed to occur: 1) a node receives a message with a larger l, and its l is updated and c is reset to reflect this, or 2) if the node does not hear from other nodes, then its l stays the same, and its pt will catch up and update its l, and reset the c.

The HLC algorithm is as shown in Figure 5. Initially, l and c values are set to 0. When a new send event f is created, $l.j$ is set to $max(l.e, pt.j)$, where e is the previous event on j. Similar to the naive algorithm, this ensures that $l.j \geq pt.j$. However, because we have removed the "+1", it is possible that $l.e$ equals $l.f$. To deal with this, we utilize the value of $c.j$. By incrementing $c.j$, we ensure that $\langle l.e, c.e\rangle < \langle l.f, c.f\rangle$ is true with lexicographic comparison.[3] If $l.e$ differs from $l.f$ then $c.j$ is reset, and this allows us to guarantee that c values remain bounded. When a new receive event is created, $l.j$ is set to $max(l.e, l.m, pt.j)$. Now, depending on whether $l.j$ equals $l.e$, $l.m$, both or neither, $c.j$ is set.

Let's reconsider the counterexample to the naive algorithm. This example replayed with the HLC algorithm is shown in Figure 6. When we continue the loop among nodes 1, 2, 3, we see that pt at nodes 1, 2 and 3 catches up and exceeds $l = 10$ and resets c to 0. This keeps the c variable bounded at each node.

To prove the correctness of the HLC algorithm, first we show that it satisfies requirement 1 and can be used

[3] $(a, b) < (c, d)$ iff $((a < c) \ \vee \ ((a = c) \wedge (b < d)))$

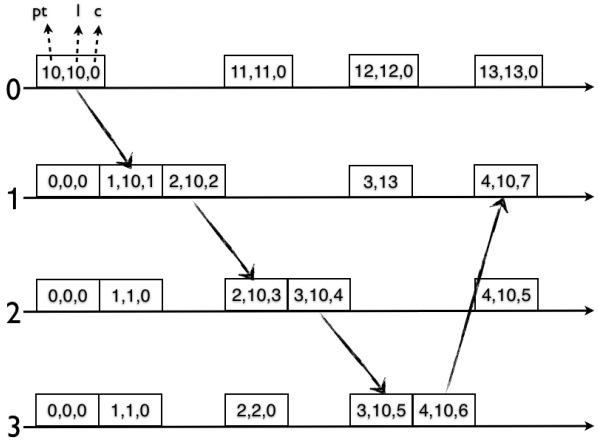

Figure 6: The trace in Figure 4 re-timestamped with HLC

for LC. This follows easily from how l and c values are updated in the algorithm.

Theorem 1. *For any two events* e *and* f, e hb $f \Rightarrow (l.e, c.e) < (l.f, c.f)$ □

Next, we show that HLC satisfies requirement 4, which asserts that the HLC value is close to PT. Based on how l is updated in the algorithm, it is easy to see Theorem 2.

Theorem 2. *For any event* f, $l.f \geq pt.f$ □

Theorem 3. $l.f$ *denotes the maximum clock value that* f *is aware of. In other words,*

$l.f > pt.f \Rightarrow (\exists g : g \text{ hb } f \wedge pt.g = l.f)$

Proof. We prove this by induction, as new events are created. In the initial state, the statement is trivially satisfied.

Consider the case where a new event f is created.

- If f is a send event and e is the previous event, then by induction, we have

  $l.e > pt.e \Rightarrow (\exists g : g \text{ hb } e \wedge pt.g = l.e)$

  Furthermore, from the HLC algorithm, if $l.f > pt.f$ then $l.f = l.e$. Also, e hb f is true. Hence, we have

  $l.f > pt.f \Rightarrow (\exists g : g \text{ hb } f \wedge pt.g = l.f)$.

- If f is a receive event. Let e be the previous event on the same node and m the received message.

  Once again, if $l.f > pt.f$ is true then $l.f$ equals $l.e$ or $l.m$. The analysis of each of these cases is similar to the previous case. Hence, we have

  $l.f > pt.f \Rightarrow (\exists g : g \text{ hb } f \wedge pt.g = l.f)$.

□

Using Theorem 3, we can show that $|l - pt|$ is bounded.

Corollary 1. *For any event f, $|l.f - pt.f| \leq \epsilon$*

Proof. We cannot have two events e and f such that e hb f and $pt.e > pt.f + \epsilon$ due to clock synchronization constraints. Hence, from Theorem 3, this theorem follows. □

Finally, we prove requirement 3, by showing that c value of HLC is bounded as well. To this end, we extend Theorem 3 to identify the relation of c and events created at a particular time. As we show in Theorem 4, $c.f$ captures information regarding events created at time $l.f$.

Theorem 4. *For any event f,*

$$\begin{aligned} & c.f = k \wedge k > 0 \\ \Rightarrow \quad & (\exists g_1, g_2, \cdots, g_k : \\ & \quad (\forall j : 1 \leq j < k : g_i \text{ hb } g_{i+1}) \\ & \wedge \ (\forall j : 1 \leq j \leq k : l.(g_i) = l.f) \\ & \wedge \ g_k \text{ hb } f) \end{aligned}$$

Proof. We prove this by induction. This is trivially satisfied in the initial state. Also, if $c.f$ is set to 0 then this statement is trivially satisfied.

In creation of send event, $c.f$ is set to $c.e + 1$ only if $l.e$ equals $l.f$. By induction, there exists a sequence of length $c.e$ that satisfies the statement of the theorem. Moreover, e hb f and $\neg(e \text{ hb } e)$. Hence, there exists a sequence of $c.e + 1$ (=$c.f$) that satisfies the statement of the theorem.

A similar analysis also applies for the receive event when $c.f$ is set to $c.e + 1$ or $c.m + 1$. □

From Theorem 4, the following two corollaries follow.

Corollary 2. *For any event f,*

$$c.f \leq |\{g : g \text{ hb } f \wedge l.g = l.f)\}|.$$

Corollary 3. *For any event f, $c.f \leq N * (\epsilon + 1)$*

Proof. From Corollary 2, for any event f, $c.f \leq |\{g : g \text{ hb } f \wedge l.g = l.f)\}|$. Also, from Theorem 2, $l.g \geq pt.g$. Also, by clock synchronization assumption of g hb f then $pt.g \leq pt.f + \epsilon$. Hence, the only events that can fall into the set $\{g : g \text{ hb } f \wedge l.g = l.f)\}$ are those that were created when physical time of the node that created them was between $[l.f, l.f + \epsilon]$. By our constraint that physical clock of a node is incremented by at least one between any two events on that node, there are at most $\epsilon + 1$ such events on any one node. Hence, the corollary follows. □

While the above bound is almost tight, adding a small reasonable assumption can substantially reduce the bound on c, and thereby reducing the space that needs to be allocated for that.

Assumption to reduce the bound on c further: We assume that the time for message transmission is long enough so that the physical clock of every node is incremented by at least d, where d is a given parameter.

Now, consider the situation where $c.f = k, k > 0$, at node j. From the above assumption, from Theorem 4, we have a sequence of k events $g_1, g_2, \cdots, g_k$ that satisfy the conditions in Theorem 4. In other words, $l.(g_1) = l.f$. Let l denote the node where g_1 was created. Hence, when g_1 was created, $pt.l$ was at least equal to $l.f$. By assumption about clock synchronization, when f is created $pt.l$ is at least $l.f + (k-1) * d$. Given clock synchronization constraints, this must be less than $pt.f + \epsilon$. Simplifying this, k is less than $\epsilon/d + 1 + (pt.f - l.f)$. From Theorem 2, we have

Corollary 4. *Under the assumption made above, $c.f$ is at most $\epsilon/d + 1$.*

Recall that for $d \geq 1$, the counterexample in Figure 4 does not hold, and the naive algorithm would become boundable and also satisfy the HLC requirements. The difference between the HLC algorithm and the naive algorithm is that the HLC algorithm did not need this assumption to show that it is bounded, but only to reduce the size of the bound.

3.4 Properties of HLC

HLC algorithm is designed for arbitrary distributed architecture and is also readily applicable to other environments such as the client-server model.

We intentionally chose to implement HLC as a superposition on NTP. In other words, HLC only reads the physical clock but does not update it. Hence, if a node receives a message whose timestamp is higher, we maintain this information via l and c instead of changing the physical clock. This is crucial in ensuring that other programs that use NTP alone are not affected. This also avoids the potential problem where clocks of nodes are synchronized with each other even though they drift substantially from *real wall-clock*. Furthermore, there are impossibility results showing that accepting even tiny unsynchronization to adjust the clocks can lead to diverging clocks [6]. Finally, while HLC utilizes NTP for synchronization, it does not depend on it. In particular, even when physical clocks utilize any ad hoc clock synchronization algorithm [17],

HLC can be superposed on top of such a service, so can also be used in ad hoc networks.

4 Resilience of HLC

4.1 Self-stabilization

Here we discuss how we design self-stabilizing [4] fault-tolerance to HLC, which enables HLC to be eventually restored to a legitimate state, even when HLC is perturbed/corrupted to an arbitrary state.

Stabilization of HLC rests on the superposition property of HLC on NTP clocks. Since HLC does not modify the NTP clock, it does not interfere with the NTP correcting/synchronizing the physical clock of the node. Once the physical/NTP clock stabilizes, HLC can be corrected based on observations in Theorem 2 and Corollaries 3 and 2. These results identify the maximum permitted value of $l-pt$ and the maximum value of c. In the event of extreme clock errors by NTP or transient memory corruption, the application may reach a state where these bounds are violated. In that case, we take the physical clock as the authority, and reset l and c values to pt and 0 respectively. In other words the stabilization of HLC follows that of stabilization of pt via NTP clock.

In order to contain the spread of corruptions due to bad HLC values, we have a rule to ignore out of bounds messages. We simply ignore reception of messages that cause l value to diverge too much from pt. This prevention action fires if the sender of the message is providing a clock value that is significantly higher suggesting the possibility of corrupted clock. In order to contain corruptions to c, we make its space bounded, so that even when it is corrupted, its corruption space is limited. This way c would in the worst case roll over, or more likely, c would be reset to an appropriate value as a result of l being assigned a new value from pt or from another l received in a message.

Note that both the reset correction action and the ignore out-of-bounds message action are local correction actions at a node. If HLC fires either of these actions, it also logs the offending entries for inspection and raises an exception to notify the administrator.

4.2 Masking of synchronization errors

In order to make HLC resilient to common NTP synchronization errors, we assign sufficiently large space to $l-pt$ drift so that most (99.9%) NTP kinks can be masked smoothly. While Theorem 2 and Corollaries 3 and 2 state that $l - pt$ stay within ϵ the clock synchronization uncertainty (crudely two times the NTP offset value), we set a very conservative value, Δ, on the $l - pt$ bound. The bound Δ can be set to a constant factor of ϵ, and even on the order of seconds depending on the application semantics. This way we tolerate and mask common NTP clock synchronization errors within normal operation of HLC. And when Δ bound is violated, the local reset correction action and the ignore message prevention action fire as discussed in the previous subsection.

Using this approach, HLC is robust to stragglers, nodes with pt stuck slightly in the past. Consider a node that lost connection to its NTP server and its clock started drifting behind the NTP time. Such a straggler can still keep up with the system for some time and maintain up-to-date and bounded HLC time: As long as it receives messages from other nodes, it will learn new/higher l values and adopt them. This node will increment its c by 1 when it does not adopt a new l value, but this does not cause the c rise excessively for the other nodes in the system. Even if this node sends a message with high c number, the other nodes will have up-to-date time and ignore that c and will use $c = 0$. Similarly, HLC is also robust to the rushers, nodes with pt slightly ahead of others. The masking tolerance of HLC makes it especially useful for last write wins (LWW) database systems like Cassandra [8, 14]. We investigate this tolerance empirically in the next section.

5 Experiments

5.1 AWS deployment results

The experiments used Amazon AWS xlarge instances running Ubuntu 14.04. The machines were synchronized to a stratum 2 NTP server, `0.ubuntu.pool.ntp.org`. In our basic setup, we programmed all the instances to send messages to each other continuously using TCP sockets, and in a separate thread receive messages addressed to them. The total messages sent range from 75,000 to 425,000.

Using the basic setup (all nodes are senders and sending to each other) within the same AWS region, we get the following results. The value "c" indicates that the value of the c component of the HLC at the nodes. The remaining columns show the frequency: the percentage of times the HLC at the nodes had the corresponding c values out of the total number of events. For each setup, we collected data with two different NTP synchronization levels, indicated by the average offset of nodes' clocks from NTP. When we allow the NTP daemons at the nodes more time

(a couple hours) to synchronize, we get lower NTP offset values. We used "ntpdc -c loopinfo" and "ntpdc -c kerninfo" calls to obtain the NTP offset information at the nodes.

Using 4 m1.xlarge nodes

c	offset=5ms	offset=1.5ms
0	83.90 %	83.66 %
1	12.12 %	12.03 %
2	3.37 %	4.09 %
3	0.24 %	0.21 %

The experiments with 4 nodes show that the value of c remains very low, less than 4. This is a much lower bound than the worst case possible theoretical bound we proved in Section 3. We also see that the improved NTP synchronization helps move the c distribution toward lower values, but this effect becomes more visible in the 8 and 16 node experiments. With the looser NTP synchronization, with average offset 5 ms, the maximum $l - pt$ difference was observed to be 21.7 ms. The 90th percentile of $l - pt$ values correspond to 7.8 ms, with their average value computed to be 0.2 ms. With the tighter NTP synchronization, with average offset 1.5 ms, the maximum $l - pt$ difference was observed to be 20.3 ms. The 90th percentile of $l - pt$ values correspond to 8.1 ms, with their average value computed to be 0.2 ms.

Using 8 m1.xlarge nodes

c	offset=9ms	offset=3ms
0	65.56 %	91.18 %
1	15.39 %	8.82 %
2	8.14 %	0 %
3	5.90 %	
4	2.74 %	
5	1.39 %	
6	0.56 %	
7	0.20 %	
8	0.08 %	
9	0.03 %	

The experiments with 8 nodes highlights the lowered c values due to improved NTP synchronization. For the experiments with average NTP offset 9ms, the maximum $l - pt$ difference was observed to be 107.9 ms. The 90th percentile of $l - pt$ values correspond to 41.4 ms, with their average value computed to be 4.2 ms. For the experiments with average NTP offset 3ms, the maximum $l - pt$ difference was observed to be 7.4 ms. The 90th percentile of $l - pt$ values correspond to 0.1 ms, with their average value computed to be 0 ms.

Using 16 m1.xlarge nodes

c	offset=16ms	offset=6ms
0	66.96 %	75.43 %
1	19.40 %	18.51 %
2	7.50 %	3.83 %
3	4.59 %	1.84 %
4	1.76 %	0.32 %
5	0.61 %	0.06 %
6	0.14 %	0.01 %
7	0.02 %	

The 16 node experiments also showed very low c values despite all nodes sending to each other at practically at the wire speed. For the experiments with average NTP offset 16ms, the maximum $l - pt$ difference was observed to be 90.5 ms. The 90th percentile of $l - pt$ values correspond to 25.2 ms, with their average value computed to be 2.3 ms. For the experiments with average NTP offset 6ms, the maximum $l - pt$ difference was observed to be 46.8 ms. The 90th percentile of $l - pt$ values correspond to 8.4 ms, with their average value computed to be 0.3 ms.

WAN deployment results. We deployed our HLC testing experiments on a WAN environment as well. Specifically, we used 4 m1.xlarge instances each one located at a different AWS region: Ireland, US East, US West and Tokyo. Our results show that with 3ms NTP offset, the $c = 0$ values constitute about 95% of the cases and $c = 1$ constitute the remaining 5%. These values are much lower than the corresponding values for the single datacenter deployment. The maximum $l - pt$ difference remained extremely low, about 0.02 ms, and the 90th percentile of $l - pt$ values corresponded to 0. These values are again much lower than the corresponding values for the single datacenter deployment.

The reason for seeing very low $l - pt$ and c values in the WAN deployment is because the message communication delays across WAN are much larger than the ϵ, the clock synchronization uncertainty. As a result, when a message is received, its l timestamp is already in the past and is smaller than the l value at the receiver which is updated by its pt. Since the single cluster deployment with short message delays is the most demanding scenario in terms of HLC testing we focused on those results in our presentation.

5.2 Stress testing and resilience evaluation in simulation

To further analyze the resiliency of HLC, we evaluated it in scenarios where it will be stressed, e.g., where the event rate is too high and where the clock synchronization

is significantly degraded. In our simulations, we considered the case where the event creation rate was 1 event per millisecond and clock drift varies from $10ms$ to $100ms$. Given the relation between l and pt from Theorem 2, the drift between l and pt is limited to the clock drift. Hence, we focus on values of c for different events.

In these simulations, a node is allowed to advance its physical clock by 1ms as long as its clock drift does not exceed beyond ϵ. If a node is allowed to advance its physical clock then it increases it with a 50% probability. When it advances its clock, it can send a message with certain probability (All simulations in this section correspond to the case where this probability is 100%). We deliver this message at the earliest possible feasible time, essentially making delivery time to be 0. The results are as shown in Figure 7. As shown in these figures, the distribution of c values was fairly independent of the value of ϵ. Moreover, for more than 99% of events, the c value was 4 or less. Less than 1% of events had c values of 5-8.

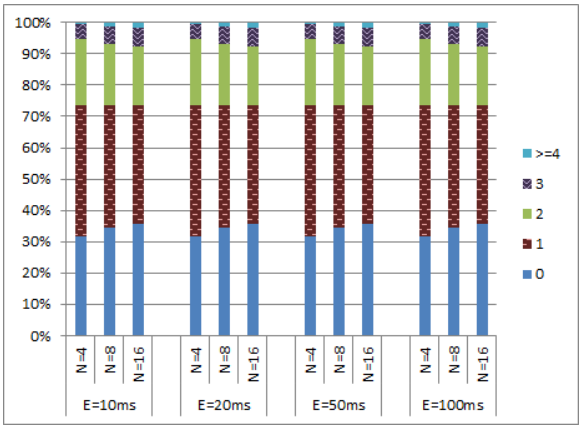

Figure 7: Distribution of c values for varying ϵ

To evaluate HLC in the presence of degraded clock synchronization, we added a straggler node to the system. This node was permitted to violate clock drift constraints by always staying behind. We consider the case where the straggler just resides at the end of permissible boundary, i.e., its clock drift from the highest clock is ϵ. We also consider the case where straggler violates the clock drift constraints entirely and it is upto 5ϵ behind the maximum clock. The results are as shown in Figures 8 and 9. Even with the straggler, the c value for 99% events was 4 or less. However, in these simulations, significantly higher c values were observed for some events. In particular, for the case where the straggler remained just at the end of permissible boundary, events with c value of upto 97 were observed at the straggler node. For the case where the

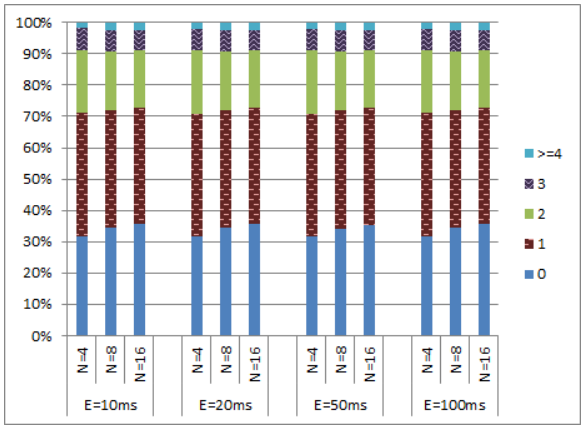

Figure 8: Distribution of c values with a straggler

straggler was permitted to drift by 5ϵ, c value of upto 514 was observed again only at the straggler node. The straggler node did not raise the c vluaes of other nodes in the system.

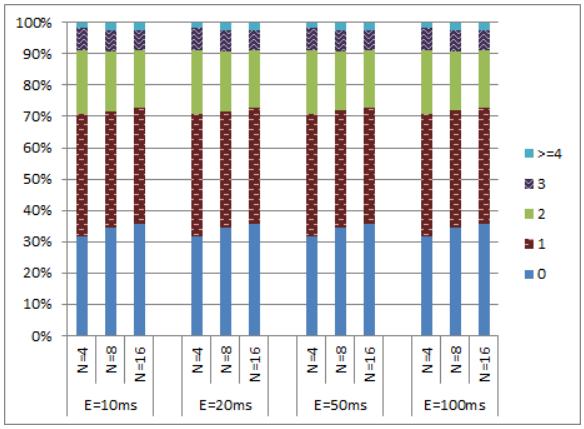

Figure 9: Distribution of c values with a straggler out of sync by 5 ϵ

We also conducted the experiments where we had a rusher, a node that was excessively ahead. Figures 10 and 11 demonstrate the results. The maximum c value observed in these experiments was 8. And, the number of events with c value greater than 3 is less than 1%.

As a result of these experiments we conclude that the straggler node affects the c value more than the rusher node, but only for itself. In our experiments, each node selects the sender randomly with uniform distribution. Hence, messages sent by the rusher node do not have a

significant cumulative effect. However, messages sent by all nodes to the straggler node causes its c value to grow.

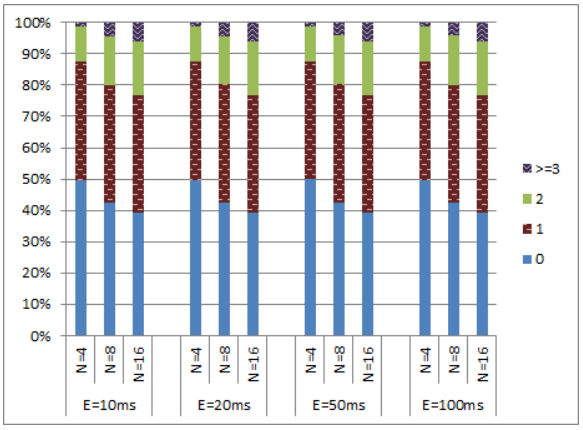

Figure 10: Distribution of c values with a rusher

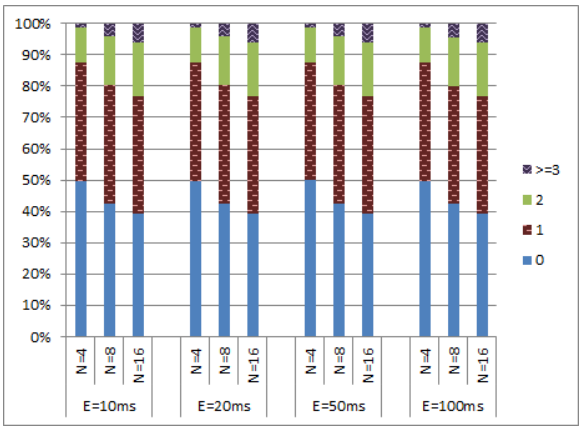

Figure 11: Distribution of c values with a rusher ahead by 5ϵ

6 Discussion

In this section, we discuss application of HLC for finding consistent snapshots in distributed databases, compact representations of l and c, and other related work.

6.1 Snapshots

In snapshot read, the client is interested in obtaining a snapshot of the data at a given time. HLC can be used to perform snapshot read similar to that performed by TrueTime. Moreover, unlike TT, there is no need to delay any transaction due to uncertainty in the clock values.

To describe our approach more simply, we introduce the concept of virtual dummy events. Let e and f be two events on the same node such that $l.e < l.f$. In this case, we introduce dummy (internal) events whose l value is in the range $[l.e + 1, l.f]$ and $c.f = 0$. (If $c.f = 0$ then the last event in the sequence is not necessary.) Observe that introducing such dummy events does not change timestamps of any other events in the system. However, this change ensures that for any time t, there exists an event on every node where l value equals t and c value equals 0.

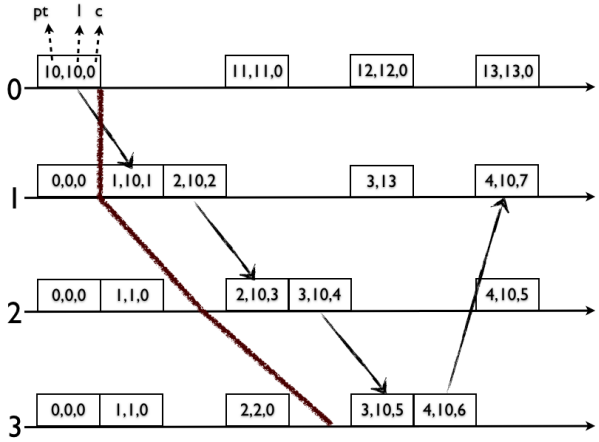

Figure 12: Consistent snapshot for $t = 10$ in HLC trace

With the virtual dummy events adjustment, given a request for snapshot read at time t, we can obtain the values at timestamp $\langle l = t, c = 0\rangle$.[4] Our adjustment ensures that such events are guaranteed to exist. And, by the logical clock hb relationship mentioned in requirement 2, we have $hlc.e = hlc.f \Rightarrow e||f$ and so we can conclude that the snapshots taken at this time are consistent with each other and form a consistent global snapshot. Moreover, based on Theorem 3 and Corollary 2, this snapshot corresponds to the case where the global time is in the window $[t - \epsilon, t]$.

In Figure 12, we show an example of finding consistent snapshot given a request for snapshot read at time $t = 10$. As per our algorithm, we read the state at each node at timestamp $\langle l = 10, c = 0\rangle$, and this corresponds to the snapshot illustrated in Figure 12.

[4] Actually we can obtain snapshot reads for any $\langle l = t, c = K\rangle$ and not just at $\langle l = t, c = 0\rangle$

6.2 Compact Timestamping using l and c

NTP uses 64-bit timestamps which consist of a 32-bit part for seconds and a 32-bit part for fractional second. (This gives a time scale that rolls over every 2^{32} seconds—136 years— and a theoretical resolution of 2^{-32} seconds—233 picoseconds.) Using a single 64-bit timestamp to represent HLC is also very desirable for backwards compatibility with NTP clocks. Being backwards compatible with NTP clocks is important because many distributed database systems and distributed key-value stores use NTP clocks to timestamp and compare records.

There are, however, several challenges for representing HLC as a single 64-bit timestamp. Firstly, the HLC algorithm maintains l and c separately, to differentiate between increases due to the physical clock versus send/receive/local events. Secondly, by tracking the pt, the size of l is by default 64-bits as the NTP timestamps.

We propose the following scheme for combining l and c and storing it in single 64 bit timestamp. This scheme involves restricting l to track only the most significant 48 bits of pt in the HLC algorithm presented in Figure 5. Rounding up pt values to 48 bits l values still gives us microsecond granularity tracking of pt. Given NTP synchronization levels, this is sufficient granularity to represent NTP time. The way we round up pt is to always take the ceiling to the 48th bit. In the HLC algorithm in Figure 5, l is updated similarly but is done for 48 bits. When the l values remain unchanged in an event, we capture that by incrementing c following the HLC algorithm in Figure 5. 16 bits remain for c and allows it room to grow up to 65536, which is more than enough as we show in our experiments in Section 5.

Using this compact representation, if we need to timestamp (message or data item for database storage), we will concatenate c to l to create the HLC timestamp. The distributed consistent snapshot finding algorithm described above is unaffected by this change to the compact representation. The only adjustment to be made is to round up the query time t to 48 bits as well.

6.3 Other related work

Dynamo [23] adopts VC as version vectors for causality tracking of updates to the replicas. Cassandra uses PT and LWW-rule for updating replicas.

Spanner [2] employs TT to order distributed transactions at global scale, and facilitate read snapshots across the distributed database. In order to ensure e hb $f \Rightarrow tt.e < tt.f$ and provide consistent snapshots, Spanner requires waiting-out uncertainty intervals of TT at the transaction commit time which restricts throughput on writes. However, these "commit-waits" also enable Spanner to provide a stronger property, external consistency (a.k.a, strict serializability): if a transaction t1 commits (in absolute time) before another transaction t2 starts, then t1's assigned commit timestamp is smaller than t2's.

HLC does not require waiting out the clock uncertainty, since it is able to record causality relations within this uncertainty interval using the HLC update rules. HLC can also be adopted for providing external consistency and still keeping the throughput on writes unrestricted by introducing client-notification-wait after a transaction ends.

An alternate approach for ordering events is to establish explicit relation between events. This approach is exemplified in the Kronos system [5], where each event of interest is registered with the Kronos service, and the application explicitly identifies events that are of interest from causality perspective. This allows one to capture causality that is application-dependent at the increased cost of searching the event dependency relation graph. By contrast, LC/VC/PT/HLC assume that if a node performs two consecutive events then the second event causally depends upon the first one. Thus, the ordering is based solely on the timestamps assigned to the events.

7 Conclusion

In this paper, we introduced the hybrid logical clocks (HLC) that combines the benefits of logical clocks (LC) and physical time (PT) while overcoming their shortcomings. HLC guarantees that (one way) causal information is captured, and hence, it can be used in place of LC. Since HLC provides nodes a logical time that is within possible clock drift of PT, HLC is substitutable for PT in any application that requires it. HLC is strictly monotonic and, hence, can be used in place of applications in order to tolerate NTP kinks such as non-monotonic updates.

HLC can be implemented using 64 bits space, and is backwards compatible with NTP clocks. Moreover, HLC only reads NTP clock values but does not change it. Hence, applications using HLC do not affect other applications that only rely on NTP.

HLC is highly resilient. Since its space requirement is bounded by theoretical analysis and is shown to be even more tightly bounded by our experiments, we use this as a foundation to design stabilizing fault tolerance to HLC.

Since HLC refines LC, HLC can be used to obtain a consistent snapshot for a snapshot read. Moreover, since

the drift between HLC and physical clock is less than the clock drift, a snapshot taken with HLC is an acceptable choice for a snapshot at a given physical time. Thus, HLC is especially useful as a timestamping mechanism in multiversion distributed databases. For example in Spanner, HLC can be used in place of TrueTime (TT) to overcome one of the drawbacks of TT that requires events to be delayed/blocked in the *clock synchronization uncertainty window*. HLC allows the application events to be produced at the rate desired by the application.

References

[1] K. Bhatia, K. Marzullo, and L. Alvisi. Scalable causal message logging for wide-area environments. *Concurrency and Computation: Practice and Experience*, 15(10):873–889, 2003.

[2] J. Corbett, J. Dean, et al. Spanner: Google's globally-distributed database. *Proceedings of OSDI*, 2012.

[3] M. Demirbas and S. Kulkarni. Beyond truetime: Using augmentedtime for improving google spanner. *LADIS '13: 7th Workshop on Large-Scale Distributed Systems and Middleware*, 2013.

[4] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.

[5] R. Escriva, A. Dubey, B. Wong, and E.G. Sirer. Kronos: The design and implementation of an event ordering service. *EuroSys*, 2014.

[6] R. Fan and N. Lynch. Gradient clock synchronization. In *PODC*, pages 320–327, 2004.

[7] J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, Feb 1988.

[8] K. Kingsbury. The trouble with timestamps. `http://aphyr.com/posts/299-the-trouble-with-timestamps`.

[9] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. *SIGOPS Oper. Syst. Rev.*, 41(6):45–58, October 2007.

[10] S. Kulkarni and Ravikant. Stabilizing causal deterministic merge. *J. High Speed Networks*, 14(2):155–183, 2005.

[11] Avinash Lakshman and Prashant Malik. Cassandra: Structured storage system on a p2p network. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, pages 5–5, 2009.

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

[13] The future of leap seconds. `http://www.ucolick.org/~sla/leapsecs/onlinebib.html`.

[14] Another round of leapocalypse. `http://www.itworld.com/security/288302/another-round-leapocalypse`.

[15] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguica, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. *OSDI*, 2012.

[16] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *SOSP*, pages 401–416, 2011.

[17] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi. The flooding time synchronization protocol. *SenSys*, 2004.

[18] A. Mashtizadeh, A. Bittau, Y. Huang, and D. Mazières. Replication, history, and grafting in the ori file system. In *SOSP*, pages 151–166, 2013.

[19] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, pages 215–226, 1989.

[20] D. Mills. A brief history of ntp time: Memoirs of an internet timekeeper. *ACM SIGCOMM Computer Communication Review*, 33(2):9–21, 2003.

[21] B. Sigelman, L. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[22] Y. Sovran, R. Power, M. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, pages 385–400, 2011.

[23] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[24] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *SOSP*, pages 292–308, 2013.

[25] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *SOSP*, pages 276–291, 2013.