Highly Auditable Distributed Systems

Murat Demirbas
University at Buffalo, SUNY

Sandeep Kulkarni
Michigan State University

Abstract

Auditability is a key requirement for providing scalability and availability to distributed systems. Auditability allows us to identify latent concurrency bugs, dependencies among events, and performance bottlenecks. Our work focuses on providing auditability by combining two key concepts: time and causality. In particular, we prescribe hybrid logical clocks (HLC) which offer the functionality of logical clocks while keeping them *close* to physical clocks. We propose that HLC can enable effective detection of invariant predicate violations and latent concurrency bugs, and provide efficient means to correct the state of the distributed system back to good states.

1 Introduction

For developing high availability, high reliability, and high performance distributed systems, it is important to improve auditability, which enables identifying performance bottlenecks, dependencies among events, and latent concurrency bugs. High auditability also helps for ease-of-programming and ease-of-extensibility, as well as security and accountability.

Current software systems still rate poorly on the auditability scale. Logging system messages, assertions, and exceptions are common approaches to providing some auditability. While those approaches are fine when used on a single computer system, concurrency makes reasoning about distributed systems tricky, and renders those naive logging-based approaches insufficient for auditability of distributed systems.

Two critical concepts for auditability of a system are time and causality. However, these concepts have traditionally been considered individually (rather than together), and there is a gap between how these concepts are utilized in theory and practice. The theory of distributed systems shunned the notion of time and considered asynchronous systems, whose event ordering is captured by logical clocks [17]. The practical distributed systems employed NTP synchronized clocks to capture time but did so in ad hoc undisciplined ways, thereby allowing the possibility that one event can causally affect another event even though both of them appear to have occurred at the same (local physical) time.

While there have been many ad hoc point solutions to employing NTP clocks [19] to improve performance in distributed systems [1, 8, 18], there has been no effort to address the underlying root research problem of devising a principled general theory of synchrony-aware clock components and integrated system primitives to achieve high auditability for large scale distributed systems with widely varying link/communication characteristics. By leveraging hybrid logical clocks (HLC) as a basic building block, we aim to enable the design of highly auditable distributed systems.

Hybrid logical clock (HLC) [16] combines the theoretical underpinnings of causality and the practicality of physical clocks by identifying how logical clocks can be improved and tuned based on the availability of NTP synchronization. The principle guiding HLC design is "uncertainty resilience". HLC is designed to be always wait-free/nonblocking and correct (albeit with reduced performance/efficiency) even when time synchronization has degraded or is not available. In other words, HLC provides survivability and resilience to time-synchronization errors and can make progress, and capture causality information even when time synchronization has degraded or is not available. HLC enables highly auditable systems since HLC can efficiently provide global consistent-state snapshots without needing to wait out clock synchronization uncertainties and without requiring prior coordination.

In order to enable highly available distributed systems, we leverage the high auditability support provided by HLC to design fault-tolerance components that detect and correct distributed system state corruptions, based on the *centralized oversight/override principle*. In distributed systems, many problems are hard to solve with local information, but become significantly easy if you can peek at global system state. Our proposed synchrony-aware detector/corrector components enable the audit of distributed system state in the background and upon detecting a fault will interfere to instate the system state to an appropriate legitimate state.

2 Efficient distributed snapshots with HLC

To implement *HLC* [16], each node j maintains timestamp of the form $\langle pt.j, l.j, c.j \rangle$, where $pt.j$ corresponds to the physical clock of process j, $l.j$ denotes the maximum physical clock value that j is aware of. The value of $c.j$ captures the length of the maximum causal chain that is of the form $(e_1, e_2, .., e_k)$ such that $e_1 \ \underline{hb} \ e_2, e_2 \ \underline{hb} \ e_3, \cdots, e_{k-1} \ \underline{hb} \ e_k$ and e_k is the latest event on process j. [1] Given a system with a maximum clock drift of ε, *HLC* guarantees that $l.j$ is in the range $[pt.j, pt.j+\varepsilon]$. Furthermore, the value of $c.j$ is bounded, because $c.j$ is reset to 0 when $l.j$ increases (which inevitably happens in the worst case when $pt.j$ exceeds $l.j$). Theoretically, the bound on $c.j$ is proportional to the number of processes and ε. Practically, we find that $c.j$ is always less than 10 even under different demanding experiment settings we deployed over AWS [3]. *HLC* provides causality information similar to that by *LC*. In particular, if $e \ \underline{hb} \ f$ then $HLC.e < HLC.f$, where $HLC.e < HLC.f$ iff $l.e < l.f \vee (l.e = l.f \wedge c.e < c.f)$. [2] Unlike *LC*, however, *HLC* provides logical timestamps that are *close* to physical timestamps. In fact since the maximum drift between l and pt value is ε and there is an uncertainty of ε in pt values due to clock drift. Thus, being a scalar, HLC timestamps are backwards compatible with NTP timestamps used in legacy applications.

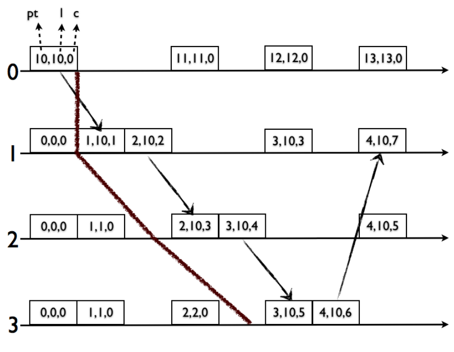

Figure 1: Finding a consistent snapshot for $t = 10$ using HLC. The example considers $\varepsilon = 10$.

The key advantage of *HLC* is its resilience to synchronization uncertainty. Specifically, it continues to satisfy the constraint $e \ \underline{hb} \ f \Rightarrow HLC.e < HLC.f$ even if clock drift increases substantially and the processes are not aware of the drift. The only effect of such a drift is to increase the drift between l and pt values and higher c values. Hence, *HLC* can easily tolerate violation of drift requirement. *HLC* is also self-stabilizing [12], i.e., if clock values are perturbed or corrupted, they are restored quickly so that causality detection will be correct in the future.

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 %	

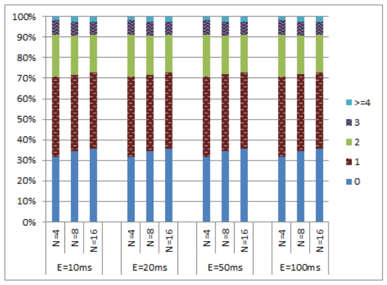

Figure 2: Results from preliminary experiments

We have conducted experiments with HLC on Amazon AWS [3] xlarge instances as well as simulations that can stress the HLC by sending a large number of messages that are received *very quickly*. In all these instances, the difference between the value of the logical clock and the physical clock was bounded by the actual clock drift. This ensures that logical clocks are substitutable for physical clocks for virtually all applications. Furthermore, the value of c, the extra storage required HLC is very small. In our experiments with Amazon xlarge instances, the maximum value of c was less than 10 (Figure 2(a) provides a brief summary of these experiments. As shown in these experiments, with $\varepsilon = 16ms$, 66.96% of events had c value of 0, 19.40% of events had c value of 1 and so on). With simulated clock drift where some node intentionally violates clock synchronization requirements, it was possible to have events where c value was in $100s$. However, the number of such events is very small (less than 0.01%). As shown in Figure 2(b), for 98% of events, the c value was less than 4.

Efficient snapshot taking using HLC. The distributed snapshot problem is to return a globally consistent-state of the distributed system by identifying pairwise concurrent local snapshots at every node. HLC simplifies the design of snapshot primitive because a collection of local snapshots taken at identical logical clock values are guaranteed to be a consistent cut. (Note that since physical clocks are not perfectly synchronized it is not possible to get a consistent snapshot by just reading state at different nodes at physical clock time t, but instead uncertainty windows need to be waited out [13].)

Our snapshot primitive differs from previous ap-

[1] $\underline{hb}$ denotes the happened-before relationship [17].

[2] By taking the snapshot of every node at *logical time t*, the combined snapshot is guaranteed to be consistent, because from this requirement we have $l.e = l.f \Rightarrow e || f$.

proaches [5,15,20] because in those approaches all snapshots are pre-planned (i.e., one can take a snapshot of the system *after* the current point of execution), and require substantial coordination among nodes. By contrast, our synchrony-aware snapshot primitive enables one to obtain a snapshot in the past by simply inputting the physical time at which the snapshot is desired to be taken. By reducing the overhead of obtaining snapshots, HLC improves adoption of snapshots for solving distributed problems. The snapshot primitive serves as the building block for providing detectors and correctors we propose in subsequent sections. It also enables highly auditable systems including synchrony-aware services for debugging, assertion monitoring, automating unobtrusive backups, and checkpointing-and-recovery.

Comparison to TrueTime and Spanner. Spanner [7] is Google's scalable, multiversion, globally-distributed database. Spanner evolved from a Bigtable-like versioned key-value store into a temporal multiversion database with SQL-based query support. To support distributed transactions at global scale, Spanner leverages on TrueTime (TT) API that exposes clock uncertainty.

We consider two main issues with Spanner's TT-based approach. Spanner's TT-based approach discards the tracking of causality information completely. Instead it goes for an engineering solution of using highly-precise dedicated clock sources to reduce the size of the uncertainty windows to be negligible and order events using wallclock time. When the uncertainty windows are overlapping TT cannot order events, and that is why in order to ensure external consistency it has to explicitly waitout these uncertainty windows upto 7ms. Secondly, this approach also requires access to GPS and atomic clocks and customized time synchronization protocols to maintain very tightly synchronized time at each spanserver.

A major advantage of the HLC-based implementation over TT is that HLC is wait-free and allows higher throughput. Since TT requires waiting-out uncertainty windows for the transaction commit, ε determines the throughput of read-write transactions on a tablet level. The HLC-based implementation, on the other hand, does not require waiting ε out, instead it records finer-grain causality relations within this uncertainty window. HLC also obviates the need for dedicated GPS or atomic clock references, and can work with NTP servers that provide an ε of several tens of milliseconds. (For a more detailed comparison, including achieving external consistency, please see [16].)

Due to these properties, our HLC clocks have recently been adopted by CockroachDB [6], an opensource clone of Google Spanner. An implementation of HLC in Go is available at `https://github.com/cockroachdb/cockroach/tree/master/util/hlc`

3 Detection

Debugging a distributed system is notoriously difficult task, because of the concurrency issues involved. Concurrent execution yields intricate and unanticipated race conditions. While providing complete assurance about the given distributed program is challenging, the developer can often identify assertions that can be checked to determine whether the program is still (likely to be) working correctly. Unfortunately, even enforcing such assertions have been difficult due to the distributed nature of the system [21].

A simple strawman approach for assertion monitoring is to represent the distributed system state as a multiversion database. In this approach, each update of a variable (i.e., version) is recorded with its associated timestamp on that node. HLC supports efficient and consistent querying of such a distributed multiversion database maintained over the nodes of the distributed system. Recording every update of each variable can be, of course, expensive. The tradeoffs between the overhead of monitoring and the types of assertions to be monitored should be investigated. For assertion predicates that are stable or that remain valid for a sufficiently long duration, an occasional periodic snapshot is enough. However, for transient predicates such infrequent snapshots are not effective. To allow both, we propose the two strategies below.

The first strategy is recency-sensitive snapshots. This approach aims to provide a fine-grained support for more recent state updates and a rough-grained support for older state. The idea is to use a sliding window within which ephemeral multiversion support is provided for critical variables. Past the sliding window, snapshots are retained only with rough granularity. The logic behind this is that more recent state is more relevant for fault-tolerance purposes than older state.

The second strategy is to take predicate triggered snapshots. The developer can provide some local predicates that can act as triggers for storing of a local snapshot at a node. The node then forwards this trigger state to the detector and the detector performs an on-demand query for that state with that timestamp. If the system employs recency-sensitive snapshots, then multiversion values would still be in the RAM of the other nodes, and the detector's query would be answered. Or else, the detector's query would trigger taking/storing a snapshot of the system.

To complement the above two strategies, it is also beneficial to investigate selective global state monitoring as in Google's Dapper service. Google Dapper [22] is a distributed system tracer/logger/profiler service, which performs sparse logging and fails to support global predicate detection. The use cases for Dapper focus on inferring service dependencies and identification of performance bottlenecks (long-tail latencies) and detecting simple correctness problems. In addition, work on detecting global predicates in distributed systems [14] has identified conjunctive predicates as amenable to efficient detection. Specifically, if the predicates are stable then this approach would detect it in the time proportional to the frequency of invoking the detector. If the predicate is conjunctive in nature then detection can be invoked only if the local predicate at some node evaluates to true, thereby reducing unnecessary invocations of the detector.

Multiple detectors. Our approach can also easily support multiple detectors since detectors are read-only. They can be used for *hot detection* where the users want to check whether the current system state (or a very recent one) satisfies the desired detection predicate. Our HLC provides the necessary information for constructing consistent global states for this detection. The detectors can be also used for *cold detection*, where the detectors are used to analyze past states of the system, this can be achieved by using a multiversion database (such as Spanner [7] or CockroachDB [6]) to store and query past states. Thus, one of the open issues in this context is identifying the frequency with which detector should be invoked so that users can perform the desired queries. Another open question in this context is whether the multiple detectors can be aggregated to reduce their overhead, especially for hot detection.

4 Correction

For correction we propose to employ distributed reset using HLC. Distributed reset [2] provides a clean-slate solution to the correction problem. The goal of distributed reset is to restore the system to a pre-determined state (e.g., initial state, previous consistent state etc). Since all processes cannot be reset simultaneously and we want to avoid blocking of all processes to complete the reset, these protocols also identify when two processes can safely communicate so that the uncorrected processes do not re-corrupt the corrected processes.

With HLC, the problem of distributed reset can be simplified. In particular, if a reset is requested at HLC time t, the initiator can choose a HLC time $t' > t$ and require all nodes to reset when their HLC clock reaches t'. Since HLC refines logical clocks and respect the causality relation, this ensures that the individual resets of all nodes are (causally) concurrent even if they do not occur at the exact same instant. Moreover, given that HLC is close to physical clocks, the developer can bound the delay in performing the reset.

The HLC-based distributed reset would scale time-wise for very large scale distributed systems even over wide-area networks (WANs) because it has constant time cost that depends upon the delay involved in notifying all processes to perform the reset. However, due to some stragglers that do not meet NTP clock synchronization or temporary partitioning of the network or network congestion, problems may occur. And, if some nodes fail to reset, it may be necessary to perform the reset again. Also, resetting the entire system can be unacceptable for very large distributed systems. Moreover, in a very large scale distributed system since faults are more likely to occur the number/frequency of distributed resets becomes an issue. As a result, the availability of correct nodes may suffer due to problems in some nodes in the system.

Resetting a subset of nodes. Resetting only the nodes that are problematic/contaminated is a better solution, but it is challenging to ensure a globally-consistent state after a partial/subset reset of the distributed system. We use the following approach to deal with this problem.

When the corrector notices a problem, say at T1, it determines a subset/region of nodes to correct to fix the problem. Then it freezes these nodes at time T2. The freeze is in effect until the reset, so the frozen nodes don't change their states or communicate with any nodes. The other (unfrozen) nodes are free to continue execution, but of course, they will not receive an answer back from the frozen nodes for a read or write request. It is as if those nodes are just unavailable or are late to reply. (This is done to isolate the nodes to be reset, so that their states should not corrupt the other nodes. In other words, we are curbing the spread of corruption.)

The corrector takes a snapshot also at time T2. When the snapshot is available at the corrector, it determines the exact state to put the frozen nodes at T2 so that their states become consistent with the rest of the system at T2. In other words, the corrector ensures that the entire system has a consistent global state at T2. Of course by the time the frozen nodes are reset, the rest of the nodes are at T3. But this is acceptable since the resulting state is still a globally consistent state. It is as if the frozen nodes lagged a little in execution and their states and time is still at T2 (but this time at a consistent state with the rest of the system at T2). It is safe to assume this, since the frozen nodes do not communicate with others between time T2 to T3.

After the reset, the nodes that were reset will catchup normally with the rest of the system in terms of time. HLC is uncertainty resilient and will not lead to unsafe comparisons and updates if the clock synchronization lagged (in this case, artificially lagged due to reset of those nodes). This approach reduces the number of nodes that need to be reset at the cost of a more complex protocol and slightly increased latency for reset.

Multiple correctors. Since the detection is read only, extending the detection scheme to multiple detectors was straightforward. For extending to multiple correctors, however, we need to provide coordination between correctors so that they do not interfere with each other.

If we design the correctors to operate on orthogonal (i.e., independently resettable) set of nodes, this problem is solved easily. In that case, part of the state of a node may be frozen/reset at T2 and the other part of state keep getting updated.

When correctors operate on same state space, we need to investigate new approaches. A hierarchical corrector setup is possible with a fish-eye like consistency, gradient-state approach. The rest of the system may provide a rough granularity state at T2 to reset this site to be in consistent with. The insight here is that it is easy to achieve consistency with respect to fewer/rougher state from the rest of the system.

Another approach for allowing multiple correctors is to enable the correctors to communicate to figure out how to achieve eventual consistency by considering cumulative joints of the states of the nodes they are responsible for.

Note that, even with multiple correctors, concurrent overlapping resets is not a problem since they are prevented by our distributed reset mechanism. Consider multiple resets such as a subset reset at T2, a subset reset at T3, and a subset reset at T4. In order to have a reset at T3, the nodes being reset at T2 need to reach T3 first. So, while T2 is being reset, we cannot have a subset reset at T3. In other words, such concurrent resets can increase the time for completion but cannot cause unanticipated behavior.

5 Related work and extensions

Crash-only software. The crash-only software [4] proposes that computer programs should handle failures by simply restarting, without attempting any sophisticated recovery. To make components crash-only, the approach requires that all important non-volatile state be kept in dedicated crash-only state stores (e.g., databases), leaving applications with just soft-state program logic.

The crash-only software approach is more suitable for loosely coupled components, since components should be able to tolerate not receiving a response from a component that is resetting/crash-recovering. The crash-only software approach deals with component-nonlocal failures by iteratively performing resets on a larger subsystem. If a previous reset of a smaller subsystem fails, the approach includes more components to reset the next time, hoping that a larger subsystem reset (and in the worst case the entire system reset) will fix the correctness problem.

Eidetic distributed systems. In [11], authors have introduced the notion of eidetic systems, i.e., systems that can recall any past state that existed on that computer, including all versions of all files, the memory and register state of processes, interprocess communication, and network input. An eidetic computer system can explain the lineage of each byte of current and past state.

The results in [11] are designed for single processor machines and do not account for issues in distributed systems. HLC provides two critical properties in the context of extending this work to distributed systems: HLC ensures that any causal effect (where one node is dependent upon the state of another node) is captured correctly, and HLC ensures that clocks of different nodes are *close* to each other. A key difficulty with this extension is that the eidetic system already suffers a significant increase in the cost of querying. Also, although the storage cost of the eidetic system on a single node is relatively low (1TB/year), it may increase substantially with a distributed system (with only a handful nodes) due to the fact that there is substantially more non-determinism in these systems.

Hot & Cold detection, and Root-cause analysis. To perform hot detection, only very recent (i.e., hot) state is needed. This state can be cached on the nodes ad hocly to be forwarded to the detector(s), and does not need to be stored. To perform cold detection, older state from nodes also need to be queried. The cold state can be stored by employing a multiversion database (such as Spanner [7] or CockroachDB [6]).

Although hot and cold detection is enough for the purposes of any global-state predicate detection and correction, a more detailed root-cause analysis is needed in order to assign blame and identify the process/user/node/component that is responsible for the violation. To this end, we can adopt a storage system as implemented in the eidetic systems paper [11]. To determine root-cause, the eidetic system runs a backward query, which proceeds in a tree-like search fanning out from one or more target states. After tracking down the root-cause for the violation, an eidetic system can perform a forward query, to figure out which data are contaminated with this faulty information and fix them. This allows for a fine-grained correction with a chisel rather than the rough-grained correction with the reset hammer. On the other hand, forward queries are involved and already slow (around 100sec) even on a single system. Therefore they would be more appropriate for forward-correction of small critical data/configuration corruptions.

References

[1] ADYA, A., AND LISKOV, B. Lazy consistency using loosely synchronized clocks. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, Santa Barbara, California, USA, August 21-24, 1997* (1997), J. E. Burns and H. Attiya, Eds., ACM, pp. 73–82.

[2] ARORA, A., AND GOUDA, M. G. Distributed reset. *IEEE Trans. Computers 43*, 9 (1994), 1026–1038.

[3] `http://aws.amazon.com`.

[4] CANDEA, G., AND FOX, A. Crash-only software. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems* (2003), HOTOS.

[5] CHANDY, K. M., AND LAMPORT, L. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions Computing Systems 63-75* (1985).

[6] Cockroachdb: A scalable, transactional, geo-replicated data store. http://cockroachdb.org/.

[7] CORBETT, J., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. *Proceedings of OSDI* (2012).

[8] COWLING, J., AND LISKOV, B. Granola: low-overhead distributed transaction coordination. In *USENIX ATC* (2012), pp. 21–21.

[9] DEMIRBAS, M. Distributed is not necessarily more scalable than centralized. `http://muratbuffalo.blogspot.com/2014/07/distributed-is-not-necessarily-more.html`, July 2014.

[10] DEMIRBAS, M., AND KULKARNI, S. Beyond truetime: Using augmentedtime for improving google spanner. *LADIS '13: 7th Workshop on Large-Scale Distributed Systems and Middleware* (2013).

[11] DEVECSERY, D., CHOW, M., DOU, X., FLINN, J., AND CHEN, P. Eidetic systems. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation* (2014), pp. 525–540.

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

[13] DU, J., ELNIKETY, S., AND ZWAENEPOEL, W. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on* (2013), IEEE, pp. 173–184.

[14] GARG, V. K., AND CHASE, C. Distributed algorithms for detecting conjunctive predicates. *International Conference on Distributed Computing Systems* (June 1995), 423–430.

[15] KSHEMKALYANI, A., AND SINGHAL, M. Efficient distributed snapshots in an anonymous asynchronous message-passing system. *Journal of Parallel and Distributed Computing 73*, 5 (2013), 621–629.

[16] KULKARNI, S., DEMIRBAS, M., MADAPPA, D., AVVA, B., AND LEONE, M. Logical physical clocks. In *Principles of Distributed Systems* (2014), Springer, pp. 17–32.

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

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

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

[20] RAYNAL, M. *Distributed algorithms for message-passing systems*. Springer, 2013.

[21] SHEEHY, J. There is no now. *ACM Queue 13*, 3 (2015), 20.

[22] SIGELMAN, B., BARROSO, L., BURROWS, M., STEPHENSON, P., PLAKAL, M., BEAVER, D., JASPAN, S., AND SHANBHAG, C. Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Google, Inc., 2010.