Retrospective Lightweight Distributed Snapshots Using Loosely Synchronized Clocks

Aleksey Charapko, Ailidani Ailijiang, Murat Demirbas
University at Buffalo, SUNY
Email: {acharapk,ailidani,demirbas}@buffalo.edu

Sandeep Kulkarni
Michigan State University
Email: sandeep@cse.msu.edu

***Abstract*—In order to take a consistent snapshot of a distributed system, it is necessary to collate and align local logs from each node to construct a pairwise concurrent cut. By leveraging NTP synchronized clocks, and augmenting them with logical clock causality information, Retroscope provides a lightweight solution for taking unplanned retrospective snapshots of past distributed system states. Instead of storing a multiversion copy of the entire system data, this is achieved efficiently by maintaining a configurable-size sliding window-log at each node to capture recent operations. In addition to retrospective snapshots, Retroscope also provides incremental and rolling snapshots that utilize an existing snapshot to reduce the cost of constructing a new snapshot in proximity. This capability is useful for performing stepwise debugging and root-cause analysis, and supporting data-integrity monitoring and checkpoint-recovery. We implement Retroscope for the Voldemort distributed datastore and evaluate its performance under varying workloads.**

I. Introduction

Logging system state, messages, and assertions is a common approach to providing auditability in a single computer system. However, naive logging-based approaches fail for the auditability of distributed systems. For distributed systems, it is necessary to collate and align local logs from each node into a *globally consistent snapshot* [1]. This is important, because inconsistent snapshots are useless and even dangerous as they give misinformation.

Unfortunately, current distributed snapshot algorithms are expensive and have shortcomings. The Chandy-Lamport snapshot algorithm [2] assumes FIFO channels and takes a proactive approach. It allows only scheduled, planned snapshots, and as such it is not amenable for taking a *retrospective snapshot* of a past state. One way to achieve retrospective snapshots is via the use of vector clocks (VCs) [3]–[5] with space complexity of $\Theta(n)$ to be included in each message in the system. This incurs intolerable overhead that grows linearly with n, the number of nodes. Moreover, VCs do not capture physical time affinity and using VCs in partially synchronized systems implies that potentially unreachable states may be reported as false positives. Logical clocks (LCs) [6] can be considered for reducing the cost of VC. However, taking a retrospective snapshot with LCs also fails because, unlike VCs, LCs capture causality partially, and cannot identify consistent snapshots with sufficient affinity to a given physical time.

To get snapshots with sufficient affinity to physical time, one can potentially utilize NTP [7]. However, since NTP 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. A globally consistent snapshot comprises of pairwise concurrent local snapshots from the nodes, but the local snapshots at T may have causal precedence, invalidating the resultant global snapshot (cf. Figure 1). Thus, using NTP to obtain a consistent cut requires waiting out the clock uncertainty [8], [9], making it unsuitable for consistent snapshots.

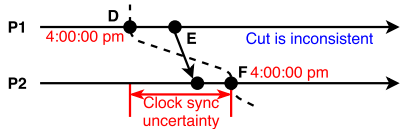


Fig. 1: Using NTP only fails to take consistent shot

Retroscope. To address this problem, we leverage our recent work on hybrid logical clocks (HLC) [10]. HLC is a hybrid of LC and NTP, and combines causality with physical clocks to derive scalar HLC timestamps. HLC facilitates distributed snapshots because a collection of local snapshots taken at identical HLC timestamps are guaranteed to be a consistent cut. Using this observation, we design and develop *Retroscope*, a lightweight solution for constructing consistent distributed snapshots by collating node-level independent snapshots.

Retroscope supports taking unplanned retrospective snapshots of a past system state in an efficient manner. Instead of storing a multiversion copy of the entire system data, this is achieved efficiently by maintaining a configurable-size sliding window-log at each node to capture recent operations. In addition to instant and retrospective snapshots, Retroscope also provides incremental and rolling snapshots that utilize an existing full snapshot to compensate the cost of snapshot exploration in proximity. After a retrospective snapshot is taken at a recent past time T, the cost of taking snapshots at $T + k$, for small values of k, becomes negligible. Using incremental and rolling snapshots, Retroscope supports performing stepwise debugging, root-cause analysis, data-integrity monitoring, and checkpoint-recovery. A *devops* team can use Retroscope to explore a problem by stepping through a time interval of interest. Retroscope can also help identify a clean snapshot, where data integrity constraints hold, in order to recover the system with minimal loss of updates.

We design and develop Retroscope as a standalone library so it can be easily added to existing distributed systems. Our Retroscope implementation is available on github as an opensource project [11]. We have used Voldemort key-value store to showcase Retroscope and evaluate its performance.

Retroscoping Voldemort. Voldemort [12] is a popular opensource system used in LinkedIn, that implements a Dynamo-like highly available distributed key-value store [13]. Our Retroscope instrumentation of Voldemort leverages the Retroscope library functionality and required less than 1000

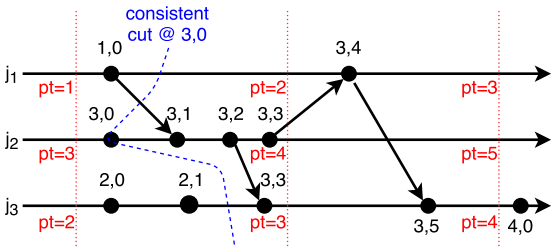


Fig. 2: Example of HLC operation with $\epsilon = 2$ on 3 process. Dashed lines denote the physical clock ticks with timestamp for each process next to it. HLC time is written above each event in the "l,c" format.

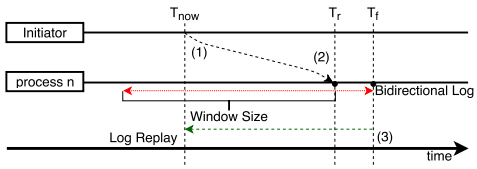


Fig. 3: Instant distributed snapshot.

lines of code to be added to the data-store. Retroscoped Voldemort maintains a sliding window log for capturing recent events, and enables any client to initiate a snapshot for time T within this window-log. When a snapshot is requested, this window-log and the database state is used for constructing the snapshot for the requested time. For a 2GB Voldemort database maintained over a 10 nodes cluster, taking and finalizing a snapshot for current time, T_{now}, requires ∼15 seconds. After a snapshot is taken, it takes only ∼100msecs for taking an incremental snapshot in the vicinity of that snapshot. The snapshots can go back to ∼10 minutes in the past. Going further in the past increases the window-log size, increasing the snapshot completion time. Certain optimizations, such as periodic window-log compaction, deferred snapshots, and speculative snapshots help improve the performance.

Outline of the rest of the paper. We describe HLC timestamping next. In Section III, we explain the Retroscope mechanisms and evaluate the performance in a Voldemort case study in Section IV. We discuss extensions in Section V. We review related work before our concluding remarks.

II. HLC Timestamping

Logical clocks (LCs) satisfy the logical clock condition: if e $\underline{hb}$ f then LC.e < LC.f, where $\underline{hb}$ is the happened-before relation defined by Lamport [6].This condition implies that if we pick a snapshot where for all e and f on different nodes LC.e = LC.f, then we have $\neg(e \ \underline{hb} \ f)$ and $\neg(f \ \underline{hb} \ e)$, and therefore the snapshot is consistent.[1]However, since LC timestamps are driven by occurrences of events, and the nodes have different rate of event occurrences, it is unlikely to find events at each node with the same LC values where all are within a given physical clock affinity.

Since HLC [10] is a hybrid of NTP and LC, it satisfies the logical clock condition: if e $\underline{hb}$ f then HLC.e < HLC.f. Thus, a snapshot where, for all e and f on different nodes, HLC.e = HLC.f is a consistent snapshot as shown in Figure 2. Moreover, since HLC logical time is driven by NTP time, it is easy to find events at each node with the same HLC values that are all within sufficient affinity of the given physical time.

HLC implementation. Figure 2 illustrates HLC operation. At any node j, HLC consists of $l.j$ and $c.j$. The term $l.j$ denotes the maximum physical clock value, p, that j is aware of. This maximum known physical clock value may come from the physical clock at j, denoted as $pt.j$, or may come from another node k via a message reception that includes $l.k$. Thus given that NTP maintains the physical clocks at nodes within a clock skew of at most ϵ, $l.j$ is guaranteed to be in the range $[pt.j, pt.j + \epsilon]$. The second part of HLC, $c.j$, acts like an overflow buffer for $l.j$. When a new local or receive event occurs at j, if $l.j$ stays the same[2], then in order to ensure the logical clock condition $c.j$ is incremented, as HLC.e < HLC.f is defined to be $l.e < l.f \vee (l.e = l.f \wedge c.e < c.f)$. On the other hand, $c.j$ is reset to 0 when $l.j$ increases (which inevitably happens in the worst case when $pt.j$ exceeds $l.j$). The value of $c.j$ is bounded. In theory, the bound on $c.j$ is proportional to the number of processes and ϵ, and in practice $c.j$ was found to be a small number (< 10) under evaluations.

Our HLC implementation in Java is based on the HLC implementation of CockroachDB [14] in Go. HLC can fit $l.j$ and $c.j$ in 64 bits in a manner backwards compatible with the NTP clock format [7] and can easily substitute for NTP timestamps used in many distributed systems. HLC is also resilient to synchronization uncertainty: degraded NTP synchronization only increases the drift between l and pt values and introduces larger c values.

III. Retroscope Snapshots

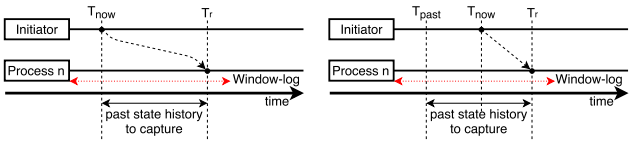
Retroscope keeps a local log at each node to record the recent state changes. This log is maintained as a sliding window, and each state change in the window-log is accompanied by an HLC timestamp. By ensuring that all nodes roll back to the same HLC time, Retroscope achieves a consistent cut. In this section, we present different flavors of Retroscope snapshots, including the instant and retrospective snapshots, and their derivatives, incremental and rolling snapshots.

A. Snapshot Models

Instant snapshots. Figure 3 depicts our distributed snapshot system with logs at each node. Any node can become the snapshot initiator. The initiator starts an instant snapshot at the current HLC time at that node, T_{now}, and broadcasts messages to other nodes. Once a node receives the message, at time T_r, $T_r > T_{now}$, it removes the bound on the growth of its local window-log and starts copying its local state/database for the snapshot. We do not freeze the state/database during copying in order to keep the nodes available for serving normal operations. The copying of local states finishes at different T_f

[1]NTP violates the logical clock condition. In Figure 1, e $\underline{hb}$ f but the NTP timestamp of e, $pt.e$, is greater than that of f, $pt.f$.

[2]This can happen if $l.j$ is updated with $l.k$ from a received message, and $pt.j$ is still behind $l.j$.



(a) Instant (b) Retrospective

Fig. 4: Instant and retrospective snapshots.

times at the nodes. However, since each state transition has been recorded to the window-log with an HLC timestamp, this allows us to roll back any changes occurring after T_{now} in order to arrive the globally consistent snapshot at time T_{now}. Figure 3 uses a green-dashed arrow to illustrate the backward application of the window-log until reaching T_{now}.

Upon successfully finishing the local snapshot, the nodes report back to the initiator. Once the initiator receives all acks from the nodes, the global snapshot has been taken. Local snapshots are not transmitted to the initiator unless explicitly requested. For example, for checking whether a conjunctive predicate is violated, it would suffice to send the information about whether the local predicate is true at that local snapshot. A distributed reset service will also benefit from the in situ local snapshots since the system is to be reset on mostly the same set of machines.

Partial snapshot may result in case of a node failure or a lost or delayed message: if a node receives a snapshot message very late, then its window-log may have moved beyond the requested point, and it may not be able to take that snapshot. If partial snapshot does not provide sufficient information, the initiator can take another snapshot.

Retrospective snapshots. The natural extension to the instant snapshot protocol is to allow for retrospective snapshots to examine system state at some time $T_{past} < T_{now}$. The procedure to take a retrospective snapshot remains the same as with the instant snapshots except that the system needs to traverse further along the window-log. Figures 4a and 4b illustrate this comparison. The window-log size can be tuned to provide a compromise between resource utilization and the needed depth of retrospection. It is also possible to persist the window-log to disk to allow going further in the past.

Incremental snapshots. Taking multiple retrospective snapshots in succession can help examine how system states evolved, but that would be computationally expensive. To perform this in a time/space efficient manner, Retroscope provides forward-incremental and backward-incremental snapshots that capture the system state after and before a given base snapshot respectively. Figure 5 illustrates taking a backward-incremental snapshot to arrive to a time T_b using a snapshot at time T_{past} as the base point. In order to get a snapshot at T_b, it is unnecessary to traverse the entire log backwards from the current state, and instead the system can just undo the changes captured in the log between T_{past} and T_b, reducing processing time of the snapshot. In addition, disk storage can be saved by only keeping the changes between the base point and the new snapshot, albeit at the increased computational cost incurred upon snapshot retrieval.

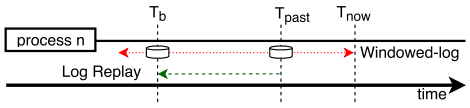


Fig. 5: Backward-incremental snapshot using bidirectional windowed-log on a single process.

Rolling snapshots. Monitoring and debugging services can benefit from a snapshot that can quickly move through the states of a distributed system. For applications, such as root-cause analysis, that examine the snapshots one at a time without the need to go back, keeping many incremental snapshots would be wasteful. Instead, rolling snapshots provide the ability to progress from one state to the next without preserving the prior snapshot, reducing the storage and processing time needed for long chains of snapshots.

B. Snapshot Limitations

Channel state snapshot. Capturing channel states can be managed by employing a similar window-log mechanism to keep both sent and received messages at each node. While some optimizations are possible in maintaining these messages log (such as, using pointers in lieu of data duplication, and recomputing instead of storing), these additional logs can unduly tax the system resources. Retroscope does not capture the channel states automatically. The lack of channel states, however, does not degrade the usefulness of our snapshots for many applications. Invariant predicates for distributed systems are often written over process states, rather than referring to the channel states. This is because, channels are unreliable in distributed systems, and important send/receive messages are encoded as process state anyways. In particular, AP systems in CAP categorization [15] are designed to be oblivious to channel state, and employ mechanisms like gossip to tolerate inconsistencies from lost messages and partitions.

Undo Limitations. While most operations are easy to record in the window-log and undo, operations that involve intrusive changes to the system state are exceptions to this rule. For example, dropping an entire database/table would require Retroscope to place the table into the window-log in order to be able to revert the operation. Even though the window-log may allocate more storage by using disk instead of RAM, keeping such large items would impact the performance and increase the storage requirements.

IV. Evaluation of Retroscope on Voldemort

We have implemented Retroscope as a Java library. The library exposes the API necessary to manage HLC time and maintain the window-logs and compute the differences between the two points of time on these logs. We used the library to add retrospective snapshot capabilities to a Voldemort key-value datastore, requiring roughly 1000 lines of codes to be added to Voldemort. These changes facilitate the HLC exchange, expose new snapshot API and allow to obtain the snapshots against Voldemorts internal BDB JE storage engine.

We evaluate our Voldemort Retroscope implementation on an Amazon EC2 cluster of 10 instances, each with 2 vCPUs

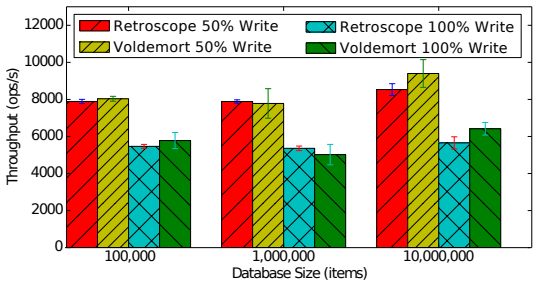


Fig. 6: Throughput comparison of snapshot enabled and unmodified copies of Voldemort.

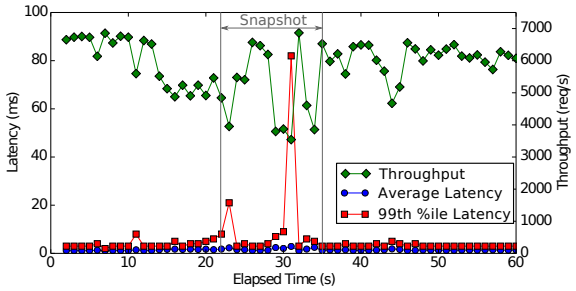


Fig. 7: Impact on Voldemort performance while taking an instant snapshot at 50% write workload.

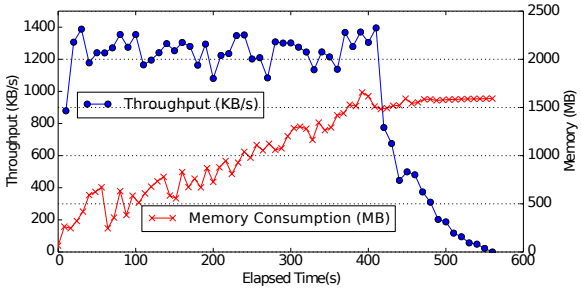


Fig. 8: Memory consumption of a signle Voldemort node.

and 8 GB of RAM. A separate VM is used for generating the workload by simulating 11 clients interacting with the system. We test Retroscope under different workloads, ranging from 10% write to 100% write, with random item selection unless otherwise stated.

A. Retroscope overhead

Retroscope introduces window-log and HLC instrumentation to Voldemort. In order to test how much overhead these additions cause over the unmodified Voldemort system, we conduct a set of experiments on our cluster using various database sizes, ranging from a small database of 100,000 items to a moderately large one at 10,000,000 key-value pairs. Since most of the changes are introduced on the write path of Voldemort, we use write intensive workloads of 50% and 100% writes to evaluate the Retroscope overheads.

Figure 6 shows the average throughput over 10 runs of the experiment for each of the chosen database sizes and workloads. Enabling snapshot capability degrades performance only slightly, despite the added overhead of HLC timestamp in each message and the need to maintain an in-memory window-log. For the small databases we observe 1.8% overhead in throughput, while the largest databases show decline of up to 10% —although we also observe bigger variances during those tests. The latency overheads were also under 10%.

After demonstrating the overhead of Retroscope instrumentation on performance, we next evaluate the overhead of actually taking a snapshot. Figure 7 illustrates how throughput and latency for normal read and write operations are affected while instant snapshot is progressing. Similar to our previous experiment, we used a Voldemort cluster with a database of 10,000,000 items of 100 bytes each. Voldemort is configured with replication factor of 2 nodes, meaning that only 2 machines in the cluster maintain the copy of each key-value pair. We collect the throughput, average latency, and 99th percentile latency for every 1 second of execution.

Soon after the snapshot initiation, we observe some performance degradation and variance in clusters performance. The overall throughput degrades by 18%, and latency increases by 25% during the snapshot execution. We also observe a spike in 99th percentile latency. The performance decline is attributed to an increased disk load: Voldemort needs to flush memory cache to disk, perform a copy of BDB JE files, and rollback the changes. Using a separate disk for snapshots would reduce disk contention and boost system's performance.

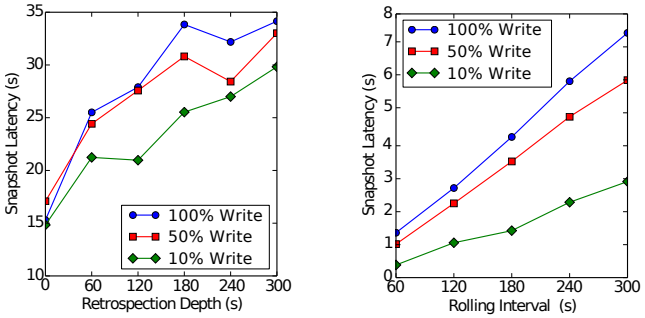
B. Reach of Retroscope snapshots

Our Voldemort prototype keeps the window-log in memory, as such the amount of RAM available to Retroscope becomes a limiting factor for the retrospection. Here we evaluate the memory-usage overhead of Retroscope, and its limitations on the extent of past state reach for retrospective snapshots.

Figure 8 shows the memory usage of a single Voldemort node with unbounded window-log serving write requests. During the first 410 seconds of operation, the node is not under memory pressure and shows high performance of 5004 operations per second or 1251 Kb/s on average, however as the memory consumption gets closer to a 2GB limit, JVM spends more time in garbage collection, greatly reducing Voldemort's performance. JVM seizes to operate with *OutOfMemoryException* after 560 seconds of runtime. Higher throughput will result in smaller maximum window-log size, but distributing the workload among the nodes in a system will prolong the reach of retrospection available to the operator.

C. Retroscope snapshot latency

The latency of a Retroscope snapshot is correlated with how far back it needs to reach, as snapshots going further in the past must traverse larger log segment to compute the difference between current and past states. Far reaching snapshots also need to perform larger number of disk I/O operations. Figure 9a shows an increasing cost of taking a full snapshot of a 10 million items database under the workloads of various write intensity. As expected, an instant snapshot, taken at 0 seconds back, is the fastest one and increasing the reach of retrospection also increases the snapshot latency. We also observe that write-intensive workloads take longer to process



(a) Depth of retrospection. (b) Rolling snapshot.

Fig. 9: Latency for retrospective and rolling snapshots.

snapshot. A snapshot under a 100% write workload takes as much as 33% longer to complete compared to a 10% write workload. This is due to the bigger window-log accumulated for the same time interval. Underlying BDB JE datastore, used as a default storage mechanism in Voldemort, also impacts the snapshot latency. Under heavy load BDB undergoes frequent log cleaning, blocking the snapshot routine from making a copy until the cleaning is complete. We learned that the log cleaning takes as much as 15 seconds on test database, thus a single node undergoing log cleaning negatively impacts the performance for the entire cluster.

Figure 9b illustrates the latency increase for rolling snapshots on the 10 million item database. Rolling snapshots do not perform full data copy, and they are bottlenecked only by log traversal and application of the differences to the base snapshot on disk.

V. Improving Retroscope Performance

We describe a few optimizations for Retroscope. These techniques are orthogonal and can be combined together for improving the performance further.

Memory Utilization. High memory utilization is a major limiting factor for Retroscope. Reducing the memory required to maintain the window-log will allow a higher depth of retrospection We have a few possible ways to improve on memory utilization. One solution is to apply data compression methods to the log. Keeping log overheads low can also help, especially for logs with large number of small items. Our current Java realization has fairly high JVM overheads for each item, and we can reduce these implementation overheads by using a lower-level language, such as C, for window-log.

Deferred snapshots. When the database size is large, taking a snapshot at all nodes simultaneously may incur significant load, reducing the throughput for the database clients. However, Retroscope does not require for local snapshots to be taken simultaneously by leveraging HLC timestamps. With our system, each node can take a snapshot individually, and the collated local snapshots still represent a consistent global snapshot. Having a snapshot where nodes start in a deferred off-phase manner can balance the processing in time, and flatten the snapshot overheads.

Deferred snapshots can be implemented using the node IDs to dictate the snapshot order in the cluster in such a way that no more than k nodes fully overlap in time when taking a snapshot. With this approach some nodes will capture more history using the window-log. This tradeoff makes sense if smoothly shedding the snapshot load in time is beneficial for ensuring a high throughput capacity for the clients.

Speculative snapshots. A rolling snapshot performs less work compared to a full snapshot as it reuses the old snapshot and does not need to copy data. This introduces the opportunity to take occasional speculative snapshots so that when a snapshot is actually needed we may have a nearby snapshot to leverage. When that is the case, we can perform a rolling snapshot and complete the request in less time.

There is a tradeoff associated with taking speculative snapshots: we are making a bet that an actual snapshot will be requested soon. Using historical data or identifying triggering conditions for a snapshot can help us make a better prediction for taking a speculative snapshot. That being said, a mispredicted speculative snapshot can also find some use as a backup.

VI. Related Work

Berkeley DB Retrospection. Retro allows past state querying and inspection against a Berkeley DB [16]. Retro, does not have the ability to inspect arbitrary past state, instead all retroactive snapshots must be planned ahead of time by issuing a *snapshot now* command. At a later time, users can query the database for items in any of the past snapshots. The system makes its own retrospective component persist on the disk alongside the BDB log without modifying the code responsible for the current state requests, preserving the API compatibility with BDB. Unlike Voldemort, which uses many independent BDB JE instances on different servers, Retro is limited to a single BDB deployment.

Eidetic systems. Eidetic systems can recall any past state that existed on the computer, including all versions of all files, the memory and register state of processes, interprocess communication, and network input. In [17], the authors develop an eidetic system by modifying Linux kernel to record all nondeterministic data that enters a process: the order, return values, memory addresses modified by a system call, the timing and values of received signals, and system time. The major space saving technique in that work is to use model-based compression: the system constructs a model for predictable operations and records only instances in which the data differs from the model. That is, the system only saves new input and nondeterministic choices and can recompute everything else. The results in [17] are for single CPU machines and do not account for issues in distributed systems.

Freeze-frame file system. The Freeze-Frame File System (FFFS) [18] uses HLC [10] to implement retrospective querying on the HDFS file system [19]. FFFS has multiple logs, persisting on low-latency storage, to capture changes on HDFS NameNode and DataNodes. An indexing scheme is used to access the logs and retrieve requested pages from the past. FFFS required intrusive changes to the underlying system and replaced HDFS append-only logs with multiple HLC-enabled logs and indexes. FFFS records every update to

data and metadata, and in effect implements a multiversion data store. In contrast, Retroscope focuses on low overhead design of a snapshot primitive, and keeps a window-log for undoing recent updates to take a retrospective snapshot.

Distributed tracing tools. There has been several work on distributed tracing tools [20]–[22] for troubleshooting of distributed systems. Two main challenges in tracing are that instrumentation is decided at development time, and dynamic dependencies in the distributed system of systems. Pivot tracing [22] attempts to overcome these challenges by using dynamic instrumentation and causal tracing. In particular, it models systems events as tuples in a streaming distributed dataset, and dynamically evaluate relational queries over this dataset using the "happened-before join" operator.

Retroscope takes a complementary approach to the tracing work. In a Retroscope snapshot, state across nodes is being considered at a given physical time in a globally consistent manner. This is particularly useful for evaluating cross-node consistency/synchronization predicates.

Conflict handling. Last write wins rule causes problem for distributed key-value stores that rely on NTP timestamping, like Cassandra [23]. Retroscope could help in investigating what went a miss. As the preventative measure, adopting HLC and substituting it for NTP would help resolve the last write wins caused problems.

VII. Concluding Remarks

We introduced Retroscope for performing lightweight, incremental, and retrospective distributed snapshots. Retroscope leverages HLC timestamping to collate node-level independent snapshots for obtaining a coherent global consistent cut. As such it avoids the inconsistent cut problems associated with NTP timestamping, and the inscalability of VC timestamping with respect to the number of nodes in the system. Retroscope provides an efficient implementation of retrospective snapshots by utilizing a configurable-size window-log to capture recent operations, and avoids the cost of maintaining a multiversion copy of the entire system data. Moreover, Retroscope introduces incremental and rolling snapshots that leverage an existing snapshot to reduce the cost of constructing new snapshots in that proximity. We implemented Retroscope for Voldemort and evaluated its performance under different workloads.

An important use case for Retroscope is re-establishing data integrity after a failure or security attack. If there have been bad inputs around time T, the operators can revert the datastore to a safe/clean state in the recent past of T to purge the bad inputs. Since Retroscope provides rolling snapshots, the operators can explore around the problematic time interval, and perform step-by-step debugging and root cause analysis. Retroscope also facilitates the global reset/revert operation. Such reset takes $\sim$8 seconds for a test 1GB Voldemort database. In future work, we aim to provide programmatic toolkit support for snapshot evaluation and distributed reset.

Acknowledgments

This project is in part sponsored by the National Science Foundation (NSF) under XPS-1533870, XPS-1533802

References

[1] O. Babaoglu and K. Marzullo, *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms*, in Sape Mullender, editor, *Distributed Systems*, pages 55–96. Addison Wesley, New York, NY, 2nd edition, 1994.

[2] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, Feb 1985.

[3] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," in *11th Australian Computer Science Conference (ACSC)*, 1988, pp. 56–66.

[4] F. Mattern, "Virtual time and global states of distributed systems," *Parallel and Distributed Algorithms*, vol. 1, no. 23, pp. 215–226, 1989.

[5] V. K. Garg and C. Chase, "Distributed algorithms for detecting conjunctive predicates," *International Conference on Distributed Computing Systems*, pp. 423–430, June 1995.

[6] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[7] D. Mills, "A brief history of ntp time: Memoirs of an internet timekeeper," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 2, pp. 9–21, 2003.

[8] J. Du, S. Elnikety, and W. Zwaenepoel, "Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks," in *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*. IEEE, 2013, pp. 173–184.

[9] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," *Proceedings of OSDI*, 2012.

[10] S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, "Logical physical clocks," in *Principles of Distributed Systems*. Springer, 2014, pp. 17–32.

[11] "Project retroscope," https://github.com/acharapko/retroscope-lib, 2016.

[12] "Project voldemort," http://www.project-voldemort.com/voldemort/.

[13] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

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

[15] E. Brewer, "Towards robust distributed systems," in *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, 2000, p. 7.

[16] R. Shaull, L. Shrira, and B. Liskov, "A modular and efficient past state system for berkeley db." in *USENIX Annual Technical Conference*, 2014, pp. 157–168.

[17] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. Chen, "Eidetic systems," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, 2014, pp. 525–540.

[18] W. Song, T. Gkountouvas, K. Birman, Q. Chen, and Z. Xiao, "The freeze-frame file system," in *Proc. of the ACM Symposium on Cloud Computing (SoCC16)*, 2016.

[19] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. IEEE, 2010, pp. 1–10.

[20] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *Proceedings of the 4th USENIX conference on Networked systems design & implementation*. USENIX Association, 2007, pp. 20–20.

[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," Google, Inc., Tech. Rep., 2010. [Online]. Available: http://research.google.com/archive/papers/dapper-2010-1.pdf

[22] J. Mace, R. Roelke, and R. Fonseca, "Pivot Tracing: Dynamic causal monitoring for distributed systems," *Symposium on Operating Systems Principles (SOSP)*, pp. 378–393, 2015.

[23] A. Lakshman and P. Malik, "Cassandra: Structured storage system on a p2p network," in *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, ser. PODC '09, 2009, pp. 5–5.