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 instant and retrospective snapshots, Retroscope also provides incremental and rolling snapshots that utilizes an existing full 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 provide implementations of Retroscope for the Voldemort distributed datastore and Hazelcast in-memory data grid, and evaluate their 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, as 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 pairwise consistent cut requires waiting out the clock uncertainty [8], [9], which makes it unsuitable for retrospective snapshots.

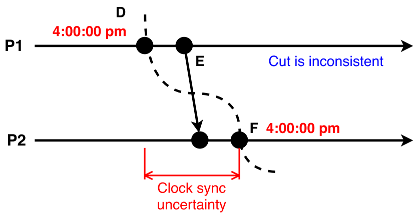


Fig. 1: Using NTP only fails to take consistent shot

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 utilizes 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 lost updates.

We design and develop Retroscope as a standalone library so it can be easily added to existing distributed systems. Our Retroscope library implementation is available on github as an opensource project [11]. To showcase Retroscope and evaluate its performance, we provide two case studies: Retroscoping Voldemort, and Retroscoping Hazelcast.

Retroscoping Voldemort. Voldemort [12] is a popular opensource system used in LinkedIn [13], that implements a Dynamo-like highly available distributed key-value store [14]. Our Retroscope instrumentation of Voldemort leverages the Retroscope library functionality and required less than 1000 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 node cluster, taking and finalizing a snapshot for current time, T_{now}, requires $\sim$15 seconds. After a snapshot is taken, it takes only $\sim$100msecs for taking an incremental snapshot in the vicinity of that snapshot. The snapshots can go back to $\sim$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.

Retroscoping Hazelcast. Hazelcast [15] is a popular opensource in-memory data grid system. Our Retroscope instrumentation for Hazelcast, tested with 1GB data in a 3 nodes cluster, has only 7.8% overhead when maintaining the window-log with a high throughput 100% write workload. An invoked snapshot incurs up to 7.3% overhead on the throughput during the snapshot completion time. Current state snapshots require little time to finish, $\sim$100msec, because they are taken in-memory. Snapshots that explore history need more time. For instance, a high throughput 100% write workload causes the window-log to grow fast, making a snapshot of 1 hour in the past take as long as 45 seconds to complete.

Outline of the rest of the paper. We describe HLC timestamping next. In Section III, we explain the basic mechanisms of Retroscope snapshots. In Section IV we present our Retroscope implementation for Voldemort and Hazelcast. We evaluate and quantify the performance of our Voldemort and Hazelcast snapshot services in Sections V & VI. We discuss extensions in Section VII. 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].[1] 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.[2]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.

[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.

[2]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, $pt.f$.

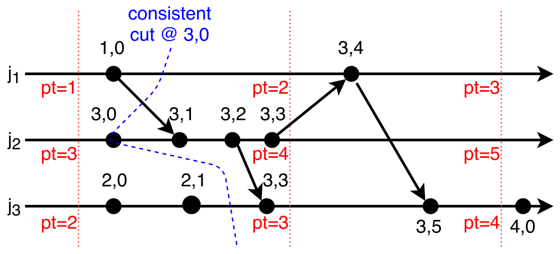


Fig. 2: Example of HLC operation with $\epsilon = 2$ on 3 process. Dashed lines denote the physical clock ticks with timestamp for each process next to it. HLC time is written above each event in the "l,c" format.

Since HLC [10] is a hybrid of NTP and LC, HLC 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, in HLC since logical time is driven by the physical time, it is easy to find events at each node with the same HLC values where all are 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 [3], 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 [16].

Our HLC implementation in Java is based on the HLC implementation of CockroachDB [17] 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: The only effect of degraded NTP synchronization is to increase the drift between l and pt values and to introduce 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

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

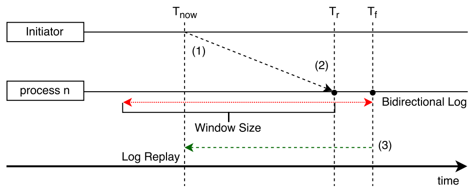


Fig. 3: Instant distributed snapshot.

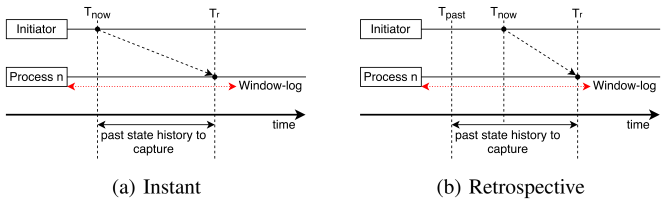


Fig. 4: Window-log differences between instant and retrospective snapshot.

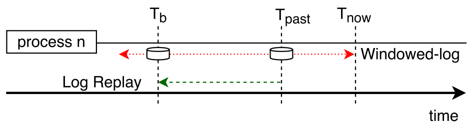


Fig. 5: Backward-incremental snapshot using bidirectional windowed-log on a single process.

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, rolling, and concurrent 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 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 wasteful. 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.

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.

Multiple concurrent snapshots. Retroscope allows for multiple initiators to take overlapping or concurrent snapshots. As an optimization, Retroscope can detect concurrent snapshots running on the node and convert some of them to incremental or backward-incremental snapshots in reference to one or more of the currently executing snapshots.

B. Snapshot Limitations

Channel state snapshot. Capturing channel states can be managed by employing a similar window-log mechanism. However, for full generality, both sent and received messages should be logged 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

TABLE I: Basic Retroscope API

Method	Description
HLC Management	
timeTick()	HLC time tick for local event
timeTick(HLCTime)	HLC time tick caused by remote event with a timestamp of HLCTime
wrapHLC(message)	Performs an HLC time tick for local event and prepends it to the message
unwrapHLC(message)	Gets HLC from message, performs HLC time tick and returns the new HLC time
Log Management	
appendToLog(logName, K, oldV, newV)	Appends a change of item K: *oldV* $\rightarrow$ *newV* to *logName*
computeDiff(logName, timeInPast)	Computes a difference between the current and prior state at *timeInPast* for *logName*
computeDiff(logName, startTime, endTime)	Computes a difference between states at *startTime* and *endTime* for *logName*

resources. Retroscope implementation does not capture the channel states. 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 [18] are designed to be oblivious to channel state, and employ mechanisms like gossip to tolerate inconsistencies from lost messages and partitions.

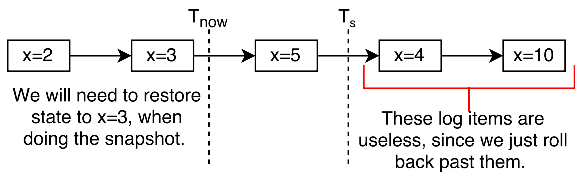
Undo Limitations. While most operations are easy to record in the window-log and undo, operations that involve intrusive change to the system state are exceptions to this rule. For example, dropping an entire database/table would require Retroscope to place the table into the window-log in order to be 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. Implementing Retroscope

We implement Retroscope in Java as a standalone library. Table I summarizes the Retroscope API.

HLC Management API provides functionality to manage HLC by invoking *timeTick()* and *timeTick(HLCTime)* methods. To accommodate for various architectures and network protocols, Retroscope also provides additional helper methods to work with timestamps in different representations, such as 64-bit integer or slice of a byte stream.

The window-log API constitutes of *appendToLog* and *computeDiff* methods. Append functionality allows an application to record the changes to its state in Retroscope's in-memory window-logs. These logs use a sliding window and provide multiple ways to control the maximum size, including truncating the state history after a given duration or erasing the old history when the size of the log reaches a given threshold. With the help of *computeDiff* methods, Retroscope users can calculate the difference between any two points in the log.



T_now
T_s
x=2
x=3
x=5
x=4
x=10
We will need to restore state to x=3, when doing the snapshot.
These log items are useless, since we just roll back past them.

Fig. 6: Redundancy that exists in the backward traversable log. All operations after T_s are shadowed by the results of traversal from T_s to T_{now}

Such differences can later be applied to a current state to get a full snapshot or to an existing snapshot to achieve incremental or rolling snapshot.

As the log is traversed by *computeDiff*, we may encounter multiple operations on the same key, but only the last operation traversed for a key will have an impact on the final state. Such operation shadowing is illustrated in Figure 6. With *computeDiff*, Retroscope eliminates the redundancy and compacts the log into the key-value map of changes between two log points. The efficiency of the compaction depends on the workload and the size of a compaction window. Many real-world workloads exhibit the tendency to access certain items more often, creating access hot-spots that benefit the compaction, as redundant same-key operations can be eliminated.

The following formula gives a rough estimate for the memory required by the window-log on a single machine in the distributed system: $S_t = \Delta t R_a(2S_i + S_k + S_{HLC} + S_o)$. S_t is the total size of the log, R_a is the average rate of log append operations per second, S_i is the average size of the data in the log item, S_k is the average size of the item key, S_{HLC} is the size of HLC timestamp, S_o is the size of various overheads due to the implementation and system requirement, and Δt is the duration of the log window in seconds. In our Retroscope implementation, S_{HLC} is 8 bytes and S_o is no less than 152 bytes and depends on JVM settings and paddings incurred due to the data.

A. Retroscoping Voldemort

Here we describe how we add the retrospective snapshots to Voldemort key-value datastore [12] using our Retroscope library. Voldemort is an open source implementation of Amazon's Dynamo [14], and is used at LinkedIn. Voldemort optimistically replicates data across multiple nodes, and falls under the AP system categorization [18], as it favors availability in the presence of network partitions. It uses version vectors along with physical clock timestamps to detect and resolve inconsistencies in the data that may arise due to the partitions or node failures. Our changes to the system were minimal, totaling around 1000 lines of code for adding HLC to the network protocol, recording changes in the Retroscope window-log, performing snapshot on BDB JE storage, and adding snapshot API to the administrative client.

Adding HLC. Figure 7 depicts the high level architecture of Voldemort snapshot system. Voldemort keeps the inter-server communication to a minimum. In the typical usage pattern of Voldemort's Java API, a client is directly responsible

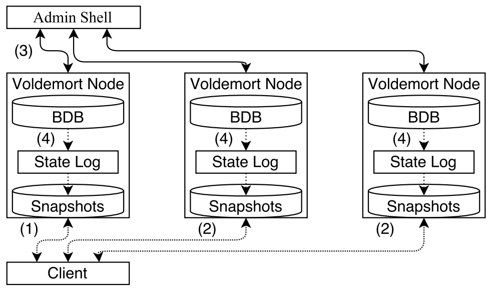


Fig. 7: Our Snapshot implementation in Voldemort. (1) Client communicates with Voldemort master for that key. (2) Client replicates the item to other servers. (3) Admin shell acts as snapshot initiator. (4) Nodes takes local snapshots.

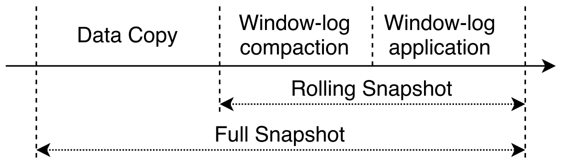


Fig. 8: Three stages of taking a Retroscope snapshot

for replicating an item to a set of nodes associated with the item's key.[4]With this replicating scheme, Voldemort nodes keep an indirect communication with each other through clients. However, HLC is still functional in this configuration, as the client contacts the nodes and pass the timestamps along with each message.

The snapshot initiator is an HLC-enabled administration application that can transmit the snapshot request for a specific HLC time to all the nodes in the cluster. The initiator can also check the progress of snapshot at each node and restart the snapshot if needed. Other clients are oblivious to a snapshot being taken in the system and continue normal operation.

HLC time is kept within the Retroscope as we use the library to perform time ticks for local events and encode timestamps for network transmission. We also replace NTP timestamp associated with each item with HLC timestamps.

Implementing the window-log. Voldemort nodes use Berkeley DB Java Edition (BDB) database as the underlying storage. We considered using BDB's log for capturing operations to be reversed, however, in the end, we went with a more general approach and used in-memory window-log provided by Retroscope. Not relying on BDB's log enabled us to improve the performance by maintaining the window-log in-memory and not needing to read from disk. In addition, BDB log cleaning was a problem to be managed, and could have negatively impacted the disk utilization.

Taking a snapshot. Figure 8 illustrates the execution stages of a retrospective snapshot in Voldemort, for both full and rolling/incremental snapshots. In the data copy stage, we obtain a copy of the data to be used for the snapshot. This copy may not correspond to the requested snapshot time. Window-log compaction phase is responsible for computing the difference between the data obtained earlier and the desired snapshot time. This step is handled by Retroscope library. Window-log application step modifies the data obtained in the first step by replaying the computed differences to reach the state at the requested time. A full snapshot is one where we create a full backup of the current database and then replay the window-log backwards until we reach the requested time. A full snapshot requires all three stages, and it produces a complete, ready to use copy of underlying Voldemort storage.

A rolling or incremental snapshots use another snapshot as the reference base, thus eliminating the need to perform an expensive data copy every time. A rolling snapshot performs only the last two stages of the snapshot process and applies the differences between base snapshot and new desired time to the base, replacing the original snapshot in the process with the one for requested time.

Similarly, incremental snapshots only perform the compaction phase, after which the compacted log is saved for future use along with the information about the parent snapshot. In case an incremental snapshot needs to be used, the system takes the compacted log difference between the two snapshots and computes the full state of the system by applying the changes recorded in the compacted log to the base snapshot. The main advantage of this approach is the small overhead at the snapshot taking time and low storage requirements. Next we describe the phases of the retrospective snapshot in detail.

Data copy. Retroscope makes a copy of live database without the need to lock the datastore and deny client requests. For this, it follows the procedures outlined in Oracle's documentation for BDB on how to perform a hot database backup [19]. The BDB captures all the data and data changes in a sequence/succession of write ahead log files that frequently undergo cleaning to eliminate all old values masked by the new writes to conserve disk space. Upon receiving the snapshot request at time T_r, Retroscope makes BDB to flush all changes to disk and close all of its existing log files, so that no further data mutations are written to those log files. At this point the BDB log captures the data-state at time T_r. Since Retroscope allows Voldemort clients to serve requests while performing the snapshot, all mutations happening after T_r are preserved in a new BDB write ahead log file that is not part of the snapshot. The backup of BDB is constructed by copying the closed BDB log files to a new directory, effectively capturing the state of the node at time of message receive T_r.

Window-log compaction. The difference between states of the data we have obtained and the desired snapshot is captured in Retroscope's window-log. We use the window-log management API to compute the difference between the states at T_r and snapshot time $T_{snapshot}$.

Window-log application. The difference map is then used to arrive to the target snapshot. In full snapshot, the difference is applied to the copy of the data we have made earlier, while rolling snapshot modifies the base point, or parent, snapshot. Since a data copy or parent snapshots are instances of BDB, we can open such database for changes and add all the differences to get the data at $T_{snapshot}$. The incremental

[4]The Python API enables server-side routing, however it was not supported by Voldemort, and had problems when we tried to use it.

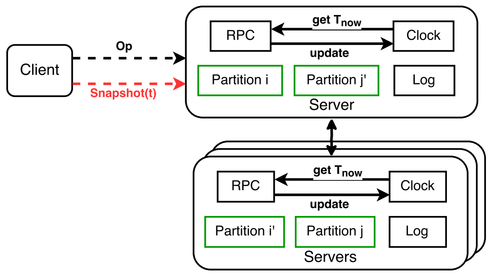


Fig. 9: Hazelcast with HLC

snapshots do not utilize the computed difference right away. Instead the difference map is stored on disk and its application is delayed until the snapshot is requested at a later time. We keep each node's snapshots locally and depending on the application, the snapshots can be used locally or made available to the initiator and/or other nodes upon the request, e.g., by copying the local snapshot to a mountable shared storage, such as EBS in AWS.

B. *Retroscoping Hazelcast*

To showcase Retroscope snapshots in the context of a distributed in-memory data store, we implemented Retroscope as an internal service in Hazelcast [15], a popular open-source in-memory data grid. The Hazelcast API provides several distributed data structures build on top of the NodeEngine with RPC service. These data structures are represented as sets of key-value records internally. Hazelcast distributes the data by partitioning the keys. Each key is hashed over a total of 271 partitions (by default) to identify the partition container it belongs to. The partitions are distributed equally among the nodes of the cluster, with configurable backups of partitions in neighboring nodes for redundancy. In the example shown in Figure 9, the server on top holds partition i and the backup of partition j, while the bottom server maintains partition j and the backup of i.

We implemented Retroscope in Hazelcast such that when the snapshot feature is enabled, an HLC timestamp is implanted in every remote operation in the RPC layer. The RPC operations injected with HLC are not limited to data query/updates, but also include replication, metadata exchange, health monitoring, etc.

Two snapshot methods were added to Hazelcast's existing distributed `Map<K,V>` datastructure: (1) `snapshot()`, without a parameter, initiates the snapshot at T_{now} on the directly connected server, then broadcasts to the entire cluster with T_{now} as the HLC timestamp. (2) `snapshot(`t`)`, by given a specific time in the past as its parameter, the snapshot is initiated with the corresponding HLC timestamp t. If Δ is the desired amount of time to refer back to, t can be easily given as system's current time in milliseconds using $t_c - \Delta$.

We implement the snapshot operation at the partition level, i.e. snapshots are taken concurrently within each partition. This design choice simplifies the relation between snapshots because partitions may be moved around among member nodes. More importantly, this also reduces the impact on normal operation's overall throughput with fine-grained concurrency control. Once a server receives a snapshot request, a snapshot Op_i is invoked on partition #i. Op_i first copies the current partition state, then traverses back the log for t. Each node has an aggregator that collects all snapshots of partitions it holds. The aggregator is also responsible for persisting the snapshot to the disk asynchronously. Window-log in Hazelcast implementation is bounded by a user-specified maximum size.

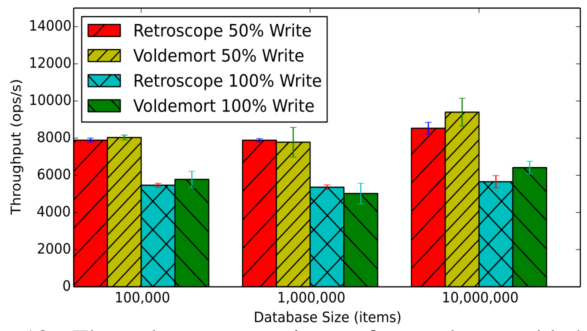


Fig. 10: Throughput comparison of snapshot enabled and unmodified copies of Voldemort.

V. Evaluation of Retroscope on Voldemort

We evaluate our Voldemort Retroscope implementation on an Amazon EC2 [20] cluster of 10 instances, each with 2 vCPUs 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. The evaluation is conducted through a snapshot API we have exposed to the Voldemort administrative client. The main method used is *doSnapshot(HLCtime, store, snapshotDirectory, baseDirectory)*, which allows taking snapshots in all supported modes. For instance, if *baseDirectory* is empty, Voldemort performs a full snapshot, but if a parameter is specified, the data-store will carry out either incremental or rolling snapshot, depending on *snapshotDirectory*.

A. *Retroscope overhead*

Retroscope introduces window-log and HLC instrumentation to Voldemort. In order to test how much overhead these additions cause over the original 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 10 shows the average throughput over 10 runs of the experiment for each of the chosen database size and workloads. Very little difference in performance is incurred from enabling snapshot capability, 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

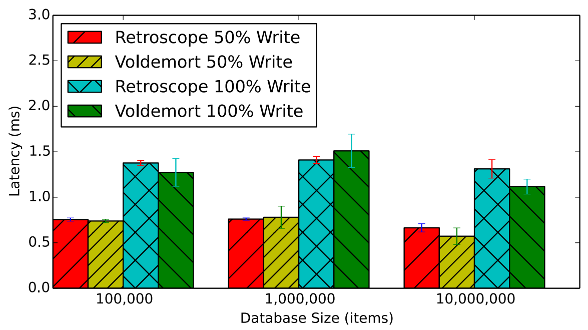


Fig. 11: Latency comparison of snapshot enabled and unmodified copies of Voldemort.

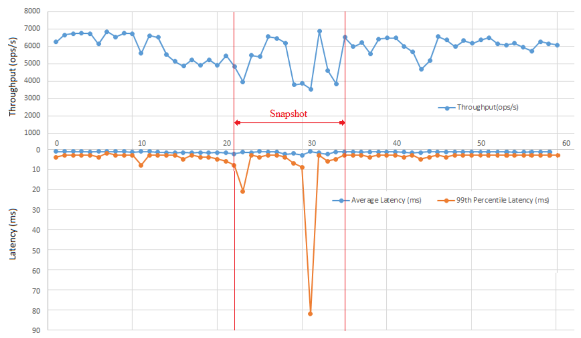


Fig. 12: Impact on Voldemort performance while taking an instant snapshot at 50% write workload.

overhead of up to 10% —although we also observe bigger variances during those tests.

We use the same set of experiments for evaluating the latency overhead of Retroscope instrumentation, and find that the average latency for Voldemort showed little degradation from our additions as seen in Figure 11.

After demonstrating the overhead of Retroscope instrumentation on performance, we next evaluate the overhead of actually taking a snapshot. Figure 12 illustrates how throughput of a system is affected while instant snapshot is progressing. Retroscope snapshots allow Voldemort to stay available for client requests during the snapshot. Similar to our previous experiment, the Voldemort cluster is used 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.

In Figure 12, we observe some performance degradation and variance soon after the snapshot is initiated on the cluster. The overall throughput degrades by 18%, and latency increases by 25% during the snapshot execution. We also observe a spike in 99% latency during the snapshot processing time. The decline in performance can be attributed to multiple factors: flushing any in-memory changes held by the underlying BDB storage engine to disk, copying the database and writing to window-log changes to the BDB copy. Using a separate disk to store snapshot would alleviate some disk contention and improve system performance.

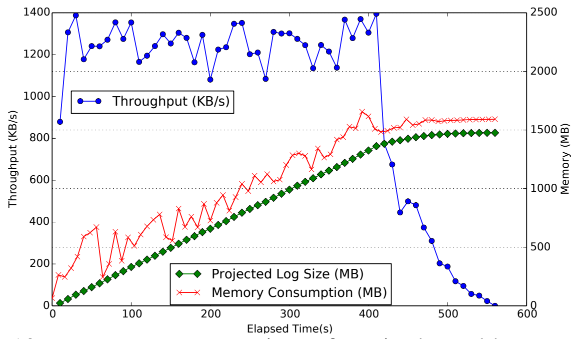


Fig. 13: Memory consumption of a single Voldemort node under write load.

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 13 shows the memory usage of a single Voldemort node with unbounded window-log serving write requests. We include a projected log size based on our estimation formula in Section IV. 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 Voldemorts performance. JVM seizes to operate with *OutOfMemoryException* after 560 seconds of runtime. Obviously, higher throughput will result in lower maximum window-log size, but distributing the workload among the nodes in a system will prolong the reach of retrospection available to the operator.

Our formulaic estimation foresees that in the first 410 seconds of experiment, the window-log would require at least 1362 MB of memory. In fact, the JVM was using 1509 MB. The difference between our estimation and true memory utilization are mostly due to the rest of the Voldemort system not accounted in the estimate.

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 will likely also have more data to revert, increasing the number of disk I/O operations. Figure 14 shows the time of taking a full snapshot of a 10 million items database under the workloads of various write intensity. We measure the snapshot latency as the time between issuing a snapshot request and the time last node completes the procedure. The figure shows an increasing cost of taking a full snapshot. 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 snapshot. A snapshot under

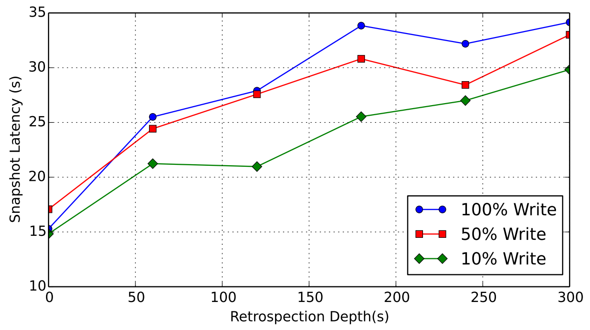


Fig. 14: Latency for different depth of retrospection

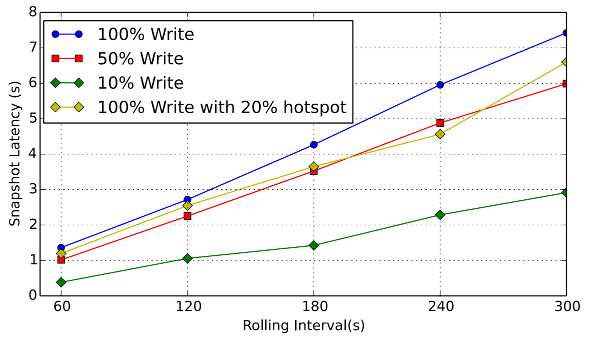


Fig. 15: Latency of rolling snapshot

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. Overall snapshot latency is also impacted by BDB JE operation. Under heavy load, BDB undergoes frequent log cleaning, however this procedure keeps the data files open, preventing snapshot routine from making a copy, thus a system must wait for cleaning to complete. We employ our Retroscope library to monitor of BDB JE activity itself and learn that the cleaning procedure takes as much as 15 seconds on the database this size and a single node undergoing log cleaning negatively impacts the latency for the entire cluster.

The latency increase for longer retrospection in the previous experiment is due to the time it takes to compute and write larger changes to a disk. However, the high variances in BDB operation make it difficult to observe the pattern of latency degradation. Rolling snapshots are less susceptible to effects of BDB cleaning. We use a 10 million items database of 75 byte items under a 10%, 50% and all write workloads with rolling snapshots to measure the latency. Figure 15 shows a linear latency increase as the rolling interval grows.

In order to test how frequently accessed items impact the snapshot performance, we run a workload where 20% of the items are accessed 80% of the time. The latency shows improvement due to more efficient log compaction as more of the keys in the window-log shadow each other, reducing the amount of data written to disk. The improvement is little for the 10% and 50% write-intensive workloads.

VI. Evaluation of Retroscope on Hazelcast

To evaluate our Retroscope implementation in Hazelcast, we conduct a series of experiments on a cluster of 3 AWS EC2 medium instances. Our micro-benchmark clients run in a separate instance within the same datacenter. Each experiment consists of 10 clients generating 100% write workload over the same `Map<K,V>` object. Each write size is 100 bytes.

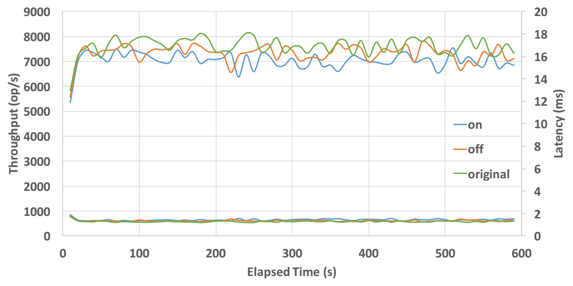


Fig. 16: Retroscope overhead in Hazelcast

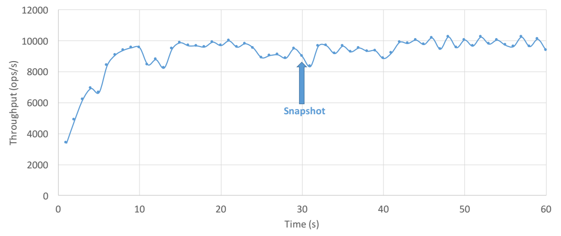


Fig. 17: Overhead of an ongoing snapshot operation on Hazelcast throughput

A. Retroscope overhead

Figure 16 shows the overhead of Retroscope instrumented Hazelcast, compared to the original Hazelcast system in terms of average throughput and latency. Retroscope instrumented Hazelcast is tested under two modes: "on" means window-log is enabled, and "off" means window-log is disabled but HLC instrumentation is still present. Our benchmark records average throughput and latency every 10 seconds on a workload of 10 million keys. The experiment shows that the off-mode has a 3.9% overhead in throughput, while the on-mode has a slightly bigger overhead of 7.8% compared to the original.

A snapshot request incurs additional overheads over the normal Hazelcast operation. Most importantly, while the nodes perform the data copying, they lock the keys being copied, making all requests writing these keys to block momentarily. To evaluate the impact of an ongoing snapshot operation on the system throughput, we run 10 clients with 100%-write workload, and then make one of the clients issue a `snapshot()` request at the 30th second mark. Figure 17 shows that the throughput only drops 7.3% in next second and then returns back to normal.

B. Reach of Retroscope snapshots

To evaluate how far back Retroscope full snapshots can reach, we start the workload at time t_0, with ten clients sending 100% write requests. The average throughput is taken every second. We take a snapshot of t_0 at the end of every 5 minutes, Figure 18 shows both the latency of each snapshot and impact on background throughput. Given 4GB physical memory and 2GB window-log limit, even in the worst case scenario of pure write workload, Retroscope can travel back 60 minutes

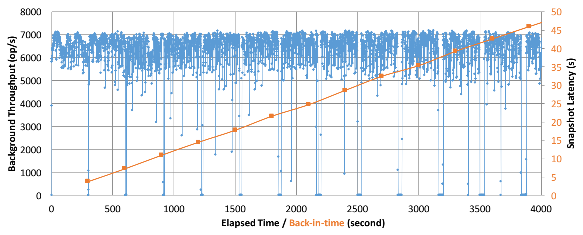


Fig. 18: Snapshot latency and impact on Hazelcast throughput with increased window-log size

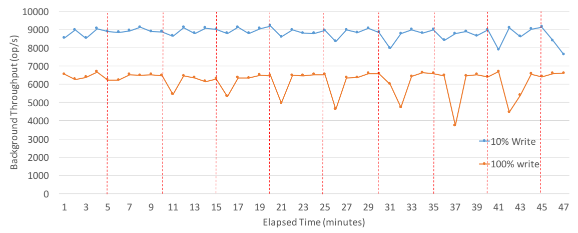


Fig. 19: Snapshot throughput for 10% and 100% write workload in Hazelcast

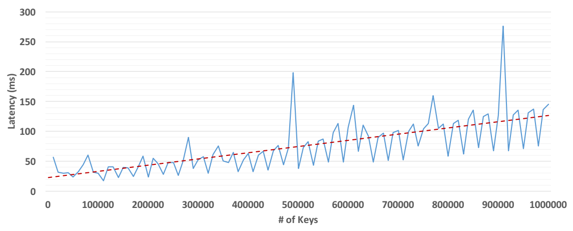


Fig. 20: Snapshot latency with respect to varying Hazelcast database sizes

with an end-to-end snapshot latency of less than 45 seconds. Figure 19 shows the differences between 10% and 100% write workloads over average throughput every minute. We took a snapshot of t_0 every 5 minutes, the drop of throughput due to snapshot is less noticeable with the 10% write workload.

C. Retroscope snapshot latency

The snapshot operation latency depends mainly on the database size, since a large database takes longer to copy all the data. Figure 20 shows an experiment where we generate 10000 new records of 1000 bytes at every step, and measure the snapshot latency for each of the database sizes. The red line indicates the linear trend of end-to-end snapshot latency with respect to increasing size of the database. At the end of the experiment, database reaches 1GB, and the snapshot completes in around 100 milliseconds.

The snapshot operation latency also depends on the write workload. A heavy write workload will cause a larger window-log that Retroscope needs to traverse back to reach T_{now}. However, for instant snapshots this effect remains small compared to the effect of the database size.

VII. Improving Retroscope Performance

We describe four optimizations for Retroscope performance. 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 can apply data compression methods to state history to improve the log memory utilization. Our current log implementation is realized in Java and has high memory overhead for the internal housekeeping needs. In applications dealing with small data items, this penalty is especially noticeable. We can reduce 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 load that reduces the throughput for the database clients. However, Retroscope snapshot does not need to be taken simultaneously at all the nodes. Retroscope leverages HLC timestamps so that each node can take a snapshot individually, and the collated snapshots still represent a consistent global snapshot. Having a snapshot where nodes start in a deferred off-phase manner can balance the snapshot processing in time, and flatten the overhead of snapshot.

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. Node i+k will start the snapshot some Δt time after i^{th} node. With the deferred snapshot approach nodes with higher IDs will be required to 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.

Periodic window-log compaction. As a retrospective snapshot extends far in the past, the window-log increases in size. This can make the window-log compaction phase the overwhelming factor in determining the latency of snapshot completion, as shown in Figure 8.

One way to reduce the cost of window-log compaction phase is to perform periodic window-log compactions in the background. This way, when a snapshot is ordered, a compact window-log would be readily available. This also helps reduce the memory-usage of the window-log and can extend the reach of retrospective snapshots. A drawback of this approach is that it restricts the granularity of where a Retroscope snapshot can travel in time to that of the compaction period. This tradeoff makes sense when improving the latency of the snapshot is more important than the precision of target snapshot time, and we can pay a small fraction of background work for periodic window-log compaction.

Speculative snapshots. As Figure 8 shows, a rolling snapshot performs less work compared to a full snapshot as it skips the data copying phase, relying on a reference base snapshot instead. This introduces the opportunity to take occasional speculative snapshots so that when a snapshot is actually needed we may have a nearby reference base 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. A good prediction for when a snapshot will be required improves the benefit we get from speculative snapshots. Using historical data or identifying certain triggering conditions for a snapshot can boost the hit rate of a speculative snapshot. That being said, a mispredicted speculative snapshot can also find some use as a backup.

VIII. Related Work

Berkeley DB Retrospection. Retro allows past state querying and inspection against a Berkeley DB [21]. 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 [22], the authors develop an eidetic system by modifying Linux kernel to record all nondeterministic data that enters a process: the order, return values, and memory addresses modified by a system call, the timing and values of received signals, and the results of querying the 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 returned data differs from the model. That is, the system only saves nondeterministic choices or new input and can recompute everything else. The results in [22] are for single CPU machines and do not account for issues in distributed systems.

Freeze-frame file system. The Freeze-Frame File System (FFFS) [23] uses HLC [10] to implement retrospective querying on the HDFS file system [24]. FFFS uses multiple logs to capture data changes on HDFS NameNode and DataNodes and such logs are meant to persist to a low-latency storage, such as an SSD. 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 [25]–[30] 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 [28] 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. Pivot tracing employs a nice SQL-like querying interface for monitoring logs. In future work, we plan to use a similar interface to facilitate system operators to query distributed snapshots.

Conflict handling. Last write wins rule causes problem for distributed key-value stores that rely on NTP timestamping, like Cassandra and Riak [31], [32]. 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. Conflict-free Replicated Data Types (CRDT) [33] provide eventually consistent commutative data structures that can help deal with partitions.

IX. 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 full snapshot to reduce the cost of constructing new snapshots in that proximity. We demonstrated implementations of Retroscope for Voldemort and Hazelcast datastores, and evaluated their performance under different workloads.

An important use case for Retroscope is for 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 reset/revert operation. Since a Retroscope snapshot is globally consistent, it can be used for performing a consistent reset for the entire system. After identifying a suitable clean snapshot, in order to complete the reset for Voldemort, the database needs to be closed, the BDB files copied from the snapshot location into the environment location, and the database reopened. Most of the time in this operation is spend in copying the files, which for our 1 GB test database takes $\sim$8 seconds. In future work, we aim to provide programmatic toolkit support for snapshot evaluation and distributed reset.

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] "Linkedin," https://www.linkedin.com/.

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

[15] "Hazelcast, in-memory data grid," http://www.hazelcast.com, 2015.

[16] "Aws: Amazon web services," http://aws.amazon.com, accessed May 10, 2016.

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

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

[19] "Oracle berkeley db, java edition. getting started with transaction processing," https://docs.oracle.com/cd/E17277_02/html/TransactionGettingStarted/BerkeleyDB-JE-Txn.pdf.

[20] Amazon Inc., "Elastic Compute Cloud," Nov. 2008.

[21] M. A. Olson, K. Bostic, and M. I. Seltzer, "Berkeley db." in *USENIX Annual Technical Conference, FREENIX Track*, 1999, pp. 183–191.

[22] 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.

[23] W. Song, T. Gkountouvas, K. Birman, Q. Chen, and Z. Xiao, "The freeze-frame file system."

[24] 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.

[25] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat, "Mace: language support for building distributed systems," in *ACM SIGPLAN Notices*, vol. 42, no. 6. ACM, 2007, pp. 179–188.

[26] 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.

[27] 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

[28] 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. [Online]. Available: http://sigops.org/sosp/sosp15/current/2015-Monterey/printable/122-mace.pdf

[29] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the unexpected in distributed systems." in *NSDI*, vol. 6, 2006, pp. 115–128.

[30] Y.-Y. M. Chen, A. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer, "Path-based failure and evolution management," Ph.D. dissertation, University of California, Berkeley, 2004.

[31] 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.

[32] R. Klophaus, "Riak core: building distributed applications without shared state," in *ACM SIGPLAN Commercial Users of Functional Programming*. ACM, 2010, p. 14.

[33] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," *Stabilization, Safety, and Security of Distributed Systems*, pp. 386–400, 2011.