Retroscope: Retrospective Monitoring of Distributed Systems

Aleksey Charapko, Ailidani Ailijiang, Murat Demirbas, and Sandeep Kulkarni

Abstract—Retroscope is a comprehensive lightweight distributed monitoring tool that enables users to query and reconstruct past consistent global states of the system. Retroscope achieves this by augmenting the system with Hybrid Logical Clocks (HLC) and by streaming HLC-stamped event logs for storage and processing; these HLC timestamps are then used for constructing global (or nonlocal) snapshots upon request. Retroscope provides a rich querying language (RQL) to facilitate searching for global predicates across past consistent states. The search is performed by advancing through global states in small incremental steps, greatly reducing the amount of computation needed to construct consistent states. The Retroscope search algorithm is embarrassingly-parallel and can employ many worker processes (each processing up to 150,000 consistent snapshots per second) to handle a single query. We evaluate Retroscope's monitoring capabilities in two case studies: Chord and Apache ZooKeeper.

Index Terms—distributed debugging, distributed applications, tracing, query languages.

1 Introduction

Logging system state, messages, and assertions is a common approach to provide auditability. Logs can be used to identify performance bottlenecks, diagnose various issues, and even perform a post-mortem analysis of the system in case of a catastrophic failure. Analyzing the logs involves looking through a series of events leading to a problem, making it important to keep all events in the proper causal order. A log that reverses the order of causally related events is misleading and may suggest a wrong conclusion regarding the problem at hand. Moreover, it is also crucial to have events logged with affinity to real physical time: without the timestamp, it becomes impossible to pinpoint where within the log events of interest occur.

Naive logging-based approaches, however, 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* or cut [1], where no event in the cut happened-before any other event. This is important since 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 supports only scheduled, planned snapshots. One way to achieve *retrospective snapshots* of past states is to use vector clocks (VCs) [3], [4], [5]. But, this requires VCs with size $\Theta(n)$, the number of nodes, to be included in each message. 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 [6]. Logical clocks (LCs) [7] can be considered for reducing the cost of VC. However, taking a retrospective snapshot with LCs fails again because LCs cannot identify consistent snapshots/cuts with affinity to a given past physical time.

To get snapshots with sufficient affinity to physical time, one can potentially utilize NTP [8]. 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 [9], [10] (e.g., in Figure 1, P1 needs to block sending a message during the clock uncertainty period), which makes it undesirable for retrospective snapshots.

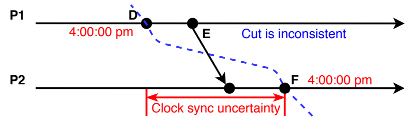


Fig. 1: Using NTP only fails to take consistent snapshot

Retroscope. To address this problem, we leverage our recent work on hybrid logical clocks (HLC) [11]. 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 is guaranteed to be a consistent cut.

Using this observation, we design and develop *Retroscope*, a highly-scalable solution for reconstructing and querying past consistent distributed snapshots by collating node-level independent snapshots. A *devops* team can use Retroscope to monitor application states for nonlocal predicates, investigate problems by stepwise debugging, perform root-cause analysis, data-integrity monitoring, and checkpoint-recovery. To achieve this, Retroscope exposes a query interface, allowing users to easily interact with a progression of globally consistent states. Figure 2 illustrates a user-centric perspective of Retroscope.

Contributions.

1) We introduce Retroscope, a toolkit that provides a novel "cut monitoring" approach for monitoring large scale distributed systems in a lightweight and scalable manner. Cut monitoring tracks global distributed states by identifying consistent cuts over streaming logs of state changes. This makes Retroscope useful for invariant-based reasoning [12], [13], [14] and invariant checking via global predicate detection. Cut monitoring approach is complementary to the request tracing based monitoring solutions [15],

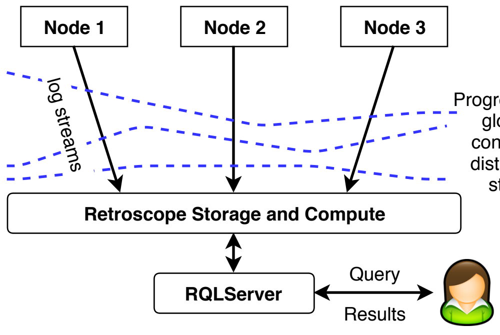

Node 1
Node 2
Node 3
log streams
Progression of globally consistent distributed states
Retroscope Storage and Compute
RQLServer
Query
Results

Fig. 2: Retroscope overview from user perspective

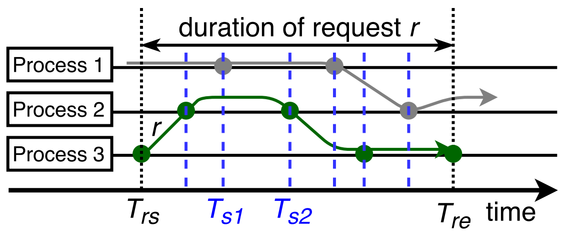


Fig. 3: Difference between monitoring with Retroscope and request tracing methods

[16], [17], [18]. Figure 3 illustrates this difference. Request tracing for a request r monitors the path of r on the machines between time T_{rs} and T_{re}. Retroscope, however, looks at a single snapshot at a time, such as T_{s1} or T_{s2}, covering the global state of the system and not just the nodes visited by the request. While request tracing is suitable for performance monitoring and tracing individual requests as it travels along multiple nodes, Retroscope's cut monitoring approach is useful for providing across-node context to diagnose race conditions, nonlocal state inconsistencies, and nonlocal invariant violations.

2) To provide effective, flexible, and extensive querying and compute capabilities in Retroscope, we introduce Retroscope Query Language (RQL). RQL allows to perform computations against nonlocal state, search for nonlocal predicates, filter unwanted variables or nodes out and restrict the search time intervals. RQL shows linear query performance scalability with over 150,000 consistent cuts processed by each worker node in a second.

3) We have implemented Retroscope in under 22,000 lines of Java code. It is available on GitHub as an opensource project [19]. Retroscope comprises of several components that contribute to scalability and performance of RQL. The *RetroLog* component is integrated at every node of the target end-system, and provides HLC instrumentation and logging to maintain a history of recent events at that node. The RetroLog streams the node's log to *Apache Ignite* for storage and future processing. *RQLServer* handles RQL queries received from *RQLClient*. The RQLServer also schedules *worker nodes* to perform the search over a progression of past system-states. The worker nodes collate HLC-stamped logs into snapshots and employ rolling snapshots to quickly move through the state-history and perform predicate evaluation on consistent states.

4) We showcase RQL based monitoring with case studies on Apache ZooKeeper [20] and Chord [21] monitoring. We use Retroscope and RQL to study the data staleness of replica nodes in ZooKeeper cluster. Staleness is a nonlocal property that cannot be easily observed by other monitoring techniques. We find that even under a normally operating cluster, it is possible to have replicas lagging by as much as 22 versions behind the rest of the cluster. This staleness may have detrimental effects on client applications relying on ZooKeeper. For instance, an application using ZooKeeper for synchronization may experience some performance degradation, since staleness elongates the overall time required to achieve synchronization between application servers.

Outline of the rest of the paper. We describe HLC timestamping next, followed by HLC snapshots in Section 3. In Section 4, we describe Retroscope, its components and the query language. Section 5 showcases RQL querying for monitoring ZooKeeper and Chord, followed by Retroscope performance evaluation in Section 6. We discuss current limitations and future extensions in Section 7. We review related work before our concluding remarks.

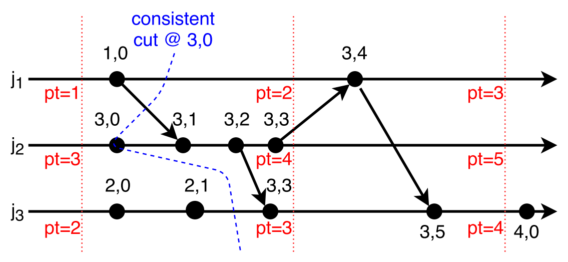


Fig. 4: Example of HLC operation with $\epsilon = 2$ on 3 process. Dashed lines denote the physical clock ticks with timestamp for each process next to it. HLC time is written above each event in the "l,c" format

2 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 [7].[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 events, it is unlikely to find events at each node with the same LC values where all are within a given physical clock affinity. In contrast, 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. Moreover, since HLC [11] 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 4.

HLC implementation. Figure 4 illustrates HLC timestamping. 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

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. Consistent cuts do not work with NTP, because 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, $pt.f$.

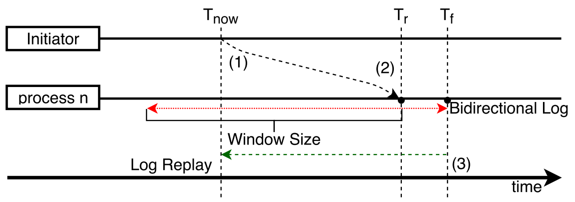


Fig. 5: Instant distributed snapshot

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 [22].

HLC can fit $l.j$ and $c.j$ in 64 bits in a manner backwards compatible with the NTP clock format [8] 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. The CockroachDB [23] adopted HLC and provides an implementation of HLC in Go. We provide an implementation of HLC in Java as part of our Retroscope framework.

3 HLC Snapshots

The ability to efficiently take consistent snapshot enables our cut monitoring approach. HLC allows us to quickly and efficiently identify consistent cuts from independent process states. However, to allow for retrospective examination of states, we also need to record state mutations at each process.

To that order, an HLC-based snapshot system can keep a log of recent state changes for every node. This log can have a limited capacity, depending on the desired depth of retrospection, and may purge all data in a sliding window manner. Every state change written to the log is accompanied with an HLC timestamp. By ensuring that all nodes roll back their states to the same HLC time, we acquire 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.

3.1 Snapshot Models

Point snapshots. Figure 5 depicts our distributed snapshot system with logs for each node/process. To acquire a point snapshot, capturing a single consistent cut of the recent systems state, we can use a current state of each process and the window-log to undo the changes from the current state until arriving to the state at desired HLC time T_{snap}.

It is not necessary to block the entire application from changing its current state while making a copy of a it and unrolling backwards to T_{snap}. Instead, the state may be copied at lower granularity, one variable at a time, as long as these changes continue to record in the log. For instance, at time T_r process n received a command to take a snapshot, it then began to copy its state and finished at time T_f. The copy of the state we have at T_f may not be consistent, however, the history of recent event changes allows us to correct the problems by undoing all operations that were recorded in current state snapshot after T_r. At this point we can continue undoing changes until reaching the process state at T_{snap}. Figure 5 uses a green-dashed arrow to illustrate the backward application of the window-log until reaching T_{snap}. When the state is rolled back to T_{snap} for all process, we have finished the snapshot operation and have system's consistent state for that time.

Incremental snapshots. Taking multiple point 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, we can leverage the window-log to obtain an incremental snapshot from an older base snapshot. Unlike a point snapshot, incremental one does not store the entire state, and stores only the changes from the base point. Figure 8 illustrates taking an incremental snapshot to arrive to a time T_p using a snapshot at time T_{base} as the base point. In order to get a snapshot at T_p, it is unnecessary to traverse the entire log backwards from the current state, and instead the system can just redo the changes captured in the log between T_{base} and T_p, 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. Retroscope's cut monitoring requires a snapshot that can quickly move through the states of a distributed system. We extend the incremental snapshots into rolling snapshots by providing the ability to progress from one state to the next without preserving the prior snapshot. This reduces the processing time and storage requirements. In rolling mode, once a snapshot has been examined, an incremental change is applied directly to it to move to the next examination point. Each incremental step, therefore, destroys the previously examined snapshot.

4 Retroscope

We have implemented Retroscope prototype in Java in a little over than 22,000 lines of code. Retroscope uses JFlex [24] and jacc [25] for parsing RQL and constructing the abstract syntax tree (AST) of the query. The generated parser accounts for roughly a third of Retroscope's source code.

We designed our system around Apache Ignite to provide high performance and scalability. Figure 6 provides a high-level overview of the system. Conceptually, Retroscope is separated into two separate parts: monitored system and distributed processing. At the monitored system, a component called RetroLog integrates into each node of the application and provides HLC and state logging services. Distributed processing is a larger part of Retroscope, and it is completely decoupled from the monitored application. It stores all the state transitions and handles predicate searches.

RetroLog component integrates into the monitored application to log application's state and stream it over to the streaming and storage components on the processing side. When clients need to perform a search across past application states, they issue a query through the RQLClient component. RQLClient sends the query to the RQLServer which processes the query, and distributes the

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

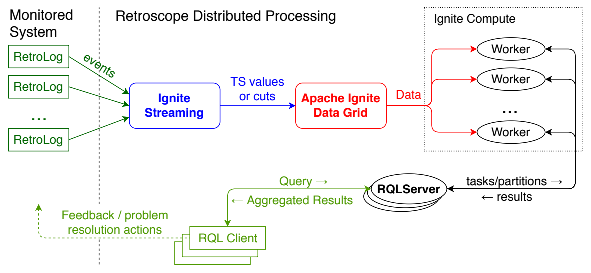


Fig. 6: Retroscope architecture on top of Apache Ignite

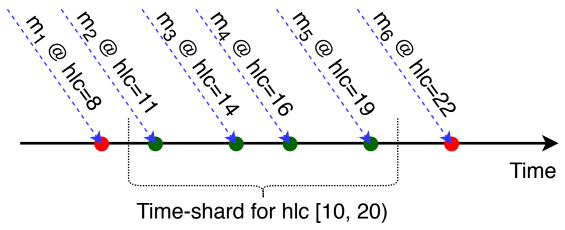


Fig. 7: Example of time-sharding. All events get assigned to a time-shard based on their HLC time

predicate search across many stateless workers. Workers retrieve the assigned sections of application state-transition log from storage and perform the search. RQLServer also carries out the final aggregation of results from the worker nodes and sends the results back to the RQLClient. The client at this point is responsible for handling the results of the query, such as outputting results in proper format or performing some automation.

Since Retroscope operates both inside the monitored system for data collection and HLC instrumentation and outside of the monitored system for storing and processing distributed states, it is subjected to the failures of both the monitored system and its own storage and processing components. Similar to normal logging tools, Retroscope can persist logs to the local disk up to the point of failure. These logs can be uploaded manually or streamed back to the Retroscope upon recovery for aggregation and further use in post-mortem investigations. Streaming component failures are handled similarly by restreaming recent logged state changes from the RetroLogs. The data at rest is replicated with configurable level of replication for redundancy and better reliability.

The combination of different components and different failure modes and assumptions, some of which are outside of our control and depend on the monitored application, makes Retroscope a best-effort-system. We try to preserve all captured data and recover from the failures to the best of our abilities, but some failures, especially on the monitored system may result in the data-loss, while other failures may result in late convergence as logs may be added to the system after some delay.

4.1 RetroLog Server

RetroLog servers handle two crucial tasks at the target system: establishing common HLC time and collecting data. RetroLog library provides an HLC API to the target application including `timeTick()` and `timeTick(remoteHLC)` methods. These methods used to advance time when sending and receiving remote messages respectively.

RetroLog also collects and records local events similar to traditional logging tools, such as log4j [26], except that each logged event is paired with an HLC timestamp instead of physical time. Users need to instrument the target system in order for RetroLog to collect the data: the instrumentation defines the variables and parameters recorded. RetroLog allows operators to log their data in 4 distinct ways: individual variables, structures, lists and sets. Each of the four formats can hold numeric or textual data. We represent the logged data in JSON-like format, since it is human-readable and simplifies querying the logged data by allowing queries to use the same format.

Consider an example of counting the number of active connections served by a node. A user may want track changes to a counter variable capturing the number of active connections. To do that, the user should add the following logging statement to all places in the code where connection counter is changed: `log.setVar("numClients", numClients);`. This will trigger RetroLog serer to log the new value for the "numClient" variable, along with the HLC timestamp and node identification. The user can also create a more complicated record with additional information about the connections by utilizing a structure instead of the variable: `log.setRecord("numClient").setVar("n", numClients).setVar("client", clientIP);`. Here the user records the IP of last connected client along with the counter. If the connection monitoring task requires to track all clients, the user may use a list or a set for logging the client connections: `log.getRSet("clients").addVal(clientIP);`

RetroLog streams all events to an Apache Ignite cluster acting as a storage and compute back-bone for Retroscope.

4.2 Ignite Streaming

In order to minimize the overhead of RetroLog on the target system, we aim to keep its memory and CPU footprint as small as possible by having only a minimal buffer of logged events in memory and streaming them for further storage and querying to the Retroscope processing components. The data ingestion is supported by Ignite Streaming API of the Apache Ignite.

Each streamed event is assigned to a time-shard, which is a collection of all events in the log from all nodes in some small time interval, as shown in Figure 7. Time-shard assignment determines the Ignite node responsible for ingesting the event, and all events from all RetroLogs belonging to the same time-shard arrive to the same node.

Making one node responsible for ingesting all events in the time-shard can put stress on a single node in the cluster if all RetroLogs send the events for a time-shard at roughly the same time. We compensate this in two ways: introducing random-size buffers to spread the time in which events of the same time-shard are streamed, and keeping the configurable duration of a single time-shard small to quickly rotate through the nodes.

Alternatively, a more mature streaming or publish-subscribe system, like Apache Kafka, may be used for initial data ingestion in large deployments. The overall architecture will not change drastically with the addition of a dedicated pub-sub component. RetroLogs will be publishing their events to Kafka topics, and aggregators will subscribe to these topics and construct time-shards for storage and further use.

Each time-shard aggregates the events with identical HLC timestamps into incremental snapshots. These snapshots represent the change in a global state occurring at such time [27]. Retroscope uses these snapshots later for quick navigation across

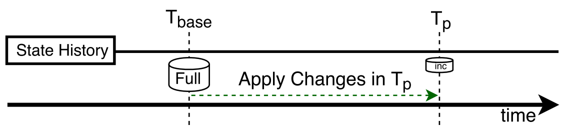


Fig. 8: Incremental snapshot applied to the prior reference point

system's state history. However, since the incremental snapshots only record change in state and not the full state, they need a reference point to be useful, as depicted in Figure 8. In order to arrive to the full snapshots at some time T_p, we need to have a full snapshot at T_{base} and an incremental snapshot at T_p with all changes between T_{base} and T_p.

For obtaining a reference point, we require each node of the monitored system to submit a local snapshot at the beginning of each time-shard. Streaming system accumulates these local snapshots together to produce a single global snapshot. This mode works well when monitoring a small set of variables or system parameters, however, when the state of monitored parameters is large, we can reduce the frequency of taking local snapshots. For instance, applications with large monitoring state may submit one local snapshot for every hundred time-shards.

4.3 Ignite Storage

After a time-shard is compiled at the streamer node, the shard is stored within Apache Ignite cluster. The system replicates the time-shard to a configurable number of nodes for fault tolerance and redundancy. The placement of each time-shard in the cluster is determined by the hash of a starting timestamp of the shard. Ignite enables us to scale storage by simply adding more servers to the cluster and adjusting the mapping of hashes to server. In addition to using Ignite storage for time-shards, we also store Retroscope metadata, such as a list of active nodes and time-shard duration.

4.4 Retroscope Query Language

Users interact with Retroscope through the Retroscope Query Language (RQL). RQL was inspired by both SQL and a model checking and specification language TLA+ [12]. The main construct of the language is a query. A query defines all filtering, computing and searching tasks that must be performed against a progression of globally consistent states. Each query consists of a few mandatory and optional parts, and follows the pattern:

```
SELECT variables
FROM logName
[COMPUTE expression]
[WHEN predicate]
[ON NODES idList]
[AT TIME expression TO expression]
```

`SELECT` clause is a required list of variables a user wishes to use in the query and see in the query output. These variables may either exist in the monitored state of the application or created later in the query.

`FROM` clause specifies the Retroscope log from which the system is going to pull the data.

`COMPUTE` statement allows the users to specify any computations that must be performed on each examined cut. These computations can define new variables that can be emitted back or utilized later in the query for filtering. The computation takes a form of an RQL expression.

`WHEN` condition is the main filter that decides which consistent cuts will be emitted to the user. The condition used with `WHEN` is an RQL predicate and must evaluate to either `true` or `false`. Depending on the result of predicate evaluation, system either returns the cut or discards it.

Consider an example RQL query in which a user investigates some nodes processing more client connections (denoted by the variable `numClients`) than the other nodes:

```
SELECT numClients, diff
FROM app
COMPUTE GLOBAL diff
  AND diff := Max(numClients) - Min(numClients)
WHEN EXISTS c IN numClients : (c > 10)
```

This query returns two variables in each cut: `numClients` and `diff`. It uses the log called `app` to get the data. Variable `diff` does not exist in the log and instead it is defined in the `COMPUTE` clause as the difference between the highest and lowest `numClients` observed in the cut. The `WHEN` clause of the query introduces a condition that must be satisfied for the system to emit a consistent cut. In the example, only consistent cuts in which at least one node serving more than 100 clients will be emitted to the user. Similarly, `FORALL` can be used instead of `EXISTS` to emit cuts that have all nodes serve more than 100 clients.

Some additional query commands allow users to restrict the query to some specific time-interval and specific machines in the cluster. This enables engineers to closely look at only the nodes experiencing problems without taking into account healthy parts of the application. For instance, this query performs the client connection search only on specified nodes and at a given time:

```
SELECT numClients
FROM app
WHEN FORALL c IN numClients : (c > 100)
ON NODES 1,2,3
AT TIME 100 TO 200
```

In case of `FORALL` predicate, it will be satisfied if the condition holds at all nodes in a given query, even if there are more nodes in the system.

4.5 RQLServer and RQLClient

RQLServer is a component responsible for parsing Retroscope Query Language (RQL) queries, creating query execution plan, verifying all the data in the Ignite Data Grid, and assigning tasks to the Retroscope workers according to the query execution plan. RQLServers do not perform any actual query computations and can exist outside of the Apache Ignite cluster, as long as a connection to the cluster can be established.

As part of query planning and scheduling, RQLServer breaks down the query into a number of time-slices that can execute on separate worker nodes. The time-slices are designed to have all information about the slice's starting state and state changes, allowing the entire query evaluation to be separated into independent tasks and performed in an embarrassingly-parallel manner.

By default, each time-slice corresponds to the time-shard created during data ingestion, however, if a system does not capture a full snapshot at every time-shard, then a time-slice assigned to a worker can span multiple consecutive time-shards. The collection of all tasks assigned to all workers represents the entire search space covered by the query.

The RQLServer tries to allocate worker nodes to be collocated with the nodes storing time-shards to minimize the data movement

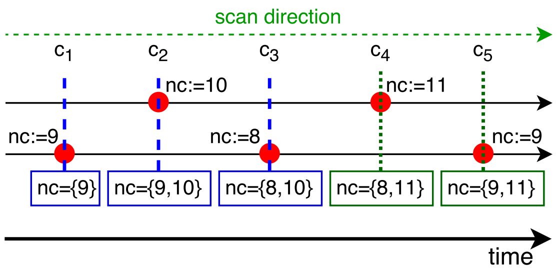


Fig. 9: Evaluating "*SELECT nc FROM app WHEN EXISTS c IN nc : (c >10)*" on a time-shard starting at cut c_1. Only cuts c_4 and c_5 are emitted. Search starts at cut c_1 and snapshots advance from least to most recent

between the machines. In addition to time-shards, each worker also receives predicates to be evaluated on every consistent cut. After receiving all required data, the worker nodes scan the cuts and report the ones satisfying the predicate back to the RQLServer.

Users do not interact with the RQLServer directly, instead they must use an RQLClient software to connect and submit their queries. The simplest example of RQLClient is a console application capable of taking user input and displaying the results of query execution. A more sophisticated client application would be an automation tool that monitors some system parameters and provides feedback or attempts to fix any detected problems at the target system.

4.6 Worker Nodes and RQL Interpreter

Upon receiving compute tasks from the RQLServer, each worker node starts up an RQL interpreter that is capable of executing RQL expressions against a progression of distributed states. The interpreter runs a virtual machine that searches through the history of past-systems states, as shown in Algorithm 1 . The virtual machine advances through the incremental snapshots in the time-shards assigned to the worker and updates its internal state with changes from the snapshot. It then and evaluates the query predicates to decide whether the cut must to be emitted or not. This process is illustrated in Figure 9. When the cut is emitted, worker node sends it to the RQLServer that handles the query.

Algorithm 1 RQL Engine Query Evaluation

```
1: Retrieve timeShard
2: Set timeShard.iterator ← 0
3: while timeShard.iterator < $|timeShard.cuts|$ do
4:     currentSnapshot ← timeShard.cuts[timeShard.iterator]
5:     currentHLC ← HLC time of currentSnapshot
6:     evaluate condition on currentSnapshot
7:     if condition = TRUE then
8:         emit snapshot at currentHLC
       timeShard.iterator ← timeShard.iterator + 1
```

Algorithm 1 shows the search algorithm used by the Retroscope worker nodes, and Figure 9 illustrates a sample run of the algorithm. The worker starts by first fetching all time-shards assigned to it (only one shard by default). It then sequentially moves through the incremental snapshots, evaluating the predicates against the state represented by each snapshot. When the predicate evaluates `true`, the cut is emitted to the RQLServer that started the worker process.

When evaluating their time-shards, workers only examine consistent cuts in which the state of the variables selected in the query changes. Doing so allows Retroscope to solely focus on parameters specified by the operators and filter out everything else. Additionally, such filtering helps improve query performance, since the cuts get pruned before Retroscope performs any computations and predicate evaluations.

5 Monitoring with Retroscope

In this section, we showcase Retroscope monitoring with RQL queries by providing case studies on Apache ZooKeeper [20] and an implementation of Chord [21] distributed hash table algorithm. All experiments were carried out on Amazon EC2 t2.small instances, with Retroscope distributed processing stack deployed over 4 nodes. The modest capacity of Retroscope virtual machines in this experiment does not affect the case studies, as it mostly impacts the in-memory storage capacity and query processing speed, and not the query result.

5.1 ZooKeeper Staleness

ZooKeeper is a popular coordination service at the heart of many large distributed applications. For improving throughput, ZooKeeper allows read operations from any replica (rather than restricting them to only read from the master). Thus, a client may read stale/outdated value from a replica, especially if the cluster is under high load or the replica is a straggler. Unlike other monitoring approaches, Retroscope's cut monitoring solution enables us to measure the data staleness in ZooKeeper accurately by observing the differences in znode versions between replicas at the consistent global states. To the best of our knowledge, no previous work investigated ZooKeeper replica staleness.

To investigate replica staleness, we deployed a 5-node ZooKeeper cluster on Amazon EC2 t2.small nodes. We added a RetroLog component to ZooKeeper replicas to instrument them with HLC; this modification was minimal and required less than 30 lines of code. A simple client performed update operations on ZooKeeper znodes or keys. We added instrumentation to ZooKeeper's final-request processor class to keep track of znode versions right before the value is applied to the internal data store. Adding Retroscope instrumentation required putting a single line of code at each instrumentation point. We performed our evaluation on a healthy ZooKeeper cluster, where all replicas have roughly the same performance.

We used a workload of 15,000 non-interleaving update operations to a single znode to test the data staleness of ZooKeeper replicas. Our workload targeted only a single znode and had a single ongoing ZooKeeper operation at any given time. After running the workload, we used RQL to obtain consistent cuts to observe how znode data was changing across the replicas. The below query outputs the cuts with staleness of 2 or more versions. In the query, $r1$ is the name of ZooKeeper znode we monitor, and $zklog$ is the log keeping the history of all update operations at the node.

```
SELECT r1 FROM zklog WHEN Max(r1)−Min(r1)>1;
```

We have observed that a *healthy* ZooKeeper cluster can sometimes have stale replicas. In our experimental runs we noticed some replicas getting as far as 22 versions behind other nodes even under a low intensity workload. ZooKeeper provides a *sync* operation to allow clients receive up-to-date version of a znode, however, we found that issuing *sync* commands have no impact on server staleness. Instead *sync* acts as a non-mutating update operation and forces a client to block and wait for the value

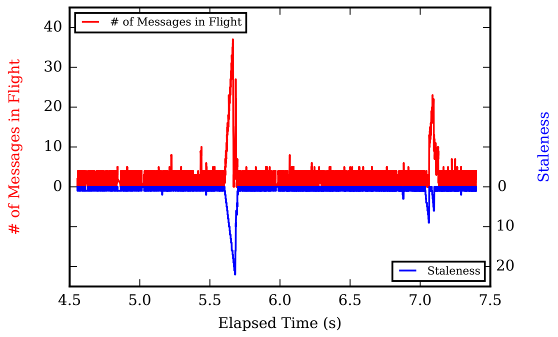


Fig. 10: ZooKeeper staleness and # of messages in transit as measured by Retroscope

following the *sync* command to preserve the "read-your-write" guarantee of ZooKeeper [28].

To explain the high staleness issue, we looked at the message flow in the system and searched for anomalies in the communication between the nodes during the intervals of high staleness. For that, we added some instrumentation to ZooKeeper to keep the counters of messages sent and received by each node. This allowed us to write a query to examine the number of messages in transit between along with the staleness metric.

```
SELECT
   r1 , sentCount , recvCount , diff , staleness
FROM zklog
COMPUTE GLOBAL diff
   AND GLOBAL staleness
   AND ( staleness := Max(r1) − Min(r1) )
   AND ( diff := NodeSum(sentCount)
     − NodeSum(recvCount))
AT TIME x TO y
```

Figure 10 illustrates the output of the above query for a roughly 3 second slice of the test workload execution. We observe a rapid increase of messages being in transit around the time staleness spikes up.

To gain further insight in how the messages are flowing we used a custom RQLClient intended to visualize the below query, where the set difference between sent and received messages are monitored.

```
SELECT sentM , recvM , inFlight , r1 , staleness
FROM zklog
COMPUTE GLOBAL staleness
   AND ( staleness := Max(r1) − Min(r1))
   AND GLOBAL inFlight
   AND ( inFlight :=
      Flatten(sentM) \ Flatten(recvM))
AT TIME x TO y
```

Figure 11 shows the amount of messages flowing between nodes. Higher intensity color designates a greater number of messages in the network for that node. This figure spans a 2-second interval around one of the staleness spikes in ZooKeeper. We can clearly observe the leader node (node #3) having more messages sent to it. However, there is an in-transit message spike at the leader and follower #4 right around the time of observed high staleness. [4]

We used the same query and RQL client to look at some consistent cuts right before the time of staleness spike and during

4. Node #1 appears to be at a constant slight disadvantage; this is because it serves the client generating the workload.

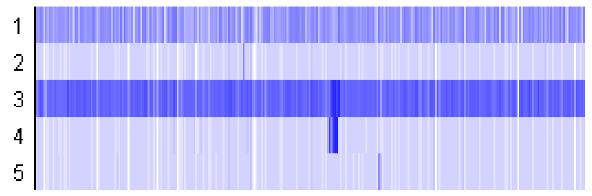
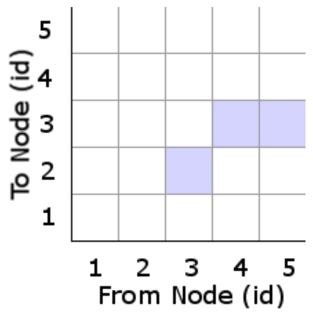
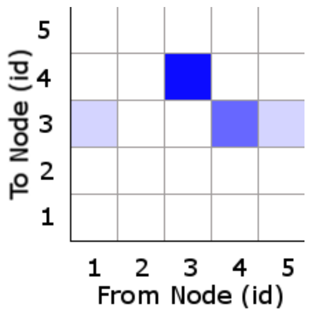


Fig. 11: Amount of inbound within a 2-second interval around the staleness spike. Higher color intensity represents larger amount of inbound messages



(a) Message flow at some cut before staleness spike



(b) Message flow at cut with max staleness

Fig. 12: In-flight messages before and at observed staleness. Higher color intensity represents larger amount of inbound messages

that spike. As seen in Figure 12, not many messages were in the network between nodes right before the staleness spike, however as the staleness developed, we saw a bidirectional increase in the number of messages being in-transit between the leader and node #4. We also did not observe any node except #4 experience the staleness at that time. These findings point to a momentary network performance degradation or a millibottleneck [29] between the leader and node #4.

In addition to the experiments on healthy ZooKeeper cluster, we conducted a study of the cluster with a straggler node. We artificially throttled down one of the follower replicas to process the incoming replication messages with a 2 ms delay, mimicking a node lacking in compute or network capacity compared to the rest of the cluster. For this experiment we used the same workload as before, except we performed updates to 10 znodes instead of one to reduce znode contention.

Figure 13 illustrates how the staleness of the struggler node changed over time in a 5-node ZooKeeper cluster. With this experiment we empirically show how a single straggler node, while not impacting the performance of ZooKeeper, affects the staleness of the cluster and consequently may negatively affect application relying on ZooKeeper.

It is curious that it took a couple of seconds for straggler to start lagging behind the rest of the cluster, but once the slow replica starts to fall behind, the effect snowballs quickly without any chance of recovery. The reason a straggler node was able to keep up with the rest of the cluster for the first few seconds is because of both JVM warm-up and ZooKeeper queue allocations. ZooKeeper processes replication commands in the queue, and initially these queues are allocated to hold only a handful of commands. As the workload ramps up, the queues reach the limit and must be reallocated, causing the healthy nodes to stutter and allowing the crippled straggler node to catch up. After a few seconds of runtime, ZooKeeper queues are allocated to handle

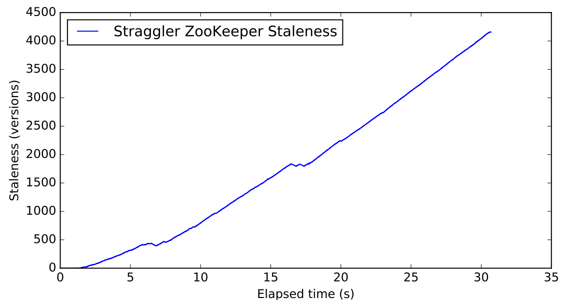


Fig. 13: Staleness progression of a 5-node ZooKeeper cluster with a straggling follower

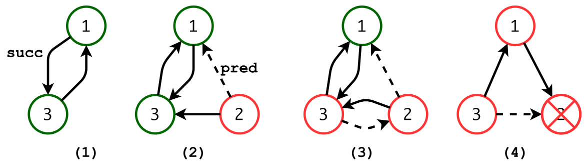


Fig. 14: Chord bug trace. Node 2 failed in middle of join operation

enough operations for the workload, allowing healthy nodes to proceed at full speed.

5.2 Chord Ring Membership

The Chord [21] protocol provides a scalable peer-to-peer distributed hash table that allows nodes joining and leaving the system. However, there exist known bugs [30] in the original description of Chord ring-membership algorithm, e.g. node failure during join operation may create gaps in the ring. We retroscoped a simple Chord implementation [31] and used RQL to monitor such violations by checking the ring condition in every consistent cut. Retroscope's cut monitoring approach allowed us to quickly check for ring correctness. Tracing solutions may often be inadequate for this task, as they lack the ability to see consistent global state of the system, even when all request traces are collected.

As shown in Figure 14, we start Chord with two nodes: node 1 and 3 have each other as their successors (variable *succ*). We then start node 2 that attempts to join the ring by connecting to a known node to find its correct successor. In this setup node 3 should become the successor of node 2, therefore node 2 should become the predecessor of node 3. Node 2 fails after step (3), and the periodic stabilization protocol sets node 1's successor as node 2's new predecessor. This is a violation of the invariant that each node's successor is correctly maintained.

We check the correctness of our ring with the following query:

```
SELECT succ, id
FROM log
WHEN succ[1]=id[2]
  AND succ[2]=id[3]
  AND succ[3]=id[1]
```

This query emits the cuts that have the correct 3-node configuration of our Chord ring, and in the problematic case, the results will be empty. Figure 14 shows a detailed trace of the consistent cuts during a faulty execution, provided by the following query:

```
SELECT succ, pred, id FROM log ON NODES 1,2,3
```

We can observe node 1 stabilization in step (4) after node 2's failure causes a gap in the ring.

6 Performance Evaluation

The performance of Retroscope query execution depends on many factors, such as the size of the Apache Ignite cluster, the amount logged events, the complexity of queries and how many cuts a query emits. In this section experimentally study the impact of these factors on performance.

6.1 Evaluation Setup

We conducted our performance evaluation of Retroscope on an AWS EC2 cluster. We deployed Retroscope over a cluster of m4.xlarge instance with 4 virtual cores and 16GB RAM. By default, we used 8 such nodes for our experiments, unless stated otherwise. We used a synthetic workload for this performance evaluation in order to have better control over the parameters influencing the results. Our synthetic workload was generated by an application distributed over 5 different EC2 m4.xlarge instances in the same AWS region.

The workload application consists of nodes communicating with each other. Each node maintains state expressed consisting of many counter variables. The state of every counter variable is tracked with Retroscope. Every node sends messages with the name and value of some counter to a randomly picked node at a controlled rate. Sending each message requires choosing the counter, incrementing its value, using RetroLog component to record it with Retroscope and finally transmitting it to an arbitrary node. Upon receiving the message, the node updates its version of the counter with the received value and records the counter to Retroscope. Occasionally a node resets the value of a counter before sending it to one of the peers. Despite being primitive, this application (available on Retroscope's GitHub page) allows controlling various aspects of systems behavior, such as number of messages exchanged per second, the size of tracked nonlocal state, and number of peers participating in the message exchange. In particular, we generated between 100,000 and 1,200,000 logged events over a 2 minute interval and varied the state size between 10 and 100 variables.

In our evaluation we measured two key performance metrics: query latency and query processing throughput. Query latency is the end-to-end execution time for a query. Query processing throughput is the rate at which Retroscope scans through the application states, measured in cuts per second.

6.2 Search Space Size

Retroscope must scan through all past consistent cuts in the query range to find the states satisfying the search criteria. In this experiment we ran workloads of different sizes to evaluate how well Retroscope scales with respect to an increased workload. For each workload size we repeatedly ran a set of different 10 queries of similar complexity (selecting 10 variables with conditional expression involving two variables) and measured the average query latency and query processing throughput.

As illustrated in Figure 15, Retroscope's query latency grows linearly as the search space increases. This predictable performance is also evident from query processing throughput, as the system moves through the cuts at roughly the same pace regardless of the workload size. This behavior is attributed to the linear nature of rolling through state history in the cut-monitoring systems.

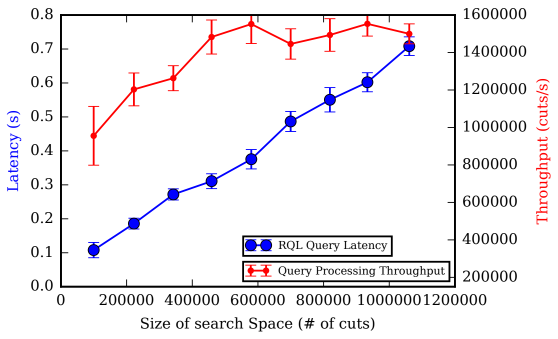


Fig. 15: RQL query performance as the search space increases

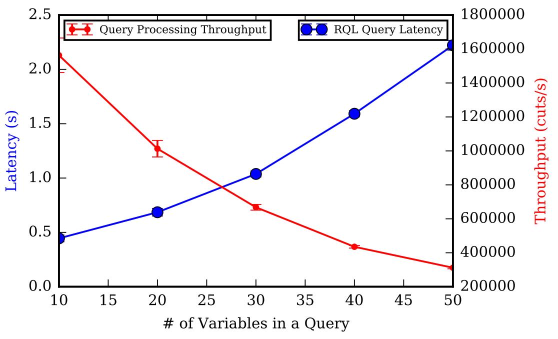


Fig. 17: RQL query performance at different query complexity

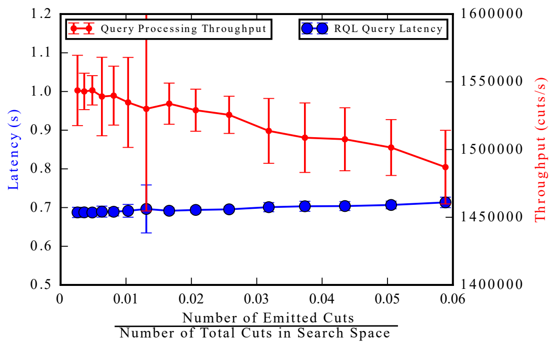


Fig. 16: RQL query performance at different ratios of emitted cuts to a total number of consistent cuts

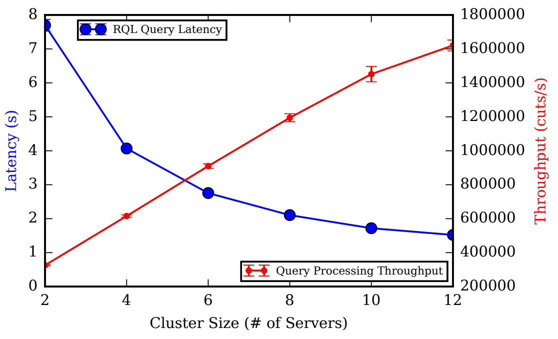


Fig. 18: Retroscope's Horizontal Scalability

6.3 Number of Emitted Cuts

Retroscope emits only the cuts that satisfy the query condition predicate. The number of cuts each worker emits has a direct impact on query performance, since these cuts must travel through the network and aggregate at RQLServer. Figure 16 illustrates the Retroscope's performance degradation as the ratio of emitted cuts to the number total cuts in the search space increases. In this experiment we evaluated our system by searching through a state history consisting of approximately one million cuts. We adjusted the workload and 10 queries to controllably return predictable number of results. We then ran the queries to measure performance for different number of returned cuts.

As the number of emitted cuts increases, the performance starts to degrade linearly, as the linearly more data needs to be passed from workers to RQLServer. Additional overhead occurs at the RQLServer when it sorts partially sorted blocks of emitted cuts coming from the workers.

6.4 Query Complexity

Retroscope can track many different variables of application state, however, each query does not need to use all of these variables. Evaluation procedure prunes all consistent cuts that do not mutate the variables specified by the user. This greatly reduces the amount of snapshots Retroscope evaluates. The system still needs to scan through all states in the query regardless, but evaluation is skipped if none of the query variables have changed. Thus, more complicated queries with more variables may run longer than the queries operating on less data.

Figure 17 illustrates the performance degradation as we increase the number of variables used in the query. In this experiment we ran the workload of 692,576 consistent cuts tracking 50 distinct variables. The queries in this experiment selected a subset of these variables. Expectedly, the performance of the system degenerated as we used more variables. Another reason for diminishing performance on large queries is the increased size of each returned cut that must be transmitted to RQLServer.

6.5 Horizontal Scalability

Retroscope's cut monitoring approach enables the system to distribute the workload to many independent workers, with each worker responsible for a small subset of state history to sift through. To evaluate the horizontal scalability of Retroscope, we used our artificial workload to generate around 2.5 million cuts with each cut containing one or more logged event. We then used Retroscope cluster of varying sizes between 2 and 12 instances to process 10 different search queries against that state history. Each of our queries targeted 50% of the variables in the tracked state and emitted roughly the same number of cuts. We repeated each query 10 times to obtain average execution latency and query processing throughput.

Figure 18 illustrates how well Retroscope scales horizontally by adding more nodes to the cluster. Our system demonstrated near perfect speedup when adding nodes to increase the available compute power. For instance, increasing the number of servers from 2 to 8 resulted in almost 3.7 throughput increase.

6.6 Impact on Monitored Systems

Retroscope is designed to minimize its impact on the performance of the monitored system, as the majority of Retroscope components responsible for streaming, storage, and query processing are deployed on the infrastructure independent from the monitored system. However, one possible point of impact is the RetroLog component that must be integrated in the target system. In [27] we have illustrated that adding HLC and recording the state of the node with RetroLog component has minimal impact on the overall performance of the system integrating RetroLog. Our current version of RetroLog used for monitoring, unlike the one in [27], has even smaller buffer for keeping state changes and streams the changes with Ignite Streaming, so as long as the network capacity is not saturated at the monitored nodes, the impact of RetroLog component will remain small.

7 Further Improvements

Our Retroscope implementation can be improved in many ways to extend its functionality and offer better performance.

Eliminating redundant calculations. Upon a query evaluation, a worker rolls from one snapshot to the next. During this process, the global state often changes slightly, by one or two variables. Despite the small changes at every new cut, RQL starts all computations from scratch and re-evaluates the expressions nested in the `COMPUTE` and `WHEN` predicates. However, parts of these expressions may not have changed from one cut to another. In the future versions of Retroscope we plan to take greater advantage of having small changes from one state to another and eliminate redundant computations.

State coverage. Retroscope relies on HLC to efficiently identify consistent cuts for predicate detection. HLC alone is incapable of finding all possible consistent cuts. However, if the predicate of interest is valid for sufficiently long time (2ϵ) or more or is recurrent, HLC is expected to detect it [32]. By contrast, if we want to remove all false negatives (without adding false positives), we need $O(n)$ sized clocks which adds a substantial overhead to the system (as these timestamps must be included in all messages). Thus, HLC provides a cost-effective approach for monitoring.

It is possible to integrate orthogonal solutions/extensions to Retroscope for dealing with false negatives. One approach is to utilize recently developed biased clocks [33]. Biased clocks provide an alternate implementation of HLC where the update timestamp for receive events is biased. Specifically, by adding a bias value to message timestamp received from another process, biased clocks allow us to reduce the false negatives substantially (to almost 0 in many cases). In [34], HLC and information about messages are used to detect violation of desired predicates with the help of SMT solver Z3. This eliminates false positives but has a higher overhead than using HLC alone. Also, we can eliminate false negatives by replacing HLC with a more capable causal clock, such as Hybrid Vector Clocks [35]. However, this also increases the cost as size of HVC can be $O(n)$ in the worst case.

Streaming queries. In some cases, it is beneficial to evaluate the global states of a distributed system on the fly as the events happen. Adding streaming queries can offer near real-time monitoring for global predicate violations and allow early problem detection to complement the post-mortem retrospection offered by normal queries. Streaming query capability can rely on the log streams coming from the RetroLog servers to continuously evaluate query on new consistent cuts as they become available.

Temporal queries. Retroscope focuses on highly parallel search through recorded states of the system one cut at a time. However, in some cases users are interested in temporal trends spanning multiple cuts. To support temporal querying natively, we plan to expand RQL with multivariate time-series data querying. A differential data-flow approach [36] may allow workers perform incremental computations more efficiently for temporal queries.

8 Related Work

8.1 Monitoring and Debugging

Various solutions to distributed systems monitoring and debugging have been proposed in the literature. We can distill the various approaches into three broad categories: loggers, tracers, and replayers. Table 1 generalizes some of the attributes of the systems in each category. The main differences between the tools across categories lie in the way they produce and present the results back to the users.

Loggers [26], [37] are the most basic, yet the most widely used debugging tools. They allow users to capture information about local parameters at the nodes of a distributed application. Most commonly, loggers such as log4j [26] allow to control amount details captured in the logs in the layered approach: different log layers exist with an increasing amount of data captured at each higher level, allowing system administrators to escalate logging level only when it is needed for debugging. Some logging systems, like log^2 [37] aim to automate the amount of details captured by using anomaly detection techniques to filter out any events that do not point to an abnormal behavior of the system. While Retroscope uses logging to record the data, traditional loggers are not equipped with tools to search and analyze logs. Retroscope also provides a mean to look at data across many machines, while loggers record per-machine data that needs to be collated together at some later time. Loggers often require orthogonal systems to process the logged data.

Tracers [15], [17], [18] rely on more advanced logging capabilities to keep track of request propagation and execution across multiple machines. Request tracing solutions allow for considerable more advanced performance monitoring and debugging capabilities than loggers due to their ability to breach a single node boundary. Pivot Tracing [15] is an example of a system that captures the information along the request path and propagates it forwards with the request. This allows the system to precisely identify request execution path and track causality between events in the same request. Pivot Tracing aggregates the data collected at every request of the same type and uses it to answer user queries. This contrasts with Retroscope that exposes consistent snapshots for analysis across the nodes in the cluster, and not individual or aggregated requests spanning over some time. Mystery Machine [18] is another tracing example. It does not use any specially constructed logs and instead relies on large amounts of regular logs being collected to deduce the most common request paths and identify performance bottlenecks along these paths.

The most complicated and comprehensive debugging tools are record-and-replay systems, or replayers. Replayers [38], [39], [40] are deterministic re-execution solutions that must record every input and nondeterministic operations performed at every node of the system during the runtime. They use this information to reproduce the execution of a system in a development or debugging environments. Retroscope is a simpler solution that does not reproduce the executions, instead in allows to trace consistent progression of states in some recorded execution without replaying or running the actual system under test.

8.2 Snapshots

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, inter-process communication, and network input.

TABLE 1: Comparison of monitoring and debugging methods

	Loggers	Tracers	Replayers
Usage	monitoring & debugging	performance monitoring	debugging
Performance overheads	low	medium	high
Ease of setup	easy	medium	medium
Requires post-processing	sometimes	✓	✓
Post-processing cost	low to medium	medium	high
Ease of use for debugging	hard	medium to hard	easy to medium
Ease of use for performance monitoring	hard	easy to medium	hard

In [41], the authors modified the 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 [41] are for single CPU machines and do not account for issues in distributed systems.

Freeze-frame file system. The Freeze-Frame File System (FFFS) [42] uses HLC [11] to implement retrospective querying on the HDFS file system [43]. 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 modified 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. Unlike Retroscope, FFFS does not provide any observability capabilities and strictly focuses on retrospective-snapshots of the databases. Additionally, FFFS is not general enough to provide an arbitrary past snapshot, as it requires a special marker to preserve snapshot data at particular HLC timestamp.

Snapshots in distributed event-sourced systems Erb et al. perform retrospective snapshots of distributed system by adopting and extending the actor model and event-sourcing [44]. In their system, each node keeps a log of events, such as data being sent and received along with the causality information associated with pairs of send-receive events captured as logical clock timestamps. They evaluate two algorithms capable of constructing globally consistent snapshots from such distributed logs. Dynamic Dependency Vector Reconstruction (DDVR) algorithm used as the baseline does not guarantee capturing snapshot all actors or nodes in the distributed system, resulting in causally-consistent, but partial snapshots. Second algorithm, Forward Causality Barrier (FCB) extends the baseline protocols and uses physical time to add causally unrelated actors to the snapshot. These algorithms capture snapshots that may not have appeared, or even had a chance to appear, in the real execution of a system. Retroscope, on the other hand, operates on tighter consistent snapshots across all nodes. Additionally, Retroscope provides tools to search or check for state violations across a large sequence of snapshots.

9 Concluding Remarks

We introduced Retroscope for performing lightweight, incremental, and retrospective monitoring of distributed snapshots. Retroscope leverages HLC timestamping to collate node-level independent snapshots for obtaining global consistent cuts, and avoids both the inconsistent cut problems associated with NTP timestamping, and the scalability problems of VC timestamping. We demonstrated the use of Retroscope for monitoring and debugging in ZooKeeper and Chord case studies. Our performance evaluation under different workloads showed good system scalability.

An important use case for Retroscope is continuous monitoring and recovery of data integrity against failures or attacks. A streaming RQL query can detect bad behavior and alert operators, upon which the operators can explore around the problematic time interval using Retroscope, and perform step-by-step debugging and root cause analysis. Another important use case for Retroscope is invariant checking via global predicate detection. In future work, we will add reset/revert capability to Retroscope to allow recovery of critical distributed states tracked in the system. Since a Retroscope snapshot is globally consistent, it can be used for performing a consistent reset for the entire system.

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," *Distributed Systems*, vol. 53, 1993.

[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] S. Yingchareonthawornchai, D. N. Nguyen, V. T. Valapil, S. S. Kulkarni, and M. Demirbas, "Precision, recall, and sensitivity of monitoring partially synchronous distributed systems," in *International Conference on Runtime Verification*. Springer, 2016, pp. 420–435.

[7] 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.

[8] J. Burbank, D. Mills, and W. Kasch, "Network time protocol version 4: Protocol and algorithms specification," *Network*, 2010.

[9] 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.

[10] 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.

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

[12] L. Lamport, *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[13] R.-J. Back, "Invariant based programming," in *International Conference on Application and Theory of Petri Nets*. Springer, 2006, pp. 1–18.

[14] E. W. Dijkstra, "Notes on structured programming." Technological University Eindhoven, 1970.

[15] 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.

[16] 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.
[17] 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
[18] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services," *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 217–231, 2014.
[19] "Project retroscope," https://github.com/acharapko/retroscope-lib, 2016.
[20] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *USENIX ATC*, vol. 10, 2010.
[21] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.
[22] Amazon Inc., "Elastic Compute Cloud," http://aws.amazon.com/ec2/, 2018.
[23] "CockroachDB: A scalable, transactional, geo-replicated data store," http://cockroachdb.org/, 2018.
[24] "Jflex - jflex the fast scanner generator for java," http://jflex.de, 2017.
[25] "Jacc: just another compiler compiler for java," http://web.cecs.pdx.edu/~mpj/jacc/, 2017.
[26] "Apache log4j," https://logging.apache.org/log4j/2.x/, 2017.
[27] A. Charapko, A. Ailijiang, M. Demirbas, and S. Kulkarni, "Retrospective lightweight distributed snapshots using loosely synchronized clocks," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 2061–2066.
[28] K. Lev-Ari, E. Bortnikov, I. Keidar, A. Shraer, E. Engineering, and M. View, "Modular Composition of Coordination Services," *2016 USENIX Annual Technical Conference*, pp. 251–264, 2016.
[29] C. Pu, J. Kimball, C.-A. Lai, T. Zhu, J. Li, J. Park, Q. Wang, D. Jayasinghe, P. Xiong, S. Malkowski *et al.*, "The millibottleneck theory of performance bugs, and its experimental verification," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 1919–1926.
[30] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak, "Crystalball: Predicting and preventing inconsistencies in deployed distributed systems." in *NSDI*, vol. 9, 2009, pp. 229–244.
[31] "Chord implementation in java," https://github.com/gabet1337/Chord, 2013.
[32] S. Yingchareonthawornchai, D. N. Nguyen, V. T. Valapil, S. S. Kulkarni, and M. Demirbas, "Precision, recall, and sensitivity of monitoring partially synchronous distributed systems," *CoRR*, vol. abs/1607.03369, 2016. [Online]. Available: http://arxiv.org/abs/1607.03369
[33] V. T. Valapil and S. S. Kulkarni, "Biased clocks: A novel approach to improve the ability to perform predicate detection with o(1) clocks in a week," in *SIROCCO 2018 (to appear)*, 2018.
[34] V. T. Valapil, S. Yingchareonthawornchai, S. S. Kulkarni, E. Torng, and M. Demirbas, "Monitoring partially synchronous distributed systems using SMT solvers," in *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*, ser. Lecture Notes in Computer Science, S. K. Lahiri and G. Reger, Eds., vol. 10548. Springer, 2017, pp. 277–293. [Online]. Available: https://doi.org/10.1007/978-3-319-67531-2_17
[35] M. Demirbas and S. Kulkarni, "Beyond truetime: Using augmentedtime for improving spanner," *LADIS '13: 7th Workshop on Large-Scale Distributed Systems and Middleware*, 2013.
[36] F. D. McSherry, R. Isaacs, M. A. Isard, and D. G. Murray, "Differential dataflow," Oct. 20 2015, uS Patent 9,165,035.
[37] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A Cost-Aware Logging Mechanism for Performance Diagnosis," *Atc*, pp. 139–150, 2015.
[38] D. Geels, G. Altekar, S. Shenker, and I. Stoica, "Replay debugging for distributed applications," *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pp. 289–300, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267359.1267386
[39] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica, "Friday: Global Comprehension for Distributed Replay Background: liblog Distributed Watchpoints and Break," *Design*, vol. 7, pp. 285–298, 2007.
[40] P. Wang and S. Lu, "RDE : Replay DEbugging for Diagnosing Production Site Failures," pp. 327–336, 2016.
[41] 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.
[42] 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.
[43] 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.
[44] B. Erb, D. Meißner, G. Habiger, J. Pietron, and F. Kargl, "Consistent retrospective snapshots in distributed event-sourced systems," in *Networked Systems (NetSys), 2017 International Conference on*. IEEE, 2017, pp. 1–8.

Aleksey Charapko received his M.S. degree in computer science from the University of North Florida in 2015. He is currently a Ph.D. candidate at the University at Buffalo. His research interests include distributed and cloud systems, distributed monitoring, distributed databases and fault tolerance.

Ailidani Ailijiang recently received Ph.D. (2018) from University at Buffalo, SUNY, in computer science & engineering with thesis focused on distributed consensus. He received B.E. (2006) in network engineering from Dalian University of Technology. His main research interests include distributed computing, fault tolerant and networked systems.

Murat Demirbas is a Professor of Computer Science & Engineering at University at Buffalo, SUNY. Murat received his Ph.D. from The Ohio State University in 2004 and did a postdoc at the Theory of Distributed Systems Group at MIT in 2005. His research interests are in distributed and networked systems and cloud computing. Murat received an NSF CAREER award in 2008, UB Exceptional Scholars Young Investigator Award in 2010, UB School of Engineering and Applied Sciences Senior Researcher of the Year Award in 2016. He maintains a popular blog on distributed systems at http://muratbuffalo.blogspot.com

Sandeep Kulkarni is a professor at the Department of Computer Science and Engineering. His interests lie in Operating Systems, Distributed Systems, Fault Tolerance. He is a member of the Software Engineering and Networks Laboratory. His research interests include Fault-tolerance, distributed systems, networks, reliability and software engineering.