SCALABLE COMPUTING SOFTWARE LABORATORY

ILLINOIS INSTITUTE OF TECHNOLOGY

ChronoLog
A Distributed Tiered Shared Log Store with Time-based Data Ordering

Anthony Kougkas
akougkas@iit.edu

CS492, Sept 26th, 2022

Agenda

- Setting the context
- ChronoLog
  - Design and architecture
  - Implementation details
    - Tail operations
    - Recording and replaying log events
  - Ramifications of physical time
- Experimental results
- Conclusions and future steps

The rise of activity data

- Activity data describe things that **happened** rather than things that **are**.
- Log data generation:
  - Human-generated: various types of sensors, IoT devices, web activity, mobile and edge computing, telescopes, enterprise digitization, etc.,
  - Computer-generated: system synchronization, fault tolerance replication techniques, system utilization monitoring, service call stack, error debugging, etc.,
- Low TCO of data storage ($0.02 per GB) has created a “store-all” mindset
- Today, the volume, velocity, and variety of activity data has exploded
  - e.g., SKA telescopes produce 7 TB/s

akougkas@iit.edu

Log workloads

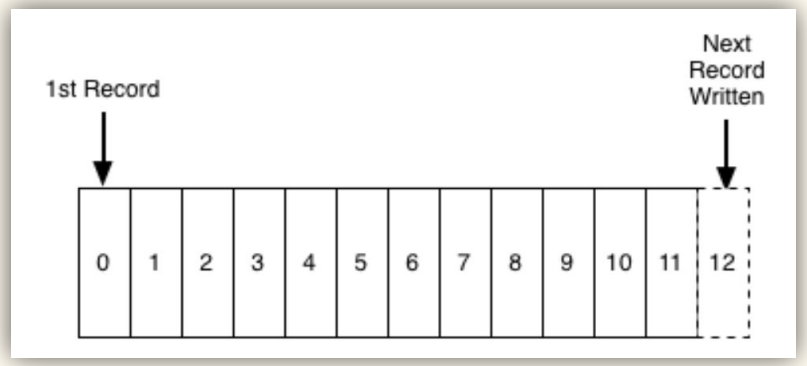
- Internet companies and Hyperscalers
  - Track user activity (e.g., logins, clicks, comments, search queries) for better recommendations, targeted advertisement, spam protection, and content relevance
- Financial applications (banking, high-frequency trading, etc.,)
  - Monitor financial activity (e.g., transactions, trades, etc.,) to provide real-time fraud protection
- Internet-of-Things (IoT) and Edge computing
  - Autonomous driving, smart devices, etc.,
- Scientific discovery
  - instruments, telescopes, high-res sensors, etc.,

Connecting two or more stages of a *data processing pipeline* without explicit control of the data flow while maintaining data durability is a common characteristic across activity data workloads.

akougkas@iit.edu

Log basics

- Simple storage abstraction
  - An append-only, totally-ordered sequence of immutable data entries (or events)
  - The ordering of events defines a notion of "time“ (leftmost entries are older than rightmost)
  - Content and format of data entries are often serialized in a binary representation
- Writes
  - Data are appended at the end of the log
- Reads
  - Proceed left-to-right in a linear scan fashion
- Not all that different from a file or a table.
  - A file is an array of bytes, a table is an array of records



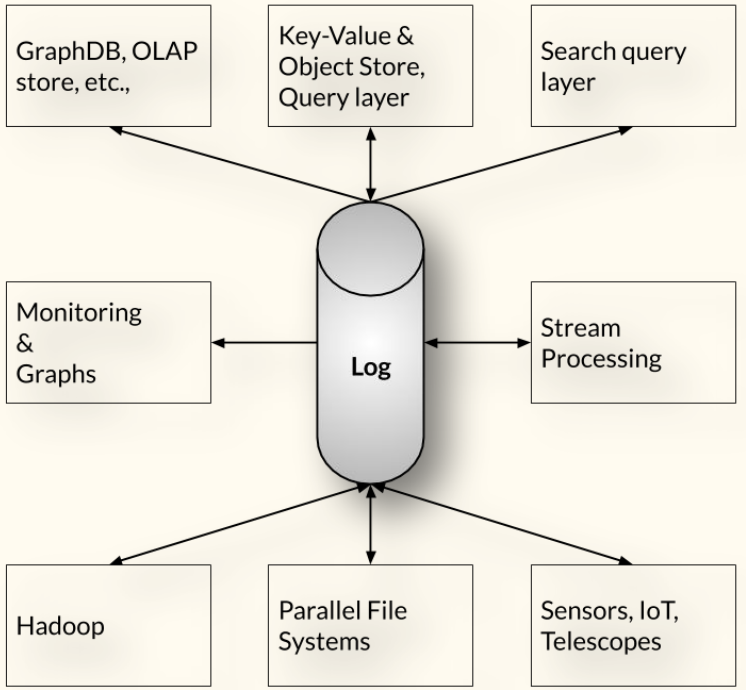
Shared Log abstraction

- A strong and versatile primitive
  - at the core of many distributed data systems and real-time applications
- A shared log can act as
  - an authoritative source of strong consistency (global shared truth)
  - a durable data store with fast appends and “commit” semantics
  - an arbitrator offering transactional isolation, atomicity, and durability
  - a consensus engine for consistent replication and indexing services
  - an execution history for replica creation
- A shared log can enable
  - fault-tolerant databases
  - metadata and coordination services
  - key-value and object stores
  - filesystem namespaces
  - failure atomicity
  - consistent checkpoint snapshots
  - geo-distribution
  - data integration and warehousing

akougkas@iit.edu

Log as the backend

- Data intensive computing requires a capable storage infrastructure
- A distributed shared log store can be in the center of scalable storage services
- Additional storage abstractions can be built on top of a distributed shared log
- Logs can support a wide variety of different application requirements



akougkas@iit.edu

State-of-the-art log stores

- Cloud community
  - Bookkeeper, Kafka, DLog
- HPC community
  - Corfu, SloG, Zlog
- Commonalities
  - The logical abstraction of a shared log
  - APIs

Features	Bookkeeper Kafka / DLog	Corfu SloG / ZLog	ChronoLog
Locating the log-tail	MDM lookup (locking)	Sequencer (locking)	MDM lookup (lock-free)
I/O isolation	Yes	No	Yes
I/O parallelism (readers-to-servers)	1-to-1	1-to-N	M-to-N (always)
Storage elasticity (scaling capacity)	Only horizontal	No	Vertical and horizontal
Log hot zones	Yes (active ledger)	No	No
Log capacity	Data retention	Limited by # of SSDs	Infinite (auto-tiering)
Operation Parallelism	Only Read (Implicit)	Write/Read	Write/Read
Granularity of data distribution	Closed Ledgers (log-partition)	SSD page (set of entries)	Event (per entry)
Log total ordering	No (only on partitions)	Yes (eventually)	Yes
Log entry visibility	Immediate	End of epoch	Immediate
Storage overhead per entry	Yes (2x)	No	No
Tiered storage	No	No	Yes

akougkas@iit.edu

Existing log store shortcomings

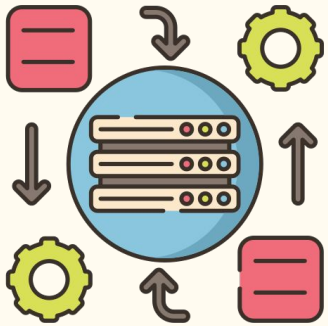
- Limited parallelism
  - Data distribution, Serving requests (SWMR model)
- Increased Tail Lookup Cost
  - Mapping lookup cost (MDM OR Sequencing)
- Expensive Synchronization
  - Epochs and commits
- Partial Ordering
  - Segment/partition and NOT in the entire log
- Lack of support for hierarchical storage
  - A log resides in only a single tier of storage

Main Challenge

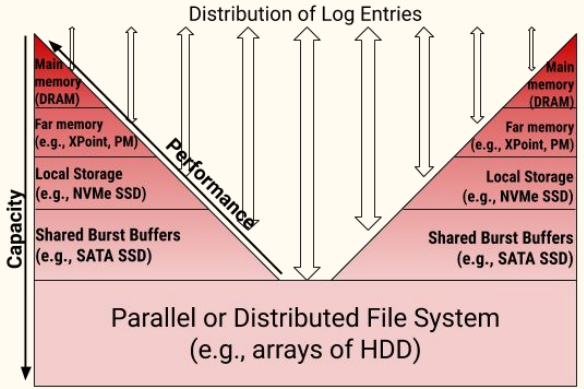
How to balance log ordering, write-availability, log capacity scaling, parallelism, log entry discoverability, and performance?

Two key insights - Motivation

- A combination of the append-only nature of a log abstraction and the natural strict order of a global truth, such as *physical time*, can be combined to build a distributed shared log store that avoids the need for expensive synchronizations.



- An efficient mapping of the log entries to the tiers of a storage hierarchy can help scale the capacity of the log and offers two important I/O characteristics: *tunable access parallelism* and *I/O isolation* between tail and historical log operations.



akougkas@iit.edu

Ramifications of physical time

- Using physical time to distribute and order data is beneficial[1]
  - Avoids expensive locking and synchronization mechanisms
  - However, maintaining the same time across multiple machines is a challenge
- Our thesis:
  - *Physical time only makes sense in a log context since it is an immutable append-only structure that only moves forward, like a physical clock does!*
- Three major challenges:
  - Taming the Clock Uncertainty
  - Handling Backdated Events
  - Handling Event Collision

ChronoLog provides solutions to these challenges

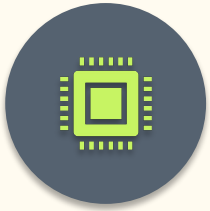
[1] Corbett, James C., Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, et al. "*Spanner: Google's Globally-Distributed Database*." In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pp. 261-264. 2012.

akougkas@iit.edu

A Distributed Tiered Shared Log Store

In a glance

- ChronoLog is a new distributed shared and tiered log store responsible for the organization, storage, and retrieval of activity data
- **Main objective**
  - *support a wide variety of applications with conflicting log requirements under a single platform*
- **Major contributions**



SYNCHRONIZATION-FREE LOG ORDERING USING *PHYSICAL TIME*

LOG SCALING VIA *AUTO-TIERING* IN MULTIPLE *STORAGE TIERS*

HIGHLY *CONCURRENT* LOG ACCESS MODEL (MWMR)

RANGE RETRIEVAL MECHANISMS (PARTIAL GET)

akougkas@iit.edu

Design requirements

Log Distribution

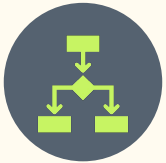
Highly parallel data distribution in the event granularity

3D distribution forming a square pyramidal frustum (3-tuple of {log,node,tier})

Log Ordering

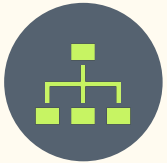
Sync-free tail finding

Total log ordering guarantee



Log Access

Multiple-Writer-Multiple-Reader (MWMR) access model



Log Scaling

Automatically expand the log footprint via auto-tiering across hierarchical storage environments

Log Storage

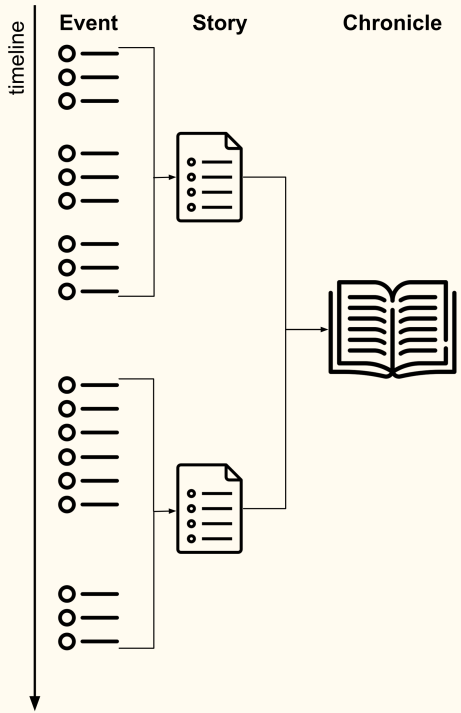
Tunable parallel I/O model

Elastic storage capabilities

akougkas@iit.edu

Data model and terminology

- Chronicle
  - a named data structure that consists of a collection of data elements (events) ordered by physical time (i.e., topic, log, stream, ledger)
- Event
  - a single data unit (i.e., message, record, entry) as a key-value pair
    - the key is a ChronoTick (time slot) and the value is an uninterpreted byte array
- ChronoTick: a monotonically increasing positive integer
  - represents the time distance from the chronicle's base value (i.e., offset from chronicle creation timestamp)
- Story
  - a story is a division of a chronicle (i.e., partition, segment, fragment)
    - a sorted immutable collection of events great for sequential access on top of HDDs



akougkas@iit.edu

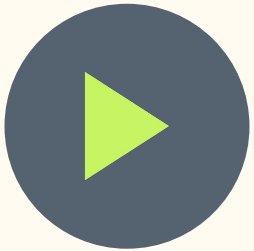
Chronicle acquisition

- Clients acquire/release a chronicle (e.g., open/close a file)
  - before any data operation can occur
- Acquisitions (projections):
  - Fully
  - Partially (forming a projection of a chronicle)
    - From chronicle-start till a given eventID
    - From a given eventID till the chronicle-end
    - Between two eventIDs
  - Time-based
    - Timer expiration (e.g., acquire for 5 mins)
    - Handles orphaned handles of acquired chronicles
- **Chronicle projections** increase resource utilization and performance

Basic Operations

- Supports typical log operations
- ChronoLog allows replay operations to accept a range (start-end events) for partial access

RECORD
AN EVENT (APPEND)



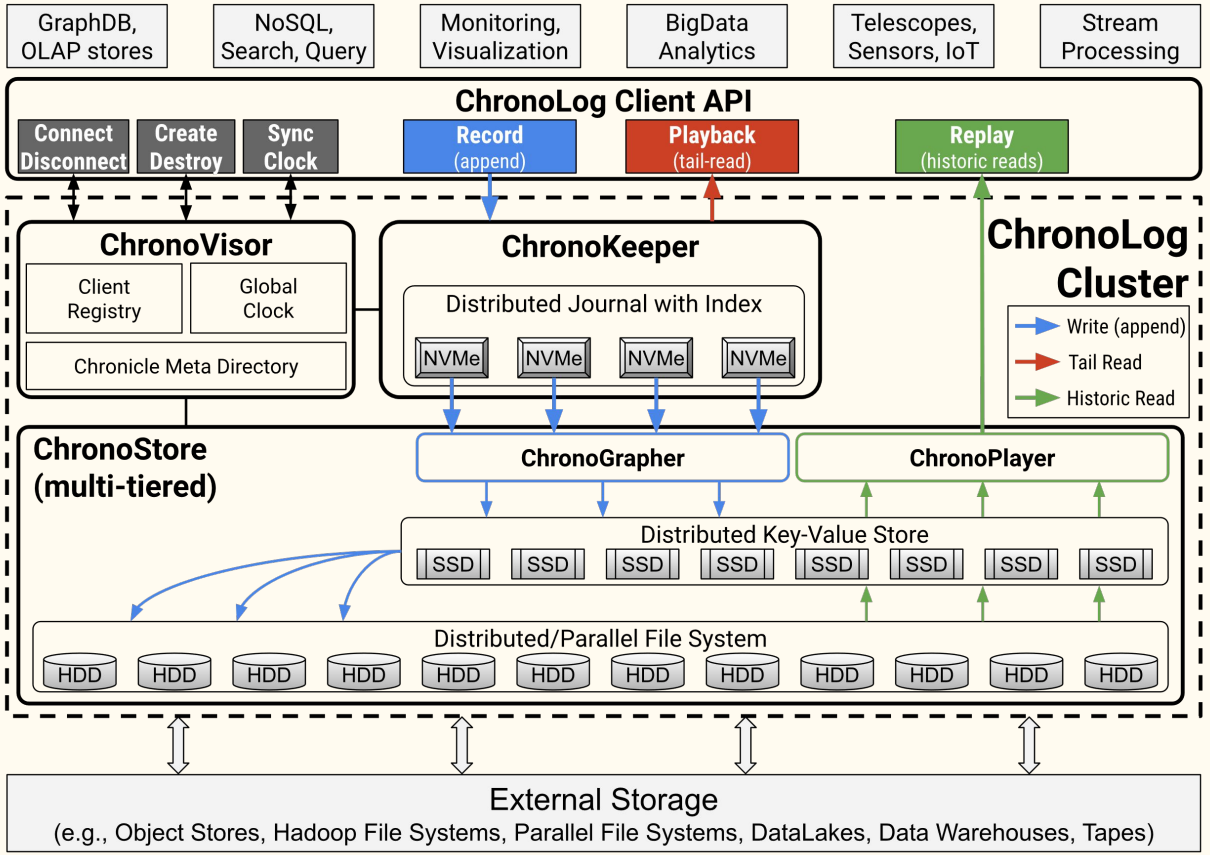
PLAYBACK
A CHRONICLE
(TAIL-READ)

REPLAY
A CHRONICLE
(HISTORICAL READ)

akougkas@iit.edu

System overview

- Client API
- ChronoVisor
  - Client connections
  - Chronicle metadata
  - Global clock
- ChronoKeeper
  - All tail operations
- ChronoStore
  - ChronoGrapher
  - ChronoPlayer



akougkas@iit.edu

ChronoLog API

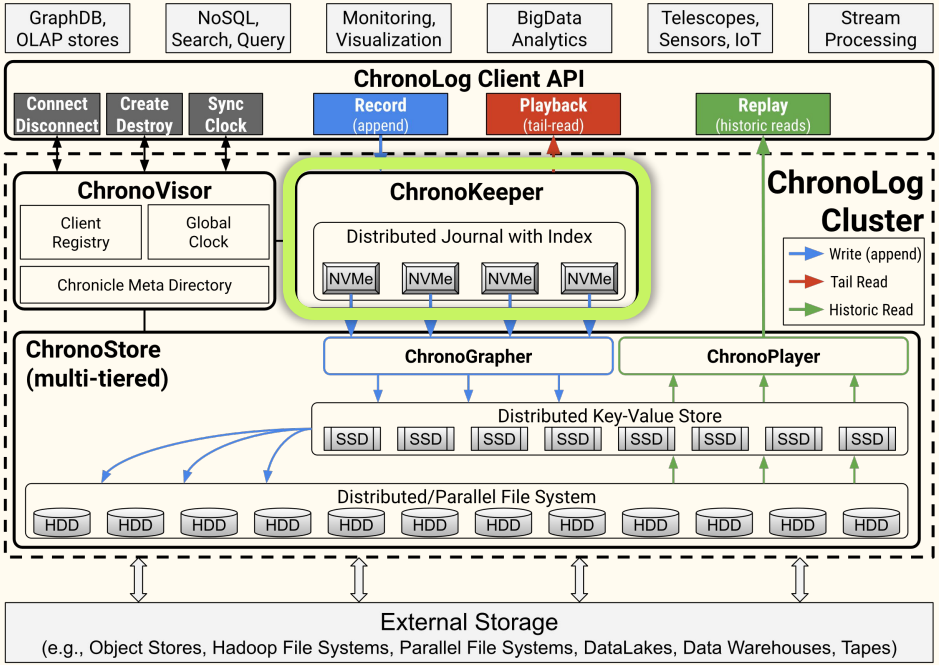
Operation	Args	Return	Description
			Admin API
`connect()`	URL	status	Connects to the cluster using the ChronoVisor's URL. (e.g., chronolog://<hostname>:<port>)
`disconnect()`	NULL	status	Terminates the connection to the cluster
`sync_clock()`	URL	base, rate	Gets ChronoVisor's (URL) global clock value (base) and its ticking drift rate (rate). Function is called when client first connects, periodically, and on chronicle creation or migration.
			Chronicle API
`create()`	name, index, tags	status	Creates a chronicle with name, with event granularity defined by index. Default indexing is in nanoseconds but larger units can also be selected. Tags is a set of attributes such as type of chronicle, access permissions, tiering policy, etc.,
`edit()`	name, index, tags	status	Edit a chronicle (e.g., renaming, re-indexing, and re-tagging).
`destroy()`	name, flags	status	Deletes the entire chronicle. Flags define a sync or async operation. ChronoLog will delete a chronicle only when all acquisitions are released (i.e., reference_count =0).
`acquire()`	name, flags	CID	Gets the ChronicleID (CID) associated with name. Type of acquisition (e.g., exclusive/shared, full/partial) defined by flags.
`release()`	CID, flags	status	Releases the acquired chronicle. Reduces reference count by 1. An expiration time can be defined by flags.
			Event API
`record()`	CID, data	EID	Appends the serialized data to the chronicle with CID. An eventID (EID) is returned upon success.
`playback()`	CID	data	Gets the data at the tail of the chronicle with CID.
`replay()`	CID, range, constraint	data	Gets any data between the requested range <startEID, endEID>. Filtering of the retrieved data by applying the constraint.

```
1  #include <chronolog.hpp>
2  ...
3  Connection conn = connect("chronolog://localhost:1234");
4  Chronicle chronicle = create("my-chronicle", DEFAULT_TAGS);
5  if (ChronoLogHandle *cl = acquire(chronicle, EXCLUSIVE)) {
6      // Write events
7      void *data = serialize("hello-MSST-Attendees");
8      EventID eid = record(cl, data);
9      ...
10     // Get latest entry
11     char* tail = deserialize(playback(cl));
12     std::cout << "latest event" << tail << '\n';
13     // Get all entries sequentially
14     auto range = std::pair<EventID, EventID>(0, 100);
15     std::vector<char *> data = replay(cl, range, NULL);
16     for (auto it = data.cbegin(); it != data.cend(); ++it)
17       std::cout << ' ' << *it << '\n';
18 }
19 release(cl);
20 destroy("my-chronicle");
21 disconnect();
22 ...
```


akougkas@iit.edu

ChronoKeeper

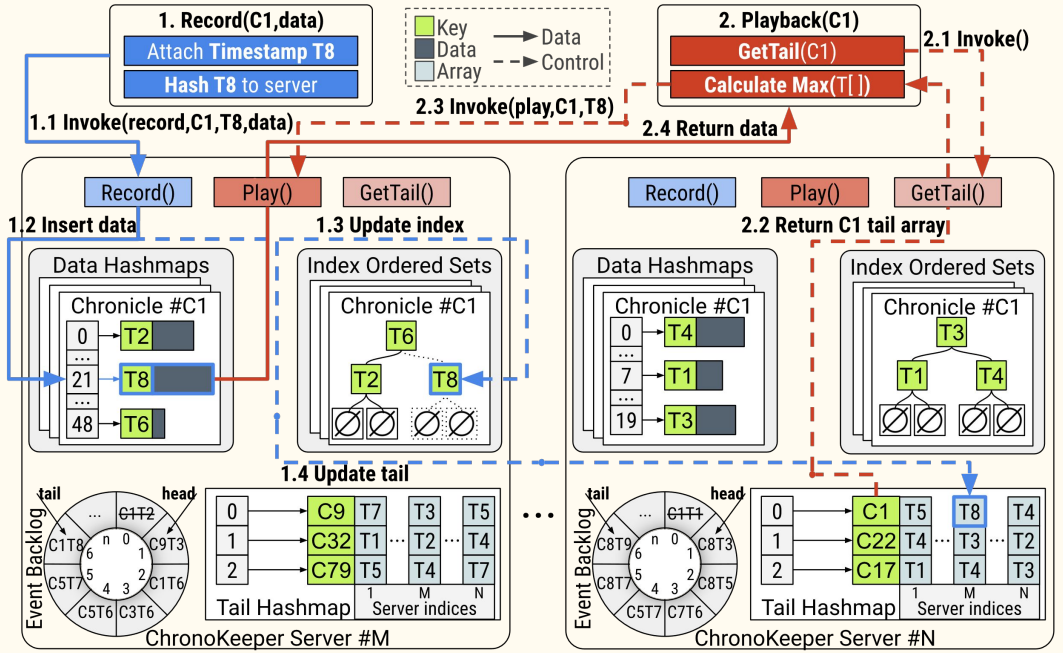
- Runs on highest tier of hierarchy (e.g., DRAM, NVMe)
- Distributed journal
- Fast indexing
- Lock-free locating the log tail
- Event backlog for caching effect



akougkas@iit.edu

ChronoKeeper – Record()

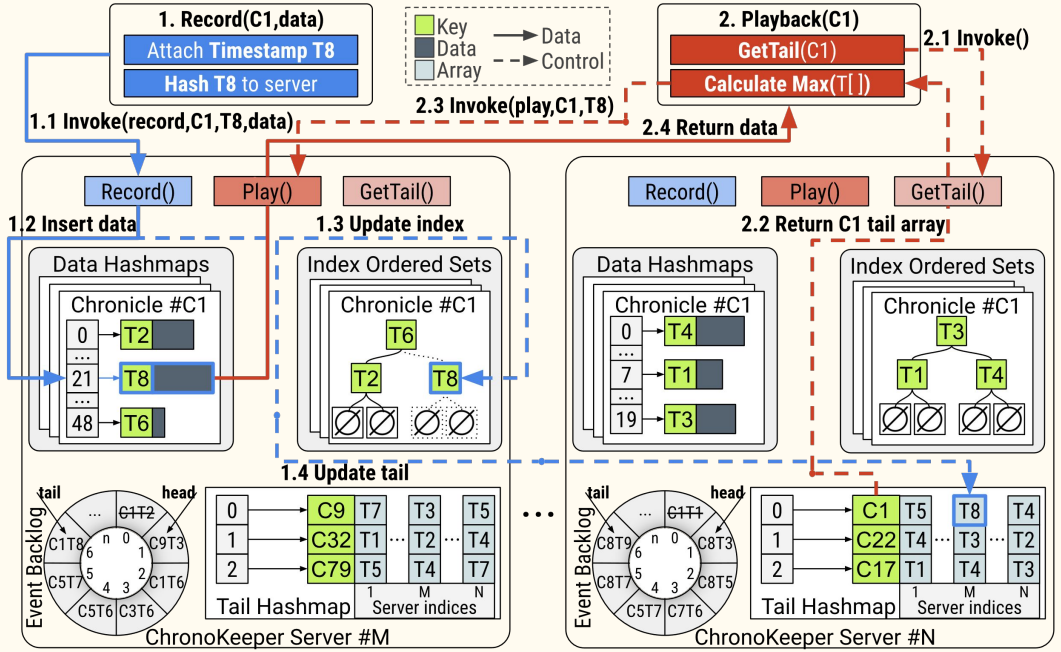
- Client lib
  - attaches ChronoTick and uniformly hashes eventID to a server
  - no need for a sequencer
- Server
  - pushes data to a data hashmap and
  - at the same time updates the index and tail hashmap atomically (overlapped)



akougkas@iit.edu

ChronoKeeper – Playback()

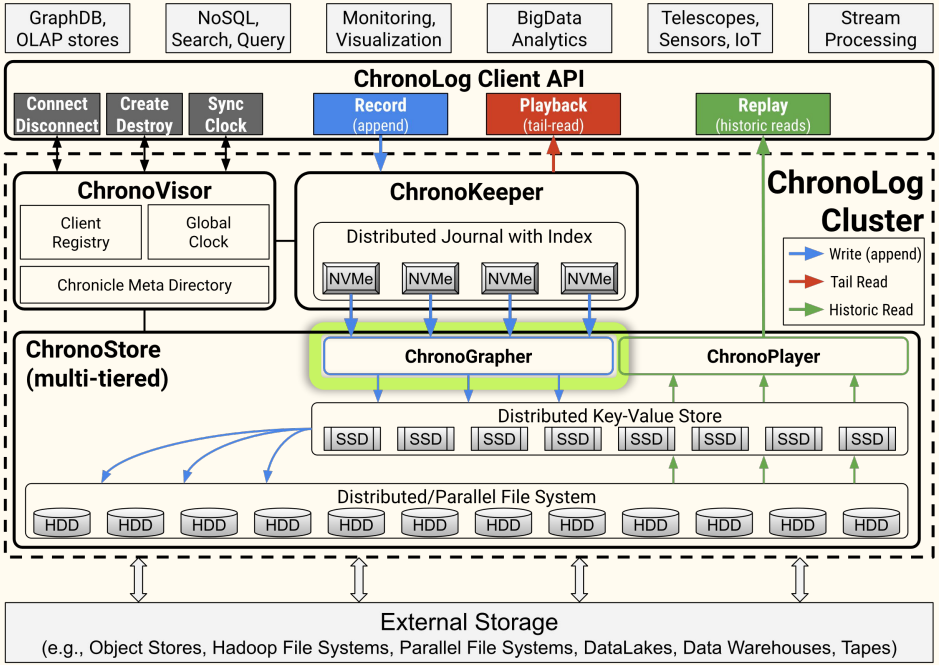
- Client lib
  - invokes get_tail() on the server
  - gets a vector of latest eventIDs per server
  - calculate the max ChronoTick
  - invoke play() on the server
- Server
  - fetches data from hashmap
- Delivery Guarantee:
  - no later event from timestamp of playback() call + network latency



akougkas@iit.edu

ChronoGrapher

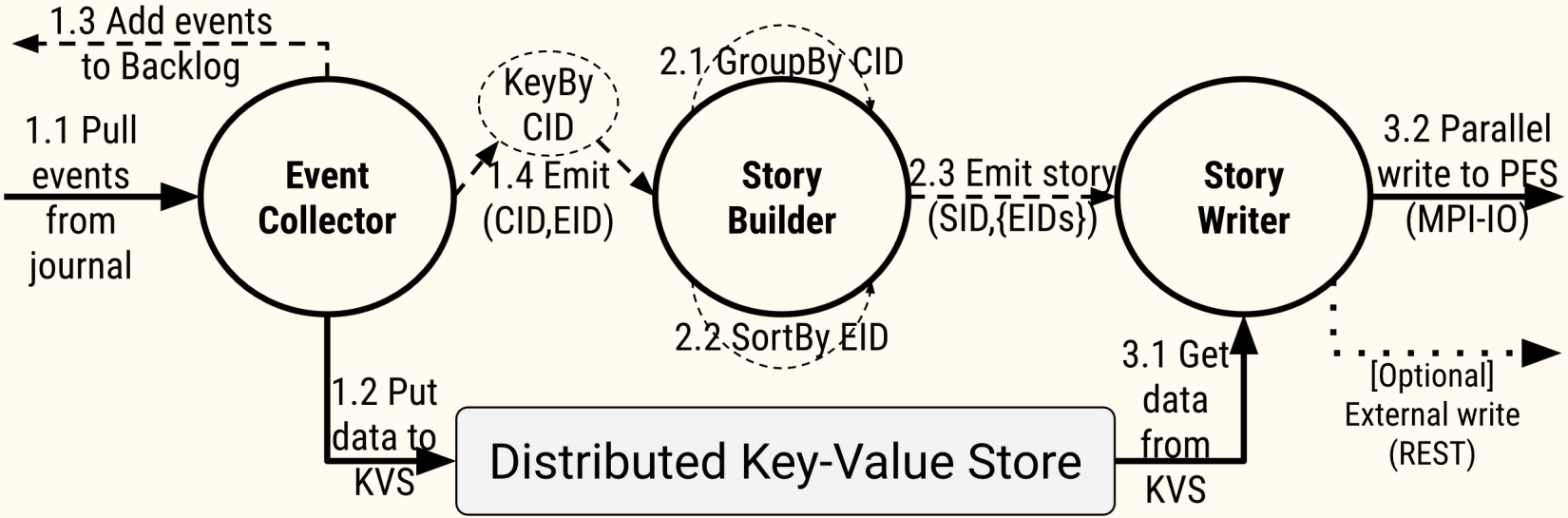
- Absorbs data from ChronoKeeper in a continuous streaming fashion
- Runs a distributed key-value store service on top of flash storage
- Utilize SSDs capability for random access but create sequential access for HDD
- Implements a server-pull model for data eviction from the upper tiers
- Elastic resource management matching incoming data rates



akougkas@iit.edu

ChronoGrapher
Recording data

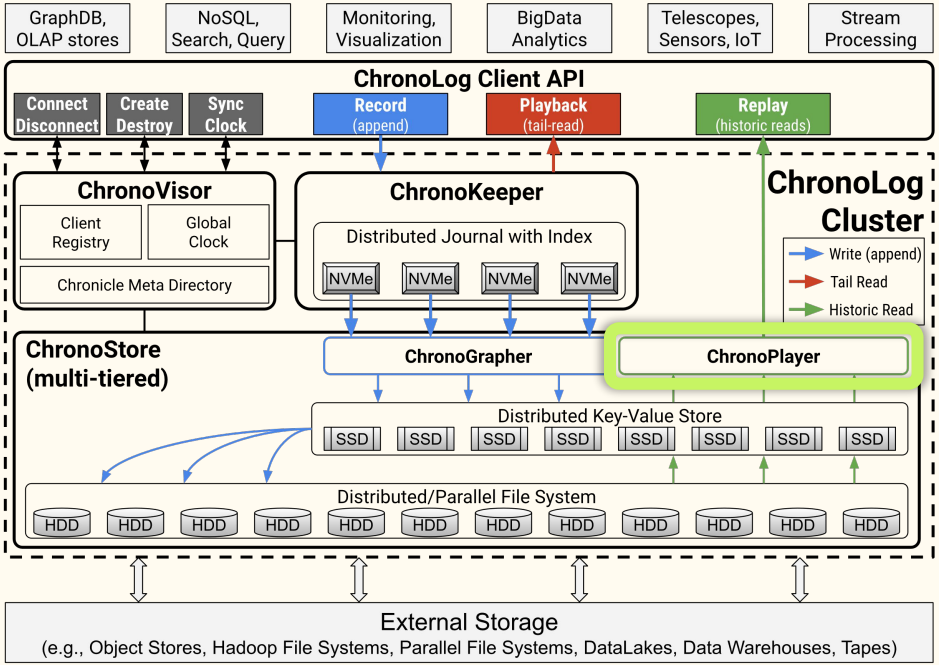
- **Event collector:** pulls events from ChronoKeeper
- **Story builder:** groups and sorts eventIDs per chronicle
- **Story writer:** persists stories to the bottom tier using parallel I/O



akougkas@iit.edu

ChronoPlayer

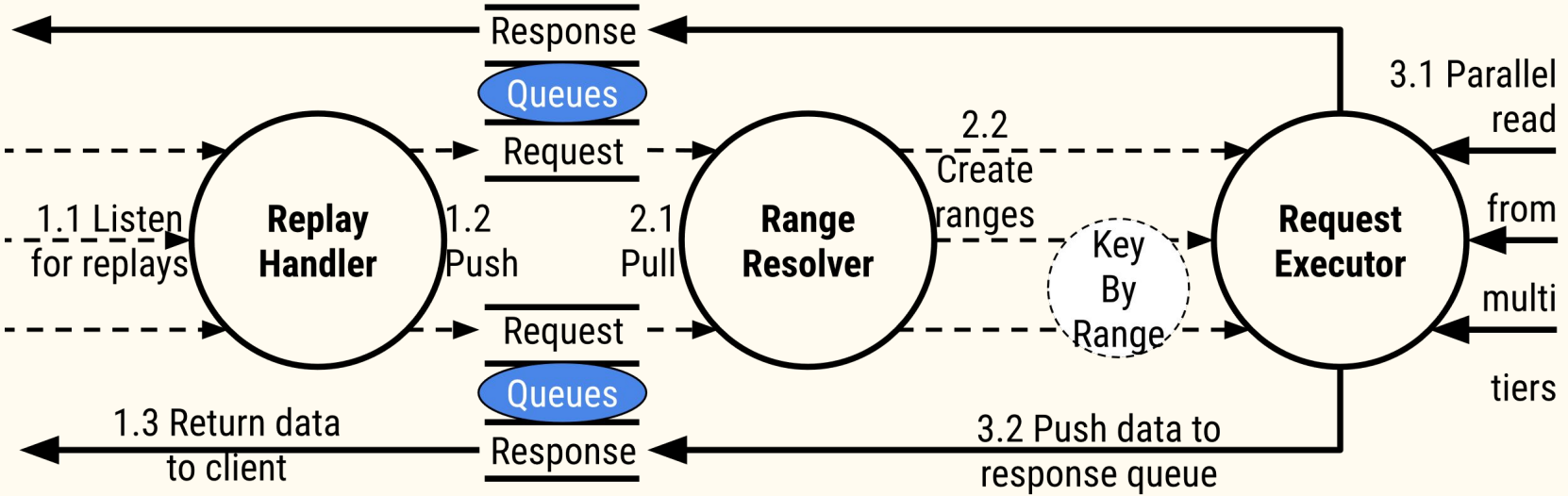
- Executes historical reads
- Deployed on all storage nodes in a ChronoStore cluster
- Locate and fetch events in the entire hierarchy by accessing:
  - PFS on HDDs
  - KVS on SSDs
  - Journal on NVMe using ChronoKeeper's indexing
- Implements a decoupled, elastic, and streaming architecture



akougkas@iit.edu

ChronoGrapher
Replaying data

- **Replay handler:** listens for requests and queues them
- **Range resolver:** processes requests and produces a vector of eventID ranges
- **Request executor:** deduplicates ranges and executes the reading

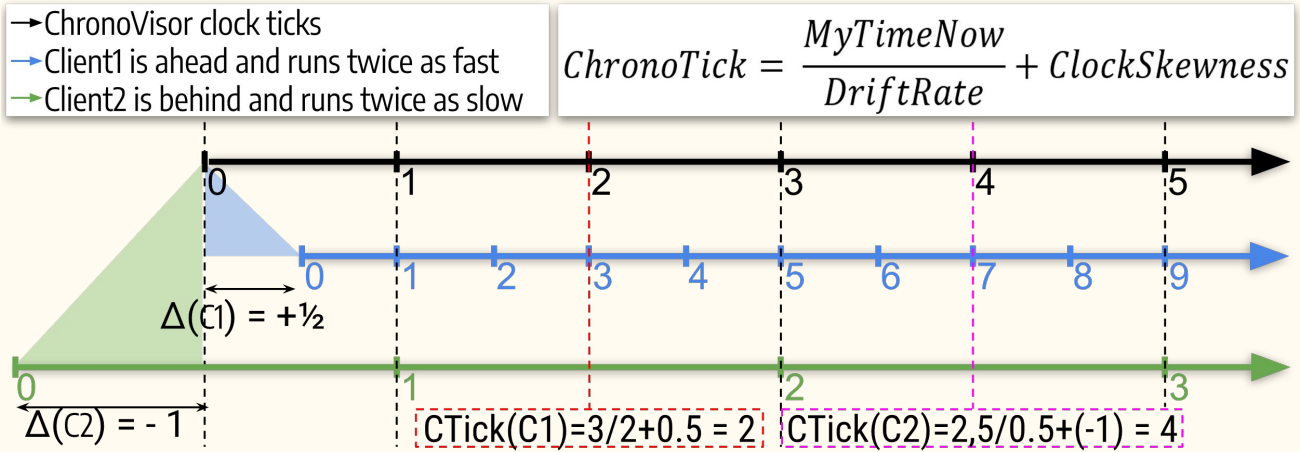


akougkas@iit.edu

Dealing with Physical Time

Taming the clock uncertainty

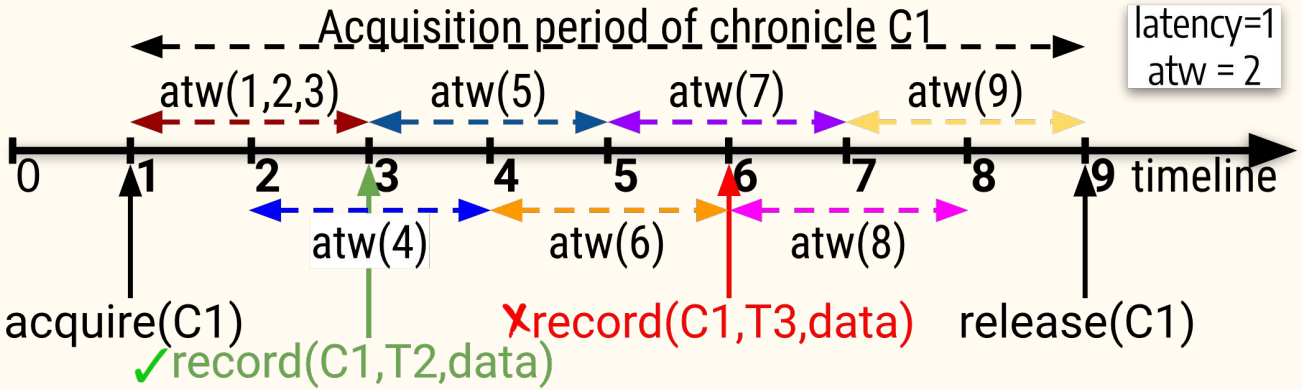
- Issues
  - Time distance between two clocks
  - Different ticking rates (a.k.a drift rates)
- Solution
  - Server nodes sync with ChronoVisor during init and periodically afterwards
  - Clients use ChronoTicks as a relative distance from a base clock



akougkas@iit.edu

Backdated events

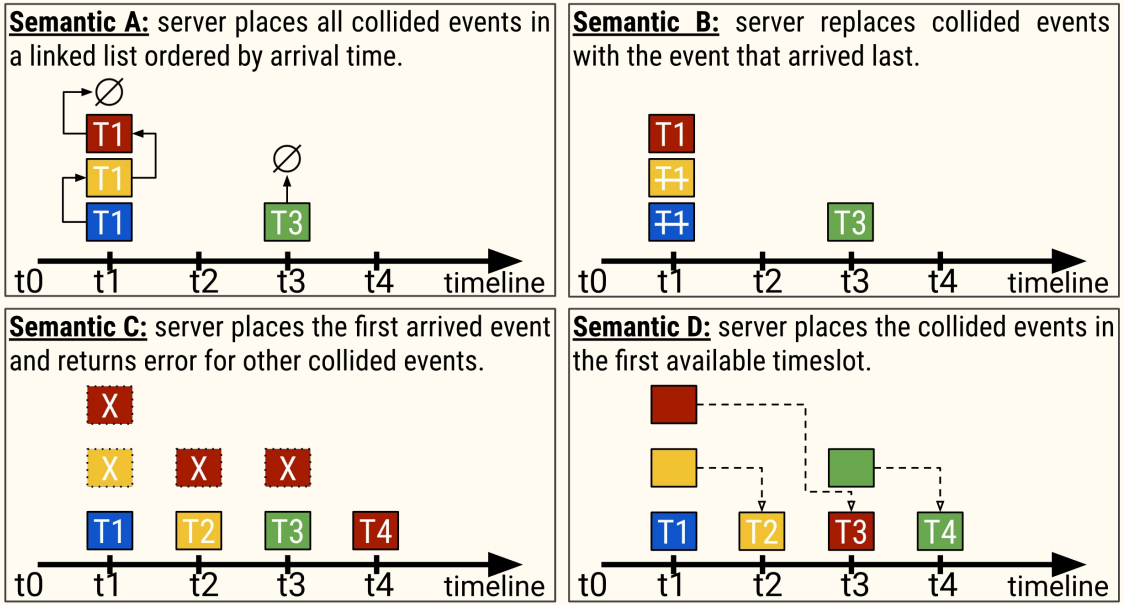
- Due to network non-determinism, events may arrive later violating the immutability and the ordering of a chronicle (backdating)
- ChronoLog defines an *Acceptance Time Window* (ATW)
  - ATW is a moving time window imposed on each chronicle acquisition period
  - ATW is equal to twice the network latency between the clients and ChronoLog
    - Latency as measured during client connection or chronicle acquisition



akougkas@iit.edu

Event collisions

- Chronicle indexing granularity is based on physical time (ChronoTicks)
- For coarser granularities, events might collide
  - How to detect a collision
  - How to correct a collision
- Workload objectives
  - SemanticA: Idempotent
  - SemanticB: Redudancy
  - SemanticC: Ordering
  - SemanticD: Sequentiality



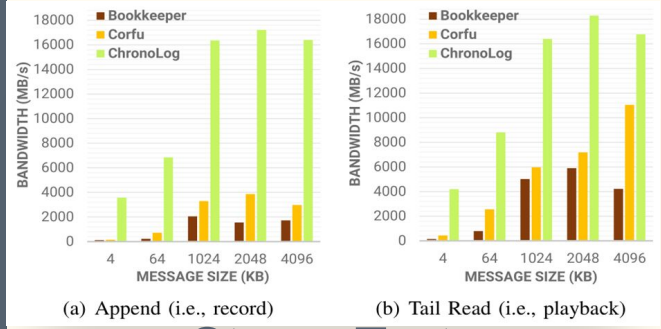
akougkas@iit.edu

All tests were conducted on the Ares cluster at Illinois Institute of Technology using:

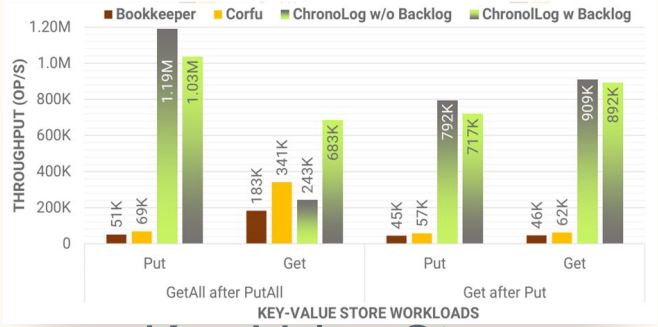
- 24 client nodes
- 8 BB nodes
- 32 storage nodes
  - various storage devices (NVMe, SSD, HDD)
- 40GBit Ethernet network with RoCE enabled

Experimental Results

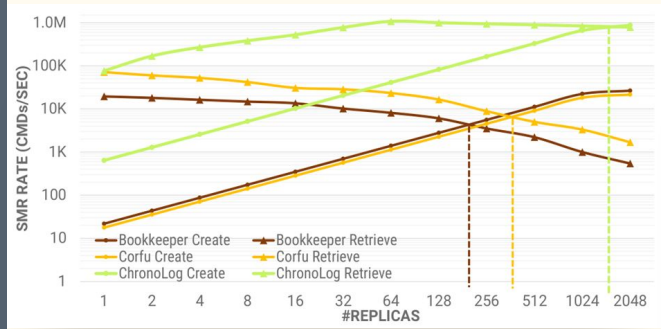
Application Workloads



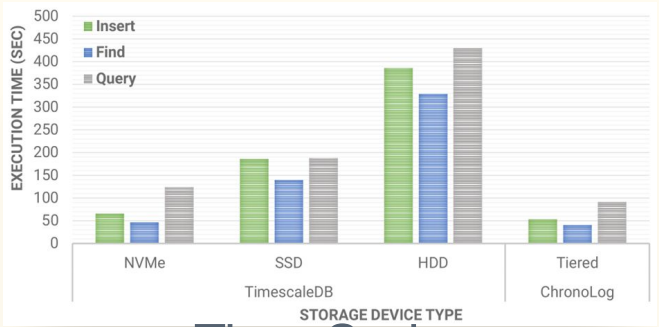
Stress Test



Key-Value Store



State Machine Replica

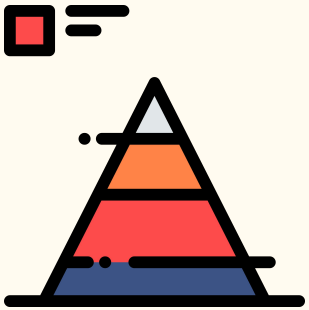


Time Series

- Stress-test:
  - All clients issue 32K record-playback requests
  - ChronoLog outperformed both by a significant margin due to its lack of synchronizations
- KVS:
  - All clients issue 32K put-get requests
  - Corfu faster than Bookkeeper due to more parallelism
  - ChronoLog is **2-14x** faster
- SMR:
  - All clients log instructions in a replica set
  - ChronoLog saturates at 1900 replicas, making it **5x** faster
- Timeseries:
  - The tiered approach and the time-based indexing provides a **25%** improvement

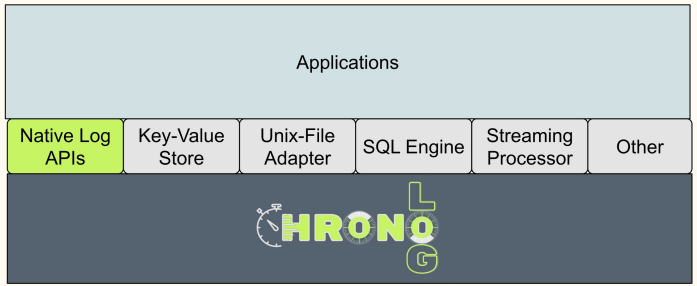
Conclusions

The rise of log data in modern applications expects a distributed shared log store with total ordering that is capable to scale well



Modern storage stacks need to be elevated to take advantage of the new types of storage devices and offer superior performance

- **ChronoLog** uses
  - A truly hierarchical design and a decoupled and elastic architecture to match the I/O production and consumption rates from clients
  - Physical time to distribute and order data to boost **performance by** eliminating a centralized synchronization point
  - Future work: extend the ecosystem



akougkas@iit.edu

SCALABLE COMPUTING
SOFTWARE LABORATORY

ILLINOIS INSTITUTE
OF TECHNOLOGY

Thank you

Anthony Kougkas
akougkas@iit.edu

Special thanks to our sponsor
National Science Foundation
NSF