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

Anthony Kougkas
akougkas@iit.edu
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
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.
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
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
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

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

- **Client API**
  - Client connections
  - Chronicle metadata
  - Global clock

- **ChronoVisor**
  - Client connections
  - Chronicle metadata
  - Global clock

- **ChronoKeeper**
  - All tail operations

- **ChronoStore**
  - ChronoGrapher
  - ChronoPlayer
ChronoLog API

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

\[
\text{ChronoTick} = \frac{\text{MyTimeNow}}{\text{DriftRate}} + \text{ClockSkewness}
\]
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.

![Diagram](Diagram.png)

- **Acquisition period of chronicle C1**
  - `atw(1,2,3)`
  - `atw(4)`
  - `atw(5)`
  - `atw(6)`
  - `atw(7)`
  - `atw(8)`
  - `atw(9)`

- **Timeline:**
  - `0` to `9`
  - `latency=1`
  - `atw = 2`

- **Actions:**
  - `acquire(C1)`
  - `record(C1,T2,data)`
  - `record(C1,T3,data)`
  - `release(C1)`

---

*Slide 26*
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
  - Semantic A: Idempotent
  - Semantic B: Redudancy
  - Semantic C: Ordering
  - Semantic D: Sequentiality
Experimental Results

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
- ChronoKeeper scales quite linearly achieving ~1M ops
- ChronoGrapher achieved max PFS BW at around 3GB/s
- ChronoPlayer shows stable performance with various event size
- Lightweight client with only 16% of the overall time
- Majority of the time for a record() call 84% is spent in ChronoGrapher
- No identifiable bottleneck issues
- **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 \(2-14\times\) faster

- **SMR:**
  - All clients log instructions in a replica set
  - ChronoLog saturates at 1900 replicas, making it \(5\times\) faster

- **Timeseries:**
  - The tiered approach and the time-based indexing provides a \(25\%\) improvement
Conclusions

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

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.
Thank you

Anthony Kougkas
akougkas@iit.edu