Exploring Large-Scale, Distributed Event-Sourced Systems in Ray

12/18/2020
Yudi Tan; yudi98@berkeley.edu
Yichi Zhang; billyz@berkeley.edu
Justin Yang; justinwyang@berkeley.edu
Mentors: David Ott; dott@vmware.com and Lars Karg; lkarg@vmware.com

Background Context (The Need For Event-Sourcing)

Many modern-day applications often require audit logs in order to facilitate various business needs. However, traditional business systems are often snapshot-based, where the global state of the system is captured and stored in databases while the intermediate actions, the “events”, which cause these state changes are not persisted. As business needs evolve, we find that there are several common and recurring issues with such snapshot-based systems [6]:

1) Backfilling data becomes an issue when intermediary events are not persisted.
2) Difficulty with exploring the global state of the system at a particular point in time in the past (i.e. time-travel)
3) We are stuck with 1 representation / view of the global state and can be constraining (i.e. not being able to go back in time and explore alternate “branches” of events means less opportunities for ML/RL, quantitative trading strategies etc).

To provide such functionality, a new system-design paradigm is to incorporate events as first-class citizens of such systems where instead of storing snapshots of global states, all events which mutate the global state are stored immutably and the global state can be constructed by replaying the stream of events. This is analogous to a write-ahead-log in databases where client commands are recorded immutably in an append-only log which is stored separately from the actual data which contains the global states.

The Problem

While event-sourcing seems to be able to solve many of the above issues with traditional snapshot-based systems, real-world implementations of event-sourced systems are often extremely complicated. Specifically, due to the asynchronous nature of such systems and the separation of events from the state, there can be issues with state-reconstruction latency, state consistency as well as implementation complexity.

While the idea and components of an event-sourced system are well-known, there isn’t a scalable framework available for practitioners to implement such a system which can also support a huge variety of different workloads. In fact, there are typically two different ways to implement an event-sourced system. On one end of the spectrum, we have event-sourced

systems implemented using custom libraries tailored to the specific use-case, while also relying on open-sourced infrastructures like Kafka as a central backbone of the system for persisting events and for relaying messages across different entities. While this is a fairly common approach in industry, the issue with this is that the code is not very portable as most of these custom-libraries target specific system designs. Moreover, when the event-sourced system needs to be distributed and fault-tolerant, these kinds of implementation often convolute business logic with systems code.

On the other end of the spectrum lies specialized event-sourced frameworks such as Akka [2] and Eventuate [1]. While these actor-based frameworks provide great and convenient abstractions for building event-sourced backends, the type of workload they support are very limited and they are not too performant. As a result, it is often a challenge in practice to integrate an event-sourced system implemented in such frameworks with other subsystems such as data-processing pipelines.

In other words, the lack of a mature, performant, scalable and generic implementation for this programming paradigm means that there is a high barrier to adopt this event-sourced style of system design.

Existing Solutions

The most popular framework for building event-sourced systems is the Akka library for actors. Many existing event-sourced systems are implemented on top of Akka, and there are also many higher-level libraries built on top of Akka, such as Eventuate. Akka and Eventuate provide many handy abstractions for event-sourcing, such as command and query actors, event-reactor actors as well as persistent actors. A typical event-sourced system built using these libraries will consist of many of these actors which communicate with each other through message-passing and form a cohesive pipeline [1].

In order to provide causal-consistency as well as persistence in face of fault-tolerance, Eventuate also provides several abstractions like CRDTs, vector-clocks as well as persistent actors. Despite providing these useful abstractions to ease development of event-sourced systems, these libraries have two major flaws. First, these libraries are typically high-level libraries which do not handle distribution across a cluster. As such, developers often need to write systems code which handle such issues in order to provide scalability and fault-tolerance. Another limitation of these frameworks is that they were never developed to be generic and performant enough to support a wide variety of workloads. As such, the possible use-cases of such frameworks can be quite limited.

However, from our research of these frameworks, it became clear to us that event-sourcing pairs really well with the actor model in practice. Multiple independent event streams (i.e. producers) can be modelled as independent actors, while event-aggregators and handlers can be modelled as other actors. Since actors do not coordinate with each other and do not depend on locks, they are very performant and scalable. This is why we decide to explore

event-sourcing with Ray, since Ray provides a highly scalable, performant and fault-tolerant actor implementation in a distributed setting, abstracting away these systems and performance concerns away from application developers.

In our work, we implemented several event-sourced applications in Ray, gaining inspirations from existing event-sourced systems built on Akka [6] [7].

Approach

After evaluating the limitations of existing solutions, we moved on to exploring Ray's capability for supporting event-sourced applications. We have two main goals throughout this project; the first goal is to explore the various design patterns and considerations when implementing event-sourced systems on top of Ray, while the second is to profile and benchmark the performance of such implementations.

Throughout the semester, we implemented two complete event-sourced applications using Ray's actor model. These two applications were architected very differently, in order to explore several design patterns as well as to evaluate their pros and cons. The first application is a banking application [appendix 3], where there is a single actor holding an in-memory hashmap representation of the ledger of the entire system. This ledger represents the typical global state of an application, and can be reconstructed on-the-fly using events persisted in a datastore. This application supports several commands (following the Command-Query-Responsibility-Segregation CQRS paradigm), including but not limited to querying a user's balance, deposits and withdrawals. These operations interact with the single actor, which will then validate these commands and either store corresponding events in an append-only event log or replay events from the log to retrieve the latest state.

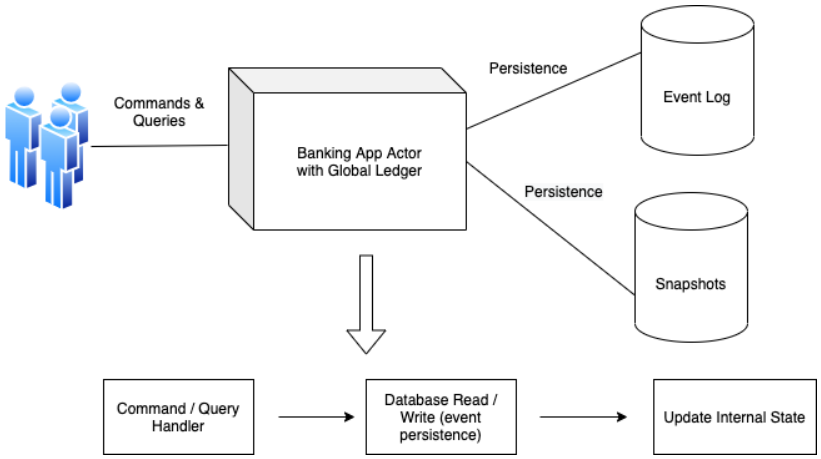

Fig 1. Banking Application System Overview

We then extended this set-up to include periodic snapshots of the global state, since without snapshots the read-latency will grow linearly with the size of the event-log (more on this in the evaluation section). With persistent events and snapshots, we were able to efficiently support

event-replay. Moreover, since Ray handles fault-tolerance for us as actors crash, the global state is mostly consistent (with a caveat explored later on in this section) even as the actors crash, since all events are persisted in a database and the global state can be fully reconstructed whenever needed. With event-replay abilities, our application was also able to support time-travelling operations since users could also input a custom event-log with mutated entries and run the event-replay method to explore alternate-realities.

As we were primarily concerned with evaluating the event-replay operation to ensure that the global state is consistent even in face of actor crashes, we did not thoroughly explore the potential of time-travel operations, though we are hopeful that time-travelling allows for increasing training data in certain use-cases of deep-learning and can be quite beneficial.

The banking application served to explore the engineering efforts needed to implement a basic event-sourced system in Ray; from our experience, Ray's Core API provides just enough abstractions to build a working implementation, though it lacks certain features needed to ensure full consistency. For instance, Ray's actors do not provide durable mailboxes and are not persistent. This means that despite persisting events and snapshots at an application level, messages that have arrived at an actor's mailbox but have not been persisted in the event-log may be lost in a crash. From Ray's perspective, this design decision makes a lot of sense since not all applications require such persistent guarantees and therefore providing it at Ray's level will increase overhead and reduce Ray's support for certain workloads which require all the performance they can get. However, such a guarantee is absolutely critical for event-sourced systems where events are the atomic units of truth. As such, this was a core limitation of Ray that we realized while implementing this banking application. Besides this, however, we are very pleased with Ray's abstractions since it allowed us to implement a complete distributed and fault-tolerant event-source backend system with minimal code changes.

The second application is a shipping application consisting of multiple independent actors, each containing their own private states [appendix 4]. This multi-actor design pattern aims to provide insight into data-sharding in Ray as well as event-aggregation across independent streams. With the shipping application, we were able to see where Ray truly shines (i.e. Ray is optimal and provides high performance and ease of parallelization across cores when events are sharded across many actors and can be processed simultaneously). However, here we realized another limitation of Ray. As we attempted to aggregate these independent event streams into a single event-log, we were met with challenges of providing causal-consistency. Since Ray schedules these actors across nodes / cores in a cluster, we cannot depend on the local timestamps to arrange the events into a global order. Fortunately, for our case, most of our events are not order-dependent and hence event-aggregation was not a major issue, but this is still a limitation for many other workloads. Borrowing ideas from Eventuate, a potential solution for this is to implement vector-clocks and/or CRDTs on an application-level event-sourcing library to provide these functionalities to event-source application developers on Ray.

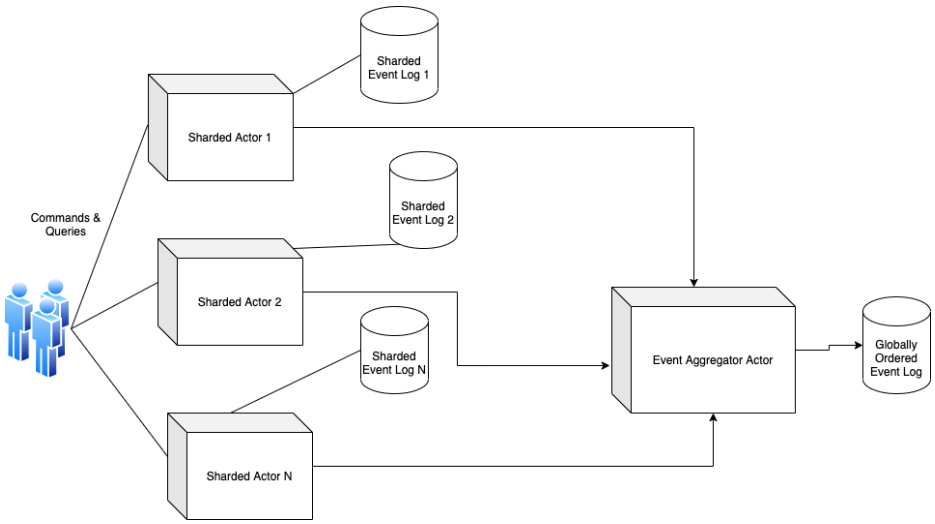

Fig 2. Shipping App System Overview

Once we had these two complete applications implemented, which have high performance and scalability through periodic snapshots and sharding, while also supporting handy features like time-travelling and event-aggregation, we moved on to benchmarks. With the aforementioned limitations in mind, we also wanted to explore scenarios where Ray truly outperformed vanilla event-sourced systems. As such, we compared the performance of the 2 applications with their Python counterparts -- we developed 2 more event-source applications written in vanilla Python (without Ray) which supports the same operations as their Ray counterparts. Since there aren't any industrial benchmarks for event-sourced systems, we also had to implement our own event-generator to benchmark these performances. Our test generator codegens test code which runs the different applications (in Ray versus their vanilla Python counterparts) with different configurations, varying parameters like number of events, snapshot frequency and sharding factor. Then, we measured the latency as a benchmark of performance. Results of our evaluations are shown and described in the following section.

Evaluation

For our first application (banking app), we wanted to evaluate the effects of snapshotting as well as to explore system designs where Ray was not ideal. We first evaluated the effects of snapshotting. In our event sourced system, snapshotting is not required for fault-tolerance, but it does have a huge impact on read-latency as well as event-replay speed. Under this model, recovery will always replay the event log until the latest state, but snapshots can be used to facilitate this process. The actor has the ability to create a snapshot of its state at any given moment, and during recovery, the state will be loaded from the latest snapshot, after which the remaining events from the offset of the snapshot will be replayed. In this sense, snapshotting does not impact how "well" we can recover from a crash, but instead the overall performance.

For simplicity, we tested the effects of snapshotting only on our single-actor banking app since the results should carry over even to the multi-actor scenario. In our benchmark, we generated and ran tests which replayed an identical number of events at various snapshot frequencies, before recovering the current state and validating its accuracy. We plotted the mean latency vs. the snapshot interval we were testing, and also compared the results of vanilla Python vs. Ray [7].

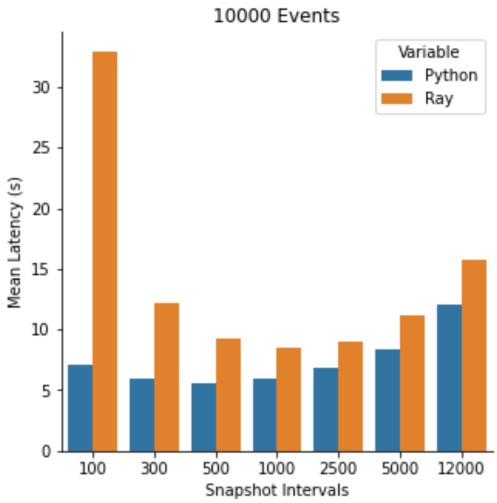

Fig 3. Plot of Latency vs Snapshot Intervals For Various Banking Application Implementations

There are several takeaways from this graph. First, it exhibits a v-shaped curve for both the Ray and vanilla Python implementations: performance initially improves (latency decreases) with increased snapshotting (due to the decrease in read and recovery time), but then decreases once snapshotting becomes too frequent (since snapshotting has considerable overhead). The results of this demonstrate that snapshot frequency should be tuned across different workloads for optimal latency. Second, we see that the vanilla Python implementation performs better than Ray in this single-actor situation, which is as expected due to the communication overhead in Ray. As such, for event-source systems which have a similar system design and do not have data sharded across multiple independent actors, we recommend against using Ray.

We also ran benchmark tests on the shipping application to evaluate the effects of data sharding. In these tests, we ran an identical number of events on the shipping app on various numbers of ships (actors), and plotted mean latency vs. number of actors [appendix 1].

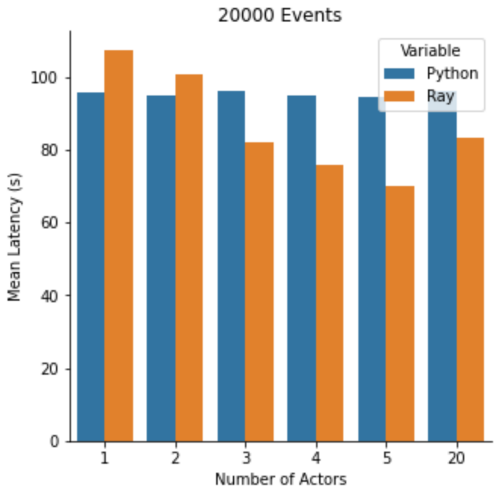

Fig 4. Plot of Latency vs Sharding Factor (No. of Actors) For Shipping Application

Similar to the snapshotting graph, we also observed a v-shaped graph, albeit only for the Ray implementation: near linear performance improvement was achieved with a greater number of actors initially; however, once the number of actors exceeds the number of cores, performance decreases due to scheduling overhead. It should be noted, though, that performance still remained better than for vanilla Python due to parallel processing of events up to the number of cores available. Ray performed worse than Python only in cases where there were only one or two actors (due to Ray's overhead, just as in the banking application's benchmark results). The vanilla Python's latency was relatively constant across different actor counts since the total number of events remain the same and they are processed sequentially on a single thread.

With these benchmarks, we are confident that Ray is indeed an ideal candidate for distributed event-sourced systems when data is sharded across multiple actors. However, just as described in the previous section, Ray's performance also results in feature-limitation since causal-consistency and actor persistent guarantees are not provided out-of-the-box. These challenges, however, are expected and should be addressed in higher-level application libraries, which we plan on developing in the future (more on this in the next section). In fact, many existing event-sourced libraries such as Akka also do not provide such features, and require even higher-level libraries such as Eventuate to provide such abstractions.

Future Work

In our project this semester, we've developed two complete backend systems, implementing event-sourcing with Ray actors. Through our work, we've learned a great deal about several common design patterns and event-sourcing building blocks common to different business use-cases, while also realizing some difficulties with building such systems on Ray. Despite some of these challenges, we still think Ray offers a much greater alternative than existing

solutions, due to its high performance and support for fault-tolerance and distribution. However, the current core Ray API itself is insufficient for building such an event-sourced system since higher-level semantics need to be implemented at application-levels.

Inspired by Akka and Eventuate and in order to fill these gaps in Ray, a future work we hope to explore is to implement a library on top of Ray (i.e. just like rllib [4] and tune [5] libraries for machine learning) which provides abstractions useful for building event-sourced systems (such as Persistent Actors, Log-Replay helpers, Command & Event Handlers, Actor Recovery etc). Due to the asynchronous nature of such actor-based systems, in our current design, it can be hard to achieve causal-consistency when merging multiple independent event streams across multiple actors. A future work to solve this issue is to implement CRDT abstractions as well as vector-clock-based causal consistency schemes into the library, such as those provided by Eventuate [1].

In addition, just as Akka provides an at-most-once (“fire-and-forget”) [3] approach to message delivery in order to decrease the burden and overhead on the core framework, actors in Ray do not provide message-delivery guarantees needed by event-sourced systems. This task is typically left to application developers to implement, in order to reduce overhead of the core framework. Even though in our design we persist all events in the log, messages in the actors’ mailboxes can still be lost as actors crash before processing and storing them in the events log. As such, another direction we hope to explore is implementing persistent actors in face of fault-tolerance.

Bibliography

[1] https://rbmhtechnology.github.io/eventuate/overview.html
[2] https://doc.akka.io/docs/akka/current/typed/guide/actors-intro.html
[3] https://doc.akka.io/docs/akka/current/general/message-delivery-reliability.html
[4] https://docs.ray.io/en/master/rllib.html
[5] https://docs.ray.io/en/master/tune/
[6] https://www.martinfowler.com/eaaDev/EventSourcing.html
[7] https://github.com/j5ik2o/akka-ddd-cqrs-es-example

Appendix

[1] https://github.com/YudiTan/cs262a-final-project/blob/main/event-sourced-banking-app/plots/Plots.ipynb
[2] https://github.com/YudiTan/cs262a-final-project/blob/main/event-sourced-shipping-app/plots/Plots.ipynb
[3] https://github.com/YudiTan/cs262a-final-project/tree/main/event-sourced-banking-app
[4] https://github.com/YudiTan/cs262a-final-project/tree/main/event-sourced-shipping-app

Artifacts

Our full implementation and benchmark code can be found at the following github project:
https://github.com/YudiTan/cs262a-final-project

Our highlights video and slide-deck can be found at the following link:
https://drive.google.com/drive/u/0/folders/1pb8XhQhujMIgcy9kbAfe2vuT3EqkZBN0