Transactions on Ray

Shubham Arora
Boston University
aroras@bu.edu

May 8, 2020

1 Abstract

We design and implement a transaction library on Ray[1], a general-purpose cluster computing framework. The transaction library provides a high level API, and is very easy to use. The Ray framework supports stateful and stateless computation models as Actors and Tasks. Actors are a model of concurrent computation. Actors have previously been used for building software using microservices architecture. Microservices architecture requires high-performance (low latency/high throughput), fault tolerance, elasticity, and load-balancing features among others. We focus on the microservices use case for transactions. We implement and evaluate Distributed Sagas, an extension of the Saga pattern for coordinating transactions on microservices.

2 Introduction

Transactions in databases are a unit of work. Transactions are expected to have ACID properties. A transaction can consist of multiple operations. By definition, a database transaction must be atomic, consistent, isolated and durable. All these properties are hard to achieve together in distributed systems. These challenges have been well studied, e.g. in : [2], and many leading cloud systems provide limited transaction support e.g. [3] within a single row, or no transaction support at all.

In real world applications, transactions come up often. Take the example of a trip booking service, that lets user book flights and car for a trip. It is desirable in such a case to process user payment only after bookings for car and flights have been successfully confirmed. This can be expressed as a transaction. We discuss this further in Section: Design and Implementation.

Microservices is a popular way of writing the middle tier in a standard 3-tier architecure. The other tiers being a stateless front end and a storage layer. We believe Ray's programming model is very flexible, to allow building applications as a set of microservices. We discuss more about this in the background section on ray. Section : .

Finally, we introduce Sagas, as a way of coordinating transactions on microservices. We describe the implementation of Sagas for Ray in Section . Finally, Section covers results. Section is the conclusion.

3 Background

3.1 Ray

Ray is a framework for general purpose cluster computing. It implements a distributed scheduler and fault-tolerant store to manage the systems control state. The unified interface of Ray can express both actor-based and task-parallel computations. It was built generally to support AI applications, more specifically it has core libraries dedicated to support Reinforcement Learning workloads.

Ray has a very easy to use API. Any python class can be annotated with a *ray.remote* to convert it into an actor.

```
@ray.remote
class Counter(object):
    def __init__(self):
        self.value = 0

    def inc(self):
        self.value += 1
        return self.value
```

This actor can then be instantiated and methods can be run in the actor as:

```
c = Counter.remote()
id = c.inc.remote()
counter_value = ray.get(id)
```

The *id* returned by any $methodName.remote()$ call is a Ray.ObjectId. This is called a future. It can be resolved using a $ray.get(...)$ command. This command takes an objectId, or a list of objectIds and returns their resolved value. All method calls of the form $.remote()$ are non-blocking. $ray.get()$ here is a blocking call.

The actor model is a conceptual model of concurrent computation. An actor is the primitive unit of computation, parallel to an object in a Object-Oriented language. Actors communicate with each other via messages. Actors can only modify their private state, but can only affect other actors indirectly through messaging. In response to a message received, an actor can make local decisions, create more actors, send more messages, and determine the response to the next message. The actor model was first introduced in 1973[4]. Many concurrent and distributed systems are built on top of the actor model [5] [6].

Naturally, the actor model lends itself to the design of microservices. Akka and Orleans have famously been used in microservice architectures [7]. For simplicity, we assume an actor to be an approximation of a small microservice.

Ray provides a very easy way to create actors in Python. Any python class can be annotated with $ray.remote$, and the actor class will be automatically run as a process in the cluster using $actor = ClassName.remote()$.

3.2 Sagas

Sagas are long lived transactions (LLTs)[4]. A LLT holds on to database services for a relatively long periods of time, affecting other shorter transactions. An LLT is a saga if it can be written as a sequence of transactions that can be interleaved with other transactions. The system supporting saga must provide the guarantee that either all transactions in a saga are successfully completed or compensating transactions have been run to amend the result of partial execution. The Saga pattern discusses a way of orchestrating sagas. It guarantees consistency for a database system.

Consider a trip booking application. The application allows users to book a flight from a source to destination, a car from the users home to the source airport and a hotel at the destination. After choosing a flight and car, the user performs payment. The user must only pay, if the flight, car and hotel bookings are all confirmed.

Lets call this a long lived transaction T.

Let a long lived transaction T be composed of sub-transactions $T1, T2, ...T_n$. Each of these sub-transactions must be real transactions in the sense that they preserve database consistency. We want the saga system to process T as a single transaction. i.e. either T is successfully completed or not done at all. It is not acceptable that only one or two of these transaction completes and the rest fail. It is acceptable however, that would rollback all the committed transactions if a sub-transaction failed.

To handle partial execution, each sub-transaction T_i should provide a compensating transaction C_i. The compensating transaction must semantically undo the effects of actions performed by T_i. For example, in our trip booking example, if T_i performs a flight reservation, C_i can cancel that reservation(say by adding a cancellation entry in the flight database, and increasing the seats available count). This may not revert the flight database to the exact state that existed before T_i ran. However, the flight database remains semantically consistent. Another example would be sending out an email for confirmation. It is not possible to undo an email send, however you can send a follow up email that compensates for the previous email.

Now that we have defined compensating transactions $S1, S2, ...S_n$ for the saga $T1, T2, ...T_n$, the system can make the following guarantees. Either the sequence $T1, T2, ...T_n$ is completely run, or a set of transactions T_i and their compensating transactions C_i were completely run.

Note that other transactions might see the effects of a partial saga execution.

3.3 Microservices on Ray

Microservice architectures require high-performance (low latency / high throughput), fault-tolerance, elasticity, and load-balancing, among others. Many data flow systems provide these features "for free" but Ray has more advantages.

1. Microservices require "*loose coupling*"

   Essentially the ability to build and test individual microservices separately. In systems like Spark[8], Naiad[9], Flink, etc. data flow operators cannot be used as standalone programs - there is no input/output API exposed to the user. Instead, users often need to build the whole data flow or at least parts of it to feed operators with input data and test their implementations. In Ray, it is super easy to test standalone operators, as they are essentially Python classes with a list of methods.

2. Synchronous and asynchronous communication

   Ray's programming model enables both modes, e.g. using features with $ray.get()$ and $ray.wait()$

3. Task-and data-parallelism

   Ray supports both task-and data-parallelism, which enable users to express more complex computations in a much more intuitive way than pure data-parallel systems.

4. Easy access to state

   Microservices have state, which we often need to query online. This is easy in Ray's programming models via RPCs to the corresponding actors. For example, a microservice actor could expose an RPC that returns its current state (or parts of it).

4 Design and Implementation

Here we implement the trip booking application that we introduced before. The flight, car and hotel booking service are handled by three independent services: $FlightBooker$, $CarBooker$ and $HotelBooker$. Payment is handled by the $PaymentService$. The user interacts with a trip booking service $TripService$. $TripService$ is responsible for interacting with $FlightBooker$, $CarBooker$ and $HotelBooker$ and $PaymentService$. We can model this as a saga as shown in Figure 1

The services $FlightBooker$, $CarBooker$ and $HotelBooker$ shown in the figure are all implemented as Ray's actors. The state of the services e.g. flight database, car database is managed by the respective actor as class object variables. All methods on these actors, such as $book_trip()$ and $get_trip_options()$ operate on these variables.

Note that the Flight, Car and hotel booking can all occur concurrently, and the Payment API must only be called after all the three previous bookings have

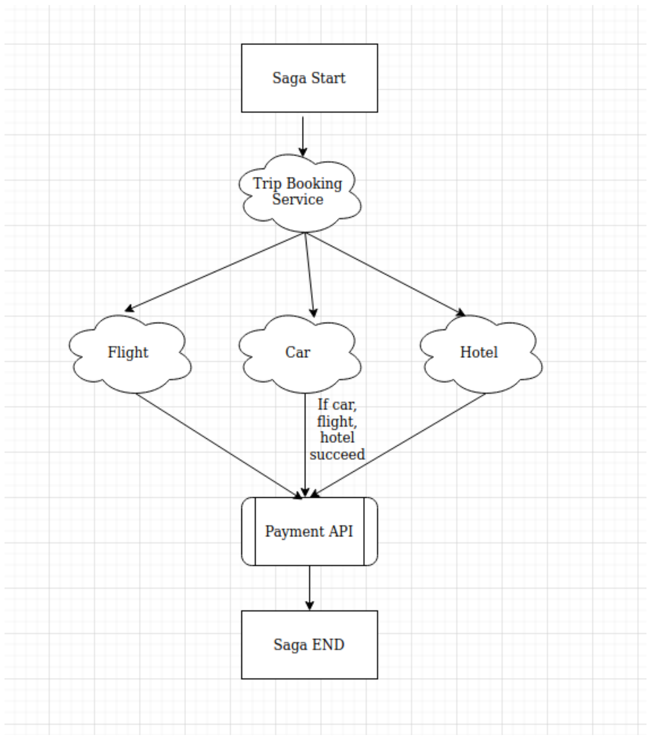

Figure 1: Trip booking service modeled as a Saga

been successful. This dependency can me modeled as a Directed Acyclic Graph (DAG). Let's now describe now the system that implements the saga system, and how it works with the Ray API.

4.1 User API

The saga transaction library that we have implemented for Ray provides a straightforward API. Sagas are described between *saga.start*() and *saga.commit*() statements. The example of using the saga library is given below:

```
saga = Saga.remote()
saga.start.remote()

txn1 = Transaction(service1.action1, arg1)
rollback_txn1= Transaction(service1.rollback1, rollbackArg1)
txn2 = Transaction(service2.action2, arg2)
rollback_txn2 = Transaction(service2.rollback2, rollbackArg2)

saga.add.remote(txn1, rollback_txn1, name="txn1")
saga.add.remote(txn2, rollback_txn2, name="txn2")

ray.get(saga.commit.remote())
```

4.2 Saga Orchestration

Now, we need a mechanism to orchestrate the transaction in a Saga. For this, we use the Saga orchestration pattern. This pattern has been well discussed. [10] [11] . We will describe here our implementation. Whenever we want to execute a Saga, we create a process, the Saga Execution Coordinator (SEC). This coordinator executes the transactions in the sagas, parses the results, and if a reservation or payment fails, executes the compensating transaction for all transactions that have completed.

The SEC is created using a *Saga.remote*() command whenever we start a saga. A *saga.add.remote*() command creates a node in the Saga DAG. The Saga is executed by the *saga.commit.remote*(). The Saga is then executed as flows as follows. The saga follows the transaction nodes in the DAG. For each transaction nodes, the SEC adds an entry to its log. We will call this the Saga Log. Note here, that the SEC can be stateless process running anywhere on the cluster, and the Saga Log is a persistent log. Hence, the state of a transaction can be recovered even if the SEC process gets killed. Each action that the SEC takes, or each message it receives is hence logged.

Now, after writing to the log, the Saga executes the transaction represented by the node in the graph. Each start of a transaction is logged by a "Start" statement. In our example, each such transaction is a service API call. We assume that the following responses are possible: 1. We receive a success response (reservation complete), 2. We receive a failure response (reservation could not complete), 3. No response (The service may not be responding due to any reason). We discuss here only the cases where services are guaranteed to respond.

When the service responds with success, the SEC logs the response to the Saga Log with an "End" statement. If all the transactions in a saga completes successfully, the saga is considered complete. In the case of a failure response, the SEC logs a "Abort" command. The normal execution is paused. The SEC then reverses our execution DAG. We must now compensate for the transactions that have completed. The SEC now starts executing compensating transactions for the reversed DAG. For each node in the DAG, the SEC queries the log, to check if the transaction status for that node. i.e. is there, a "Start", "End" or "Abort" command for that transaction.

If the transaction has no entry, mark it compensated and move to the next node. If the transaction has both a "Start" and "End" command, the transaction ran successfully, and we must now execute the compensating transaction for this transaction. This is done similarly, by first adding an entry to the Saga Log, and then after the compensating transaction completes. If a transaction has both a "Start" and "Abort" entry, the transaction did not go through and we do not need to compensate. If a transaction has only "Start" command in the log, we are unsure if the transaction may have completed at the service or not. We may have not received a response for any reason. In that case, we can not assume anything. We retry the transaction. When the transaction commits, we log an "End" command to the log, and immediately execute its compensating transaction.

When we complete executing the reverse DAG, the saga execution ensures that the database is in consistent state.

5 Experiment and Results

We conducted the following experiments to measure the throughput of our system. These were run on a machine with the following specs: quad core Intel Core i7-8550 CPU, 16 GB RAM running Ubuntu 18.04 LTS. We use Ray version 0.8.4 and Ray started with 5.62 GiB memory available for workers and up to 2.82 GiB for objects.

Transactions were wrapped between *saga.start*() and *saga.commit*() statements. To measure the throughput, we submitted a varying number of transactions, and measured the time taken between the *saga.start*() and *saga.commit*() statements. These time measurements were summed up and averaged over the number of transactions.

Figure 2 shows the results for a single saga coordinator for all the transactions. It averages around 150 tps. Figure 3 shows the results for a multi saga coordinator scenario, where a new coordinator actor is scheduled for each transactions. It averages only around 5 tps.

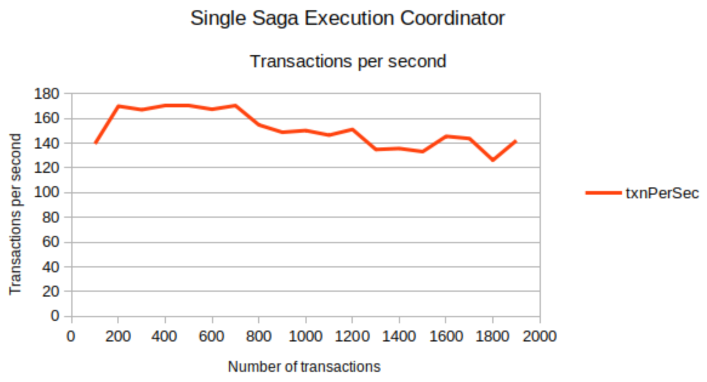

Figure 2: A single saga execution coordinator

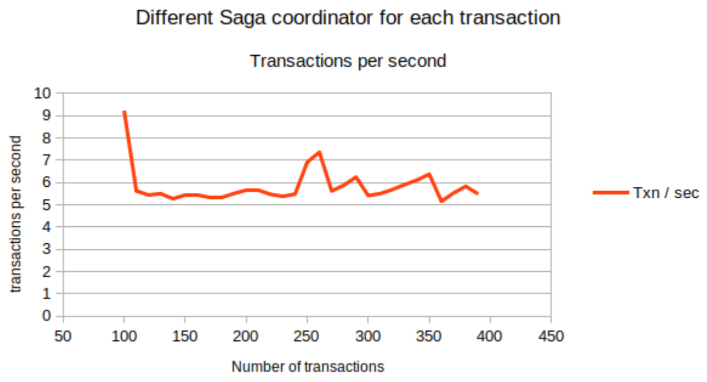

Figure 3: Multiple saga execution coordinator

6 Conclusion and Future work

In this paper we study the Saga execution pattern for orchestrating transactions in Ray. We do an experimental analysis of our implementation, and get a throughput of 150 tps. We also implemented microservices on top of Ray actors. We find this to be a promising result. We show that Ray actors can be used to develop and scale microservices, and the saga pattern can be effective to coordinate transactions among microservices. We also conclude that it is better to have a single Saga execution coordinator that manages transactions for each machine. Since the number of actors that can be spawned on a machine depends on a CPU core, and scheduling and killing actors takes time, this can be a bottleneck.

In the future, more work can be done with complicated services e.g. services that store their state in an external database or key value store. Other approaches other than saga pattern can also be evaluated. We can also make measurements for the throughput on a multi machine cluster and find what configuration of Saga executor coordinator performs best.

References

[1] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. *CoRR*, abs/1712.05889, 2017.

[2] Pat Helland. Life beyond distributed transactions. *Queue*, 14(5):69–98, October 2016.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), June 2008.

[4] Hector Garcia-Molina and Kenneth Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, December 1987.

[5] Jonas Bonér. Akka: Build powerful reactive, concurrent, and distributed applications more easily. https://akka.io/.

[6] Sergey Bykov, Alan S. Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and J. Thelin. Orleans: A framework for cloud computing. 2010.

[7] Paypal. Akka streams akka http for large-scale production deployments. https://paypal.github.io/squbs/.

[8] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker,

and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.

[9] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery.

[10] Caitie McCaffrey. Distributed sagas: A protocol for coordinating microservices. https://www.youtube.com/watch?v=0utolrtwox0.

[11] Chris Richardson. Pattern: Saga. https://microservices.io/patterns/data/saga.html.