Uber – system overview

Amitai Frey – 205588726 Ronen Sandler – 313605321

# Introduction

In this exercise, we were tasked to plan and implement a system for a distributed ride sharing service. This system raises certain challenges that we have learned about during the semester. These challenges include replication, consensus, total ordering, atomic broadcast, leadership & membership, and others. In this document we will list our design choices and explain them, while also describing our implementation – its advantages and disadvantages.

# design overview

First, we will start by explaining some of the terms we use in this document:

* Server – a single computer (VPS/VM/Docker Instance/others) that runs our application.
* Shard – a group of servers that are responsible for a certain city.
* Leader – a designated leader for a given shard. Its responsibilities will be explained later.

We must note here that a server can belong to several different shards, meaning it takes part in multiple cities’ decisions and responsibilities.

## Leadership, membership & replication

We decided to use the leadership pattern to implement total ordering and replication for each shard. The leader is responsible for all the actions, she receives the methods that need to be done and is responsible for their processing. This means she checks their validity, does the actions, and broadcasts the actions to the other members of the shard. Finally, she returns the result to the client/server that requested it.

To deal with crashes and failures, we implemented the broadcast in a fail-safe way so that the action is either done by the whole shard or by none of the valid servers in it. This is done by having the members broadcast the action as well, resulting in multiple broadcasts of the action across the shard. The order of operations if all the servers are valid is as follows:

1. Leader gets the request.
2. Leader checks its validity.
3. Leader performs the action.
4. Leader broadcasts the action to all members, waiting for each one to answer.
   1. Members check if they have already performed this action, and if so return.
   2. Members perform the action.
   3. Members broadcast the actions as well.
   4. Members return their result.
5. Leader finished the process and returns the result.

As we can see, in the case where all the servers are valid this guarantees that all the servers in the shard perform the action before returning the result. We also added a lock in the leader, so it only processes one request at a time. Additionally, members that receive requests forward them to the leader, thus maintaining the total order of actions.

We will now demonstrate two different cases of failures, and how our algorithm handles them gracefully and in a valid manner. The first example is if the leader crashes while processing a request, before broadcasting:

1. Leader gets the request.
2. Leader checks its validity.
3. Leader performs the action.
4. Leader crashes (!)

In this case, the request will never be answered, and the client/server will have to retry. We must note the shard’s state is still valid, because in all the valid servers the state is the same.

In the second example, the leader crashed in the middle of broadcasting:

1. Leader gets the request.
2. Leader checks its validity.
3. Leader performs the action.
4. Leader broadcasts the action to all members, waiting for each one to answer.
   1. Leader crashes (!)

In this scenario, we cannot guarantee that the broadcast was successful. However, we are faced with two options:

1. The first broadcast was not successful. This case is like the previous example, and the state will remain the same.
2. At least the first broadcast was successful. In this case, the first member performed the action and broadcasted it to the others. We guarantee to always broadcast in order of potential leadership, meaning that if we crash - the first server to whom we broadcasted will become the leader, and the state will remain valid because of its lock. (This guarantee’s implementation will be expanded upon later). This logic also follows if the first member crashes as well, we are faced with two options w.r.t the second member etc.

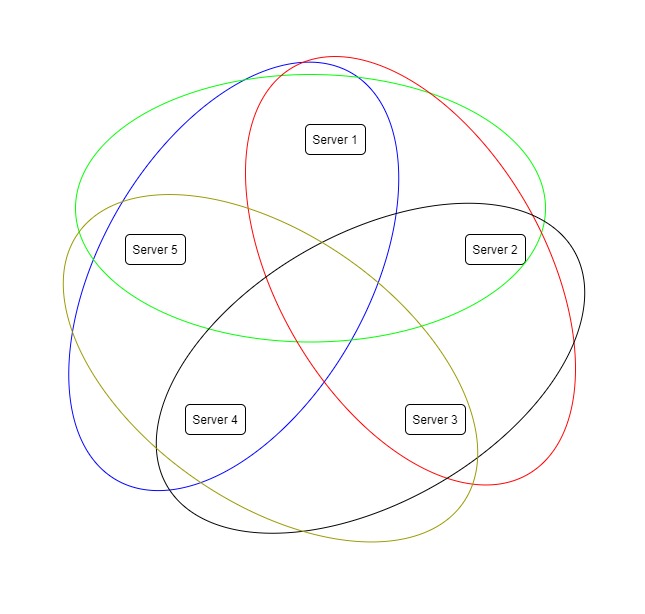
The implementation we detailed above describes how all simple actions are handled in our application. This includes new ride offers, getting city snapshots and locking/committing rides. Handling ride plans is explained later in the document.

## Zookeeper usage

To implement the leadership and membership we mentioned above, we used zookeeper as a leadership & membership service. The leadership election implementation is based on the recipe on zookeeper’s website[[1]](#footnote-1), and works as follows: At startup, each server connects to the zookeeper service and creates an ephemeral\_sequential node in: /<city\_name>/ for every city shard it belongs to and inserts the server’s IP address as the data of that node. The leader is the node with the lowest node number. Thus, if a leader crashes its node will be removed (because it is ephemeral), and a new leader will be “elected”. When broadcasting, the leader will broadcast by order of node numbers, to ensure broadcasting according to leadership priority as detailed above.

This solution also provides the membership service. To know a shard’s members, we can simply get all the nodes under /<city\_name>/ and read their IP addresses. This also allows servers to join the service while the service is already running, and not only at startup. This was not a requirement for the exercise but keep this in mind when you read the explanation of dynamic joins of servers.

This is a diagram of a potential state of zookeeper, with leader and members:



Zookeeper state:

* /city1
  + 1 – Server1.IP
  + 2 – Server2.IP
  + 3 – Server3.IP
* /city2
  + 1 – Server2.IP
  + 2 – Server3.IP
  + 3 – Server4.IP
* /city3
  + 1 – Server3.IP
  + 2 – Server4.IP
  + 3 – Server5.IP
* /city4
  + 1 – Server4.IP
  + 2 – Server5.IP
  + 3 – Server1.IP
* /city5
  + 1 – Server5.IP
  + 2 – Server1.IP
  + 3 – Server2.IP

We decided to implement the shards in this method (like a Venn diagram) to balance loads, provide a more robust service and allow handling more shards with less cities. For example, if a shard was responsible for all the same cities, then providing 5 shards with 3 servers in each shard and each server being a member of at most 3 shards would require at least 6 servers – and our implementation requires only 5. Additionally, if two servers crash in a standard replication group that has 3 servers, one server will now have to handle multiple cities’ information by itself. However, in our solution it will be solely responsible for only one city and share the responsibility of its other cities with other live servers.

Of course, this is all configurable and we can configure standard replication groups as well.

We would like to also note here that we use zookeeper with only very lightweight data, and only for leadership & membership. We decided to do keep the zookeeper as light as possible to improve performance as we discussed in a reception hour.

## Handling ride plans

One of the major parts of this service is providing ride plans for clients. This means a client can request a path of cities she wants to travel during a specific day, and our service should try fulfilling this plan if possible. This may require assigning multiple rides from different shards for the same plan. Since no server is guaranteed to be a part of all shards, let alone a leader of all of them, providing this feature poses a distributed systems challenge.

We decided to solve this challenge using 2-phase commits. Suppose a client requests a ride plan from a certain server, the server acts as follows:

1. Requests from all leaders to send him potential rides for the plan.
2. Decided if the plan can be fulfilled, and if so with what rides. If it cannot be fulfilled, return a negative result.
3. Send to the leaders of all shards that are responsible for the wanted rides a request to lock the rides for this plan. If (at least) one of the lock requests was not successful, return a negative result.
4. After all the rides are locked, request all the leaders to commit this new plan to these rides.
5. After the commits have been processed, return the result to the client.

Since we are not guaranteed the server that is processing this plan will not crash, we need to discuss what happens if it crashes. We decided to implement the ride locks with a timeout. This will guarantee that if the server crashes while handling a plan, it will not lock the rides forever but only for a short time which will be enough for it to commit the ride. If it did not commit or crashed, they will be released and will be usable again.

Additionally, we implemented a retry and timeout service for the requests. This ensures that if the handling server did not get a response from a leader, it will try again and perhaps get a new leader (since the old one probably crashed). This new leader will then process the request again, and the state is guaranteed to be valid as we discussed above.

We have also addressed the problem where multiple rides can lock each other out. This can happen if two plans require rides A and B, plan 1 locks ride A and then plan 2 locks ride B. Since now both cannot lock their second ride, they will stop and return a negative result. This can happen multiple times and can be frustrating for the user. We solved this problem by implementing the Dining Philosophers Problem[[2]](#footnote-2) solution as proposed by E. Dijkstra, in which we order the locks by the ride’s ID and prevent this from happening.

## bonus - dynamic membership

As we explained earlier, by implementing membership using zookeeper ephemeral\_sequential nodes we can add a feature that servers can join a shard dynamically and not only at startup. This can be helpful when shards become too overloaded or servers that have crashed have been rebooted. To do that, a server must:

1. Get the shard leader.
2. Request the machines state from him.
3. The leader will send the state and wait for the new server to join.
4. The new server will change to the given state and add itself to the zookeeper membership.
5. The leader finishes waiting and continues to process new requests.

As we can see, joining a shard dynamically is not complicated and only requires the leader to be able to send its current state and wait for a server to join. We implemented this using two new GRPC methods, and by saving the inner state as a Serializable object, which helps sending it over the wire.

As we stated above, this feature is a bonus and was not required. However, it was a fairly simple addition to our system because our system’s design was fit to include it easily.

## docker & docker-compose

We decided to test our application using Docker, because it is easy to use and doesn’t take a lot of system resources. We did this by build our docker image on top of the existing zookeeper image, because it already has zookeeper and Java. This image simply includes the Java application as well as configurations for all the servers. We use the docker-compose utility to deploy multiple docker instances at once, while configuring each one with its specific configuration file. The docker instances externalize a GRPC port in their local network and a REST port for clients to access, which we forward to different ports on our machine for testing purposes.

Originally, we planned to redirect REST requests from clients to non-leader members to the respective leaders using 3XX codes a s specified in the HTTP protocol[[3]](#footnote-3). However, running dockers without them knowing the ports we forward in the docker service rendered this solution impossible, so we had to settle for sending the requests internally to the leader by GRPC.

## interfaces

We have implemented the REST API to include 3 methods which the client can use as required by the specification. To handle inner function calls and replication, we implemented multiple GRPC methods for in-shard and cross-shard usage, including methods for new rides, ride locks, ride commits, getting city snapshots and getting a shard’s state (for dynamic membership). The full interfaces and implementation can be seen in our code.

1. https://zookeeper.apache.org/doc/current/recipes.html [↑](#footnote-ref-1)
2. https://en.wikipedia.org/wiki/Dining\_philosophers\_problem [↑](#footnote-ref-2)
3. https://tools.ietf.org/html/rfc2616 [↑](#footnote-ref-3)