**Description of our system:**

Shards are a partition of cities, i.e. each city is in exactly one shard, and no shard is empty.

Each server can contain multiple shards, and it can be leader on some, and follower on others.

When the system starts, each server connects himself to the following protocols for the rest of the run: Leader Elections, Failure Detection and Atomic Broadcast. Their implementation is described below.

Each server starts a GRPC server and REST server.

Each server saves the following fields in his local memory:

* A set of all the cities in the system.
* A map from a shard to the leader of this shard: This map is always in a consistent state, we update it every time we receive an election event (it happens when a leader fails). ZK guarantees that we won’t miss such an event (even if there were two leader elections in parallel), we will describe this in more details later.

We use this so that each server would know where to redirect a request that he received that wasn’t for him, he will directly contact the leader and not take a path of redirections until he finds him. This is faster in terms of less network passages between servers, and the probability of one of the two servers that are taking care of the request (the one redirecting, and the leader), is lower that the probability of a server falling in a path of redirections.

* All of the servers in the system: This is needed in order to know the addresses of all the servers for communication purposes.
* Primary shards: the shards for which this server is leader.
* Shards: the shards that he is either a leader or a follower for.
* A list of all the rides that their start city is in a shard that resides in this server.

Note: Each server, when reading/writing on the rides list, the reads and writes are synchronized locally.

In ZooKeeper we implemented the following method: Leader Election, Failure Detection, Atomic Broadcats, Global Lock and Unique Id Generation, we will describe them below.

We implement three REST methods: publishRide, reserveRide, snapshot.

* publishRide:

A server s1 receives a publishRide request along with the info about the publisher and the ride.  
s1 now checks:

* + If he is the leader for the shard that contains the start city of the given ride, then he will add the ride to his list of rides and the atomically broadcasts this ride to the followers (of the shard of the start city of this ride). The atomic broadcast here is needed in order to prevent a case where s1 sends the ride to some followers but dies before sending it to the rest, if such a case happens then after a leader election there should be a recovery phase in which the servers would consult each other in order to get to the consistent state. In our approach no recovery phase is needed, either all of the followers get the addition request, or none get it. In our approach we will get to a consistent state faster that the other approach, and in case the leader dies and no response is returned, the frontend application would retry the request.
  + If he is not the leader, then he would find the leader from the map that he saves locally from a shard to its leader – s2 for example, and forward the request to him in a GRPC call, s2 would save the ride and atomically broadcast it to the relevant followers, and if successful will return the ride that he reserved to s1, and s1 will return it back to the client (if s2 failed while handling the request then again frontend app should retry).

Before a leader server decides that he wants to reserve a ride, he first checks if he has this ride already, and only if not, he will generate a unique ID (we will describe this later, the ID is unique across the whole system).

The main part that we wanted to guarantee is that the system stays in a consistent state, a ride is either broadcasted to all relevant servers or to none.

This method is idempotent, invoking it multiple times with the same ride info will not add the ride multiple times, the ride will be added only once (even if someone already reserved the available ride in the system).

* reserveRide:

We divide into two cases:

* + s1 receives a request to reserve only one ride, i.e. the path that he got to reserve was of length 2.

s1 iterates through all of the leaders in the system (including himself if he is one), and invokes a blocking GRPC method on each one, telling him to search for a suitable ride for a given reservation request (with path length of 2) among the rides that this leader is the leader for the shard that contains the start city of the ride, and reserve it if he found one, otherwise s1 continues to the next leader.

The iteration over all of the leaders is to check if we can find a ride from a neighbor city that could pick the consumer up.

When a leader is searching among his rides for a suitable ride, he does that in a synchronized manner, meaning that he wouldn’t accept to reserve two concurrent request if he can satisfy only one of them, i.e. searching for a suitable ride and reserving one if found is a critical section that only one thread enters at a time.

When a leader finds a suitable ride and reserves it, he also atomically broadcasts the reservation to his followers. The broadcast is done atomically for the same reasons in publishRide above, we want to either let all the followers know of the reservation, or none of them, we don’t want to enter a recovery phase in case the leader dies before broadcastin the message.

Furthermore, The code that takes care of such a request for a single ride is an idempotent code, meaning one can request to reserve the same ride a couple of times, the ride would be reserved only once if it was found suitable, we take no action in such case and return that the ride was successfully reserved.

However, it is possible to reserve two different rides for the same reservation

SHOULD WE REMOVE THIS??

* + s1 receives a request to reserve a path, i.e. the path that he got was of length larger than 2.
  + First of all s1 will compete on the global lock that we describe below.

Once s1 acquires the lock, he initiates two phase protocol with the rest of the leaders in the system:

* + - s1 first asks all of the leaders to lock themselves because this is a critical phase and we don’t want other servers making reservations on rides that we are working on.
    - Afterwards, s1 asks all the leaders to send him the rides that are suitable for the given reservation (a ride is suitable if it can satisfy at least one tremp in the path), the leader will send only the rides that he is leader for the shard of the start city of the ride.

Note, s1 invokes these GRPCs on those leaders asynchronously and not blocking.  
the leaders return the rides streaming them ride by ride.

* + - Once s1 gets all the relevant rides, he first checks which leaders didn’t send him rides because they didn’t find any suitable ones and tells them to release the lock from themselves because he doesn’t need them, and we want to continue their work regularly.
    - s1 releases the global lock.
    - s1 starts a backtracking method to find rides to satisfy all of the path.
    - If suitable rides were found, s1 will atomically broadcast to all of the relevant leaders and their followers to reserve these rides. The atomic broadcast here is for the same reasons as we mentioned above.
    - After that s1 tells the leader that he is holding to release themselves from the lock, and then return the answer to client (the rides if successfully reserved, or failure if he didn’t find suitable rides.

Note: if s1 falls during this process then he would release the global lock, and the other leaders that were locked will release themselves because they are notified that s1 died. Also if one of the leaders dies, s1 would know not to wait for his response because he also will be notified (onError in GRPC…).

* snapshot:

A server s1 receives a snapshot request.

s1 will find all of the leaders in the system (again from the map from shard to leader that he has), and them a GRPC request to tell the leaders to send rides. A server s2 that receives this request will only return the rides that he is leader for the shard of their start city, and not all of the rides that he has.

s1 will invoke the GRPC request asynchronously on all of the leaders (if he is a leader too then he will take his rides – the one he is leader for).

Afterwards, s1 will wait (for a bounded time) until he receives all of the responses, and then returns them.

We use an async call so that all of the servers will start calculating their rides in parallel, furthermore, we let each server send us only the rides that he is leader for, and with this s1 wouldn’t need to do anymore processing, except of collecting and responding to the client. So the hard work is divided between the leaders and not centralized in s1.

Note that snapshot is not blocked by reserveRide.

Snapshot returns all of the ride in the system, and for each ride the names of the reservations on this ride.

Leader Election:

We implement this under the directory /Uber/elections in ZK:

When starting the system, given the shards that we defined, we create the following children in this directory: /Uber/elections/shard-1, /Uber/elections/shard-2… for each shard.

Each server that wakes up, adds an ephemeral sequential node under the shards that he is responsible for and competing on to be the leader. The server also adds a watch on the path for the elections directory /Uber/elections.

Under each path for a shard, the server with the node with the lowest sequential number is the leader for that shard, and all of the servers in the system know that he is leader (they should know because if they receive a request that should be redirected, they would know to whom to send it.

When an event is triggered under the directory /Uber/elections, all of the servers update their knowledge about who the leader is for each shard.

Failure Detection:

We implement this under the directory /Uber/active (is everyone triggered when someone dies, or only the followers/leaders of a specific group?)

Atomic Broadcast:

We implement this under the directory /Uber/mailbox

Global Lock:  
We implement the global lock under the directory /Uber/G-Lock in ZK:

A server s1 who wants to acquire the lock would:

* insert and ephemeral sequential znode to a directory /Uber/G-Lock.
* s1 then checks, if he is the child of /Uber/G-Lock with the lowest sequential number, then he acquires the lock and continues his work.
* Otherwise, he finds the child with the largest sequential number that is lower than him, and adds a watch on him. We don’t add a watch on the parent /Uber/G-Lock in order to avoid the herd effect and to avoid getting unnecessary event triggers.
* After wards he waits on a CountDownLatch(1), the latch is counted down when the watch that s1 set is triggered.
* Once s1 is released from the wait on the latch, he does this process again starting from bullet (2), because the one he was watching might have died, and this doesn’t mean that s1 can acquire the lock.
* If s1 succeeds to be the child with the lowest sequential number then he acquires the lock, otherwise waits again on the latch.

Unique Id Generation:

This is done under the path /Uber/id-generator in ZK.

For each ride added in our system, we generate a unique ID for this ride, the ID is unique system wide.  
We generate these unique IDs using ZooKeeper’s Sequential nodes, so each time a ride is published, before adding the ride we create a sequential ephemeral Znode in ZK and get its name as an ID. Of course this limits the number of unique IDs that we can generate because a sequential node number in ZK is of limited length `n`. A simple solution would be to create two persistent Znodes when starting the system, and we would join the numbers from both Znodes into one number with length `2n` so when the first Znode overflows, we increment the second Znode and start counting again until overflowing again.

To run the system:

* You should have docker installed on your host machine.
* Setup your configurations in the file ds-hw2/build-scripts/input.txt:

The first number is the number of servers that you want in your system, and under it write your cities and their coordinates. Each city in a new line, and each line is comma separated.

* Run `python3 ds-hw2/build-scripts/uberInit.py`:

This will create a network bridge called “uber-network” using subnet address 172.18.0.0/16.

All of the bellow containers will run on this bridge.  
A ZK docker container will be run.

Afterwards a cleaning process will run to initiate our directories in ZK.

The servers will run each in a docker container.

And lastly a comfortable UI will run on your host machine to help you interact with the system.

Using the UI:

* To publish a ride go to the tab `Publish Ride` and insert the info about the ride.

After that, choose whether you want to send the request to a random server or not, if not choose your desired server in the list under the `Random server` box.

When finished click on `Publish` and you should see the response of the server on the right panel.

* To reserve a ride, go to the tab `Reserve Ride` and type the info of your reservation, the path is a comma separated field with the names of the cities in your path in right order of the path.  
  After that again decide to which server you want to send the request and click on `Reserve`.
* To get a snapshot choose the server you want to request it from and click on the `snapshot` button.
* To clear the right panel, click on `Clear log` beneath it.

Graphical user interface

Description automatically generated

Graphical user interface, application

Description automatically generatedGraphical user interface, application, Word

Description automatically generated