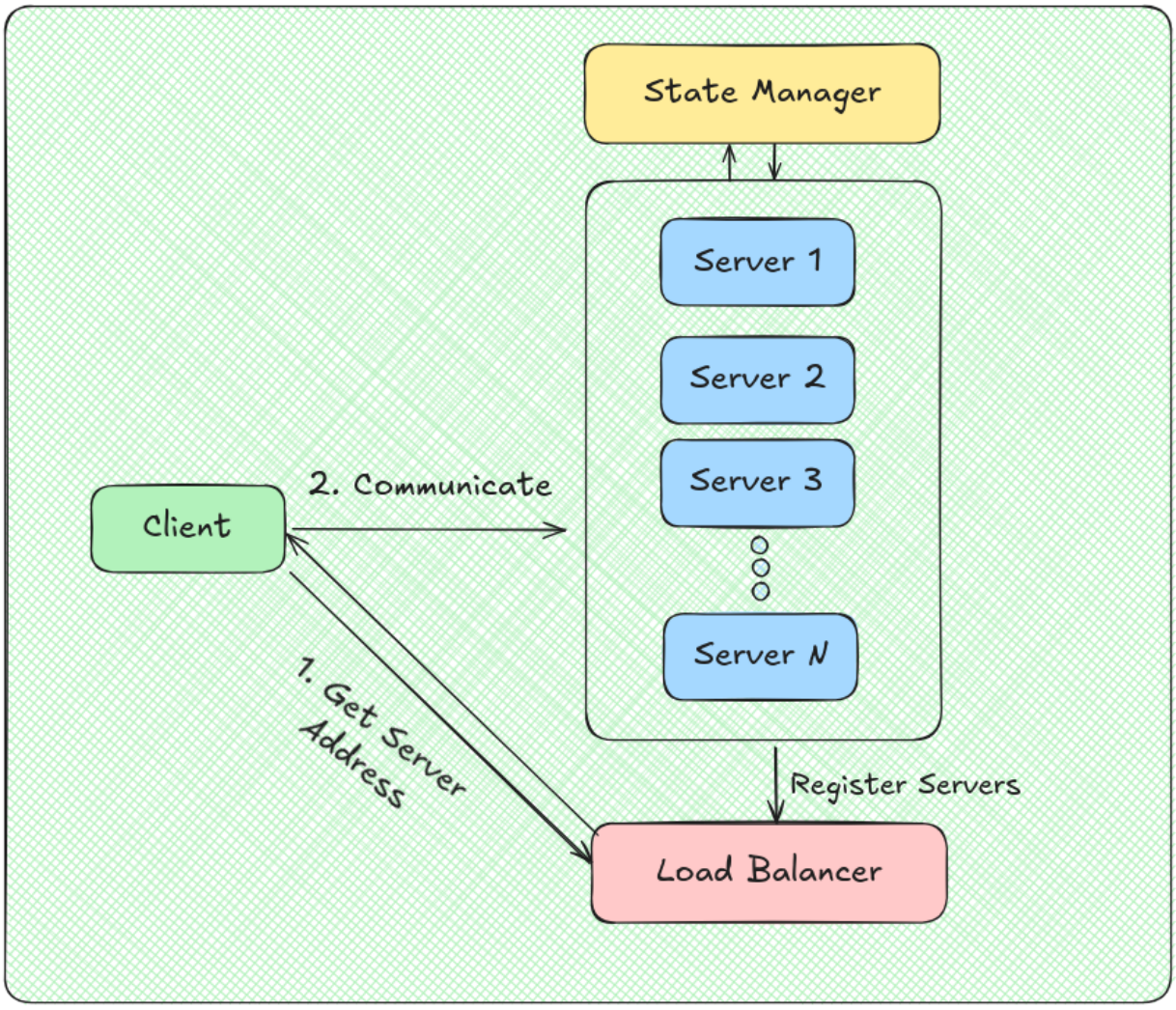
# MyUber-gRPC

## System Overview



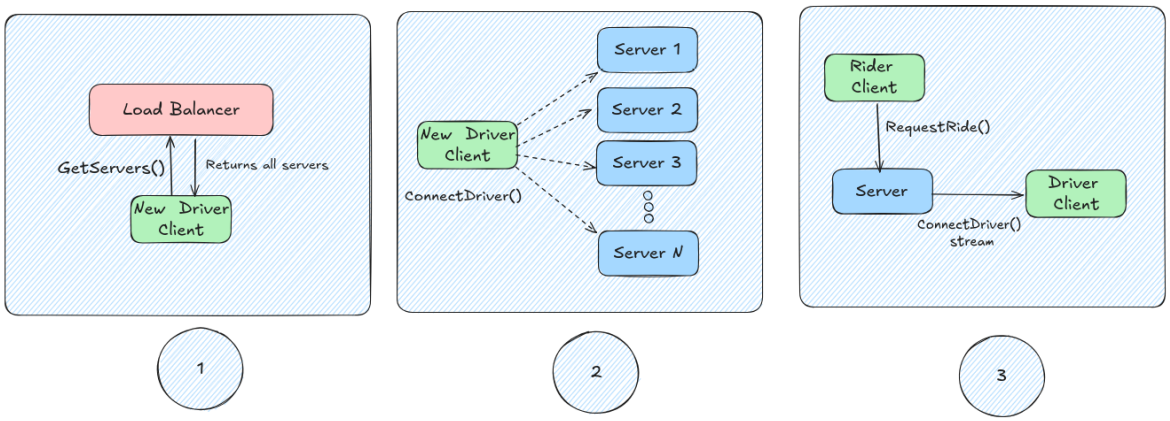

Overview of the system show-casing communications

We have a server cluster, which has a central `State Manager`. The state manager holds the information of all the ongoing rides. When the server recieves a client

request, it will always query the `State Manager` . We use locking mechanisms to ensure that multiple servers do not try to update/read  the state at the same time. All communications (shown in the arrows) happen via `gRPC` calls only. The communication between the `Client` and the `Server` (cluster) happens via secure channels only (with certificates and signatures). Ideally we would be using a database or message queues  for managing the state, but due to lack of time we just stick to `Go` programs as each component of our system. Every time a client wants to make a request to the server, it will first ask the `Load Balancer` (client side load balancing). We now discuss how the system works internally.

# Clients

## Drivers

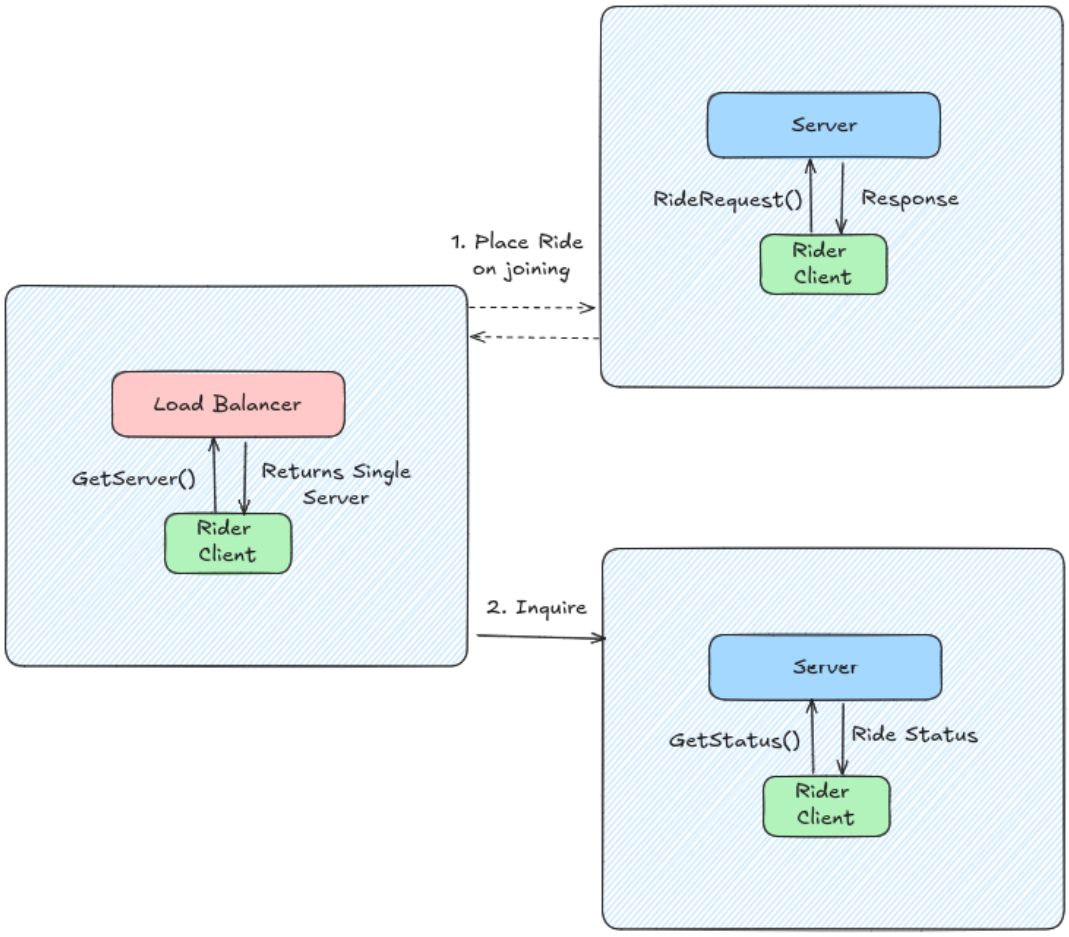



The communication with the Driver happens differently from a rider client. When a driver first connects, he would have to request for all the servers from the load balancer  (Step 1) and then connect to each of the servers. This connection is done via the server side streaming gRPC mechanism, this stream is not closed at the server side and is kept open forever (Step 2).

Finally, when we request for a new ride from the rider side (Step 3), the stream created above is used to communicate to the driver, to which driver can respond via rejection/accepting. The load balancer is not shown here, but the response of

the driver is done via a load balancer, that is, the server who sent the request to the driver  may not be the same server who got the response back. But they (server) are all kept in sync with each other via the central state.

## Riders



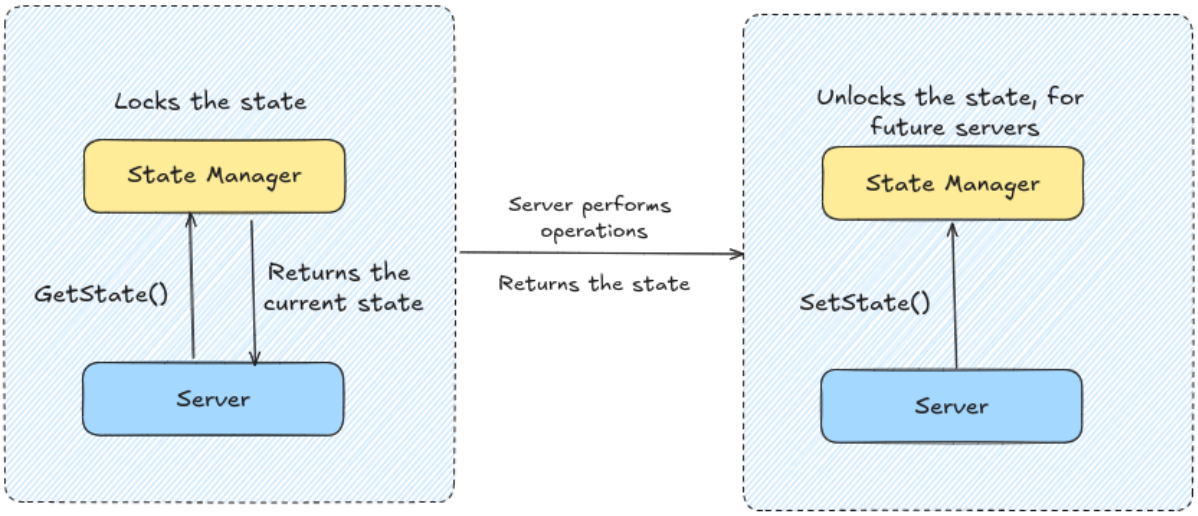

The servers here may be different, before each client request, the client will fetch the server from the load balancer and then perform operations (GetStatus/RideRequest)

The rider side is simpler. On starting the rider client, it will first place its ride and get the response straight away. From then on it will be able to inquire its ride status everytime. Preeceding all requests, the load balancer is contacted to get the server address.

> Note: The server also acts as a client, initially it registers itself to the load balancer, this facilitates service discovery, where new servers can be registered and sent to the clients later on. This is done via a AddService() call whenever a server boots. Further more, as we depict now, the server also acts like a client to get the state from the server.

## Consistency

To maintain consistency across all servers, we have a central state cordinating the information of all the rides among different servers.




We have set up locking mechanisms to ensure no two servers have the state at the same time.

```
func GetState(ctx Context, req StateRequest){
    s.mu.Lock() // lock the state of the server (s)
    ...
    s.holder = req.Server // set the holder of the state to t
he requesting server
```

```
    return State;
}
```

```
func SetState(ctx Context, req State){
    s.state = make([]rideInfo, 0, len(req.State))
    for _, rideInfo := range req.State { // process all the r
ides updated
        ride := rideInfo{
        ..
            // process the updated state
        ..
        }
        s.state = append(s.state, ride)
    }
    s.holder = "" // no holder anymore
    s.mu.Unlock() // unlock the state
    reutrn True // return a sucess response
}
```

The locking mechansims ensure consistency across all the states. We have also set logging mechansims on the server, which can be found on their terminal screen. Futher, we have three load balancing mechansims. Pick First, the default mechanism, picks any server at random. Round robin keeps track of all the servers requested and returns a server to the client based on the indexing the server list. We also implemented a weighted round robin, where we sample a server based on its weight. The motivation behind this being, in real scenarios servers are sent to the client via load balancer based on different factors such as its performance, response time, health etc and thus higher weighted servers would have a higher probability of being sampled via the load balancer.