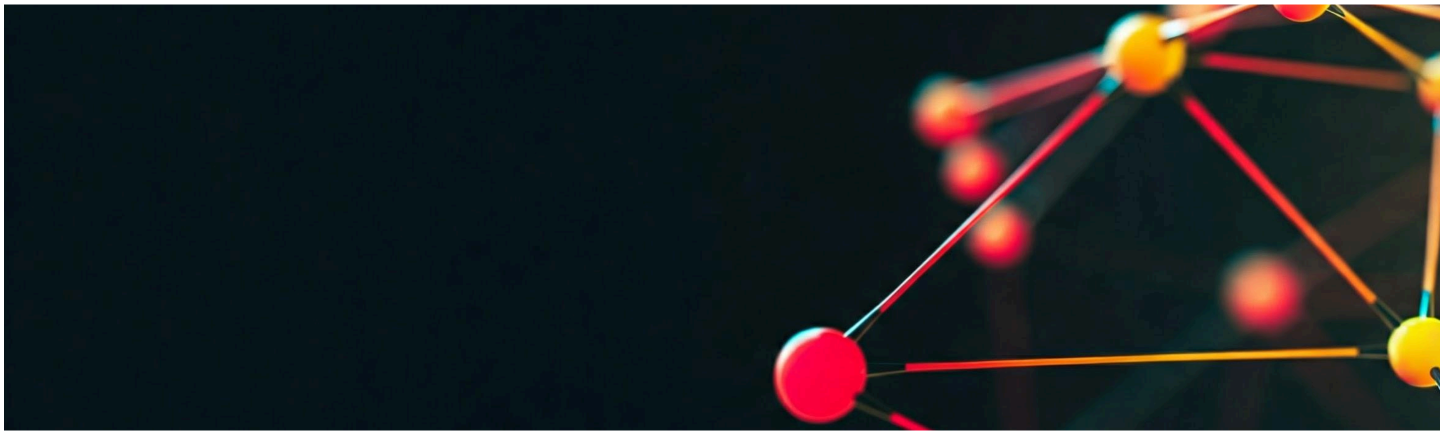


236351 - Distributed Systems

Paxos' Replicated Scooter Service

Barakat Gadban, 200384436
Itay Flam, 316497064

Table of Contents

Introduction

For the project assignment, we implemented a scooter rental service based in Paxos (Greece). Customers can reserve scooters and update the distance they travelled upon returnal.
We implemented a minimalistic web interface that supports adding, reserving and returning scooters.

The front-end was implemented using Angular, which communicates with the Go back-end service via a RESTful API. We used Docker to instantiate the replication servers. The replication server uses gRPC for inter-server communication, and etcd for failure detection and leader election.

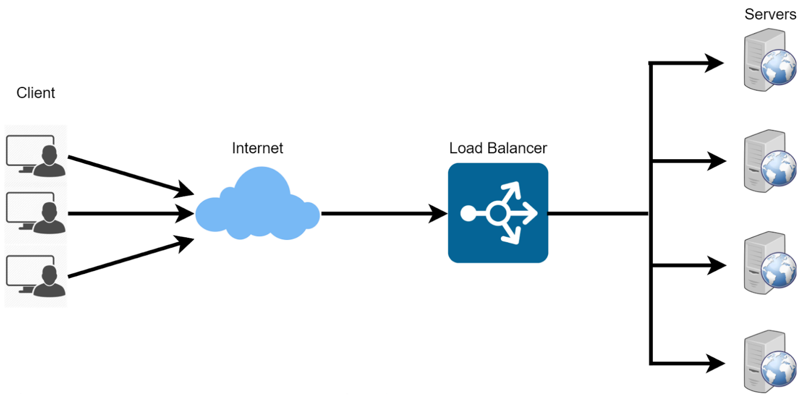
All servers maintain a replicated log of all actions carried out by clients (i.e. scooter additions, reservations and returnals) from which the shared state can be inferred. We implemented the multi-paxos protocol for the consensus layer, such that each instance of a Paxos invocation corresponds to a client’s action. Our service supports server crash failures (up to a minority of servers) and recovery. Furthermore, we periodically create snapshots of the log, thus conserving memory usage.

Implementation

We followed the project's guidelines and split the project to the following directories:

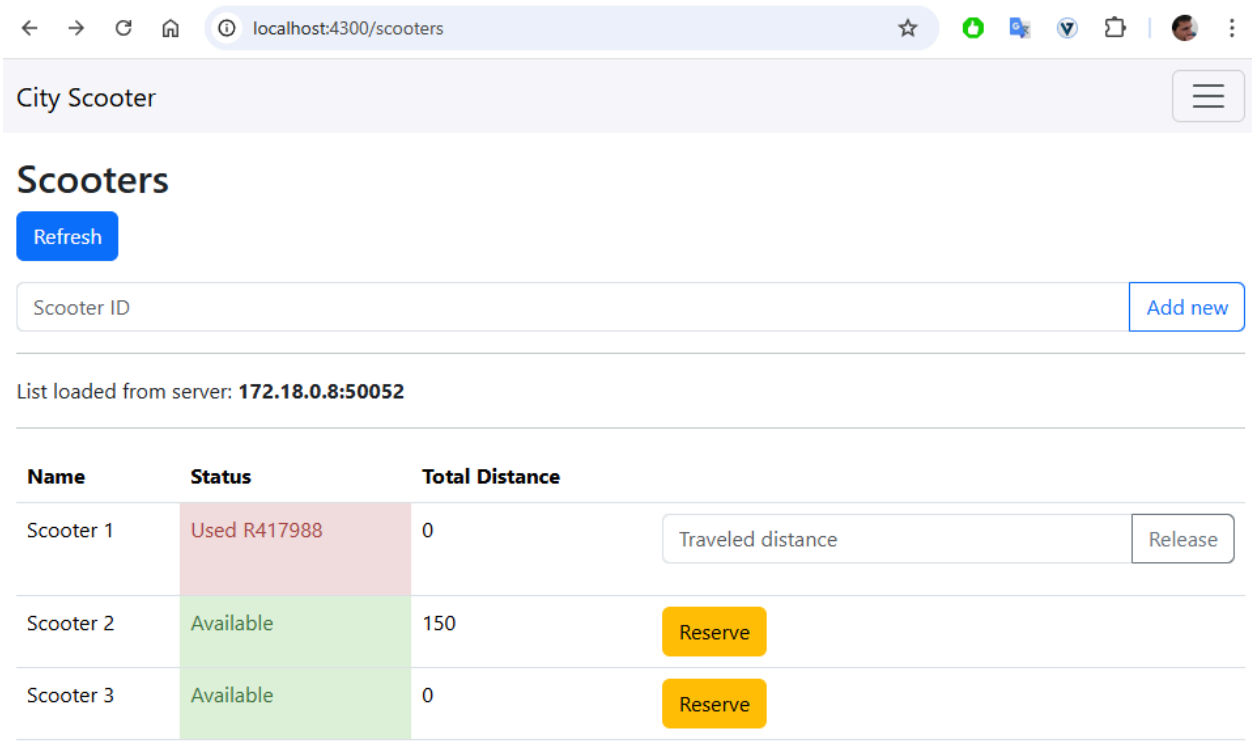
```
proj.zip
├── report.pdf ........................................   Our report
├── src/ ..................................................   Our source code
│   ├── docker/ .......................................   Docker related files
│   ├── server/ .......................................   Our server implementation in Go
│   ├── etc/ ...............................................   Our Angular front-end service
│   └── README.md ...............................   Comments for running the project
├── students.json
└── repo.url
```

Client Side



This is the actual GUI application that end user will access to interact with our system.
Our client implementation resides in the etc/ directory. We built the front end as a web single page application (SPA) using Angular, the application has 2 main pages:

- /scooters - the main demo client app, has a table of scooters that is loaded from the backend, an option to create a new scooter, and buttons to reserve and release scooters.


localhost:4300/scooters

City Scooter

Scooters

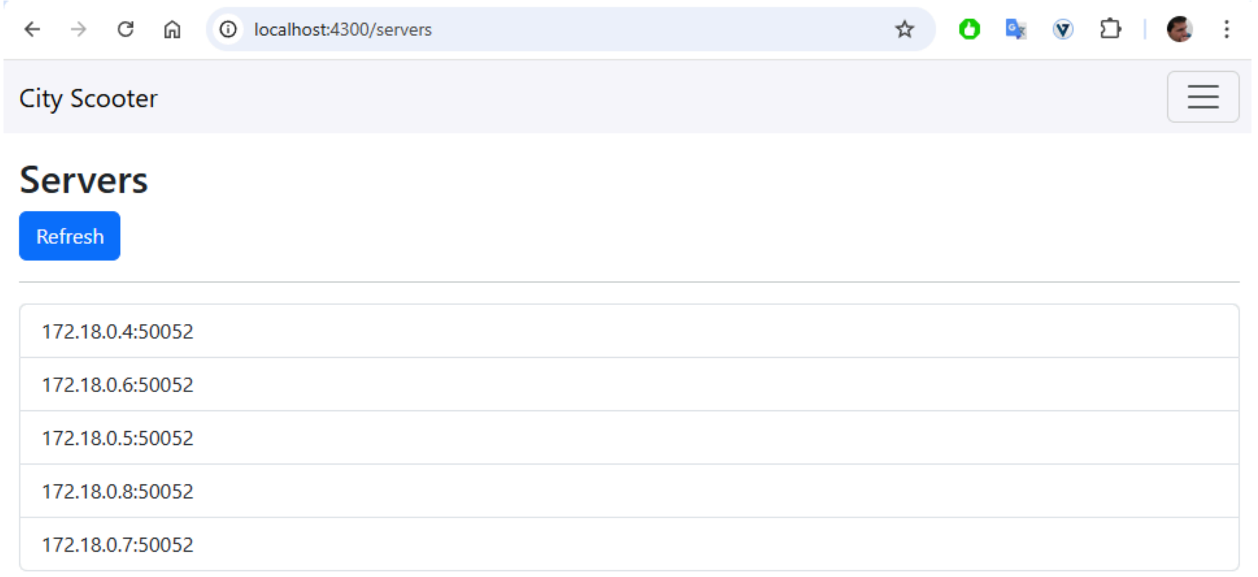
Refresh

Scooter ID | Add new

List loaded from server: **172.18.0.8:50052**

Name	Status	Total Distance	
Scooter 1	Used R417988	0	Traveled distance / Release
Scooter 2	Available	150	Reserve
Scooter 3	Available	0	Reserve

- /servers - displays the list of backend servers that are currently part of the replication cluster.


localhost:4300/servers

City Scooter

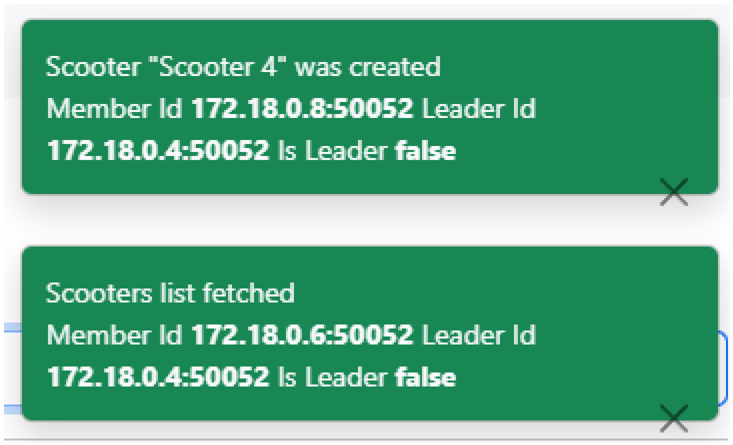
Servers

Refresh

```
172.18.0.4:50052
172.18.0.6:50052
172.18.0.5:50052
172.18.0.8:50052
172.18.0.7:50052
```

Data from both pages is loaded using HTTP requests to the backend load balancer, which forwards them to the scooter service that is part of each of the backend servers.

An important part of the client application that maybe worths mentioning is the **scooter.service.ts** (src/etc/spa/src/app/services/scooters.service.ts), this an angular service that is injected using dependency injection to relevant components in the app, each public function in it corresponds to a REST method in the scooters API, takes arguments from the caller as variables, builds the url, sends and HTTP request with the needed HTTP verb (GET/PUT/POST...), waits for the response that is JSON formatted, parses it to a JS object and returns. The service also displays success and error messages that include the ID of the responding server and the ID of the leader.



Scooter "Scooter 4" was created
Member Id **172.18.0.8:50052** Leader Id **172.18.0.4:50052** Is Leader **false**

Scooters list fetched
Member Id **172.18.0.6:50052** Leader Id **172.18.0.4:50052** Is Leader **false**

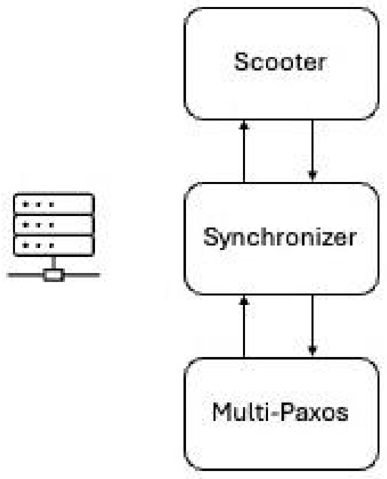
The client app has its own multi-stage Dockerfile, that:

1. Uses node:22.13.1-alpine as base image to build the app using **ng** cli. This produces static HTML, JS and css files
2. Then uses nginx:alpine as base image to create an image of nginx web server that hosts our single page application files

The final client app image is **scooter-spa:0.1**

Server Side

We used Go to implement servers. Logically, we split our implementation to 3 objects: Scooter, Synchronizer and Multi-Paxos.
The Scooter service is responsible to receive client HTTP requests and implement the RESTful API, that is map the HTTP request to the server's corresponding function. The Scooter service triggers the Synchronizer's logic.



The Synchronizer class is responsible for synchronizing the communication with the client and the multi-paxos protocol. The synchronizer maintains a logical log of the system, and the current shared state. When the client invokes a request, the synchronizer is responsible for adding the action to the replicated log via multi-paxos. The synchronizer class helps maintain order between client requests and the distributed protocol.
Finally, the **Multi-Paxos** service is responsible for the inter-server consensus communication via gRPC. Servers communicate via sending PREPARE , ACCEPT and COMMIT messages; we implemented the Paxos algorithm as seen in class. We use an array of instances to support multiple instances of Paxos; each instance has a slot. This array is in essence the replicated log.

Upon start up, each server instantiates each service and opens a connection to the etcd service. For failure detection, each server maintains a lease with the etcd service. Each server periodically renews the lease with etcd, so if a server crashes it fails to renew its lease and thus is suspected. Etcd offers a leader election service that we leverage in our implementation. For the sake of the project we assume that etcd implements an Ω-failure detector, and thus is able to elect a single leader from some point ("after GST").

We use nginx/Kubernetes for load balancing. When a client sends a request, it first goes through the load balancing service that picks a server to carry out the request.
A server that receives a request tries to add it to the log. It does so by invoking multi-paxos and trying to commit it for the latest instance of Paxos. If the receiving server is a leader, it will propose it for the latest un-committed Paxos instance, otherwise, it will channel the request to its known leader.

The server side directory tree is:

```
proj.zip
├── src/
│   ├── server/
│   │   ├── Dockerfile
│   │   ├── events_queue.go
│   │   ├── github.com/  .................. (Folder)
│   │   ├── go.mod
│   │   ├── go.sum
│   │   ├── leader_election.go ............. Manages all interaction with etcd
│   │   ├── multipaxos_client.go .......... Responsible for outgoing gRPC requests
│   │   ├── multipaxos_service.go ....... Handlers for incoming gRPC calls
│   │   ├── multipaxos.proto ................ Proto definitions of Paxos gRPC service
│   │   ├── scooter.go ......................... Definition of the scooter struct that is used
│   │   │                                        in state and snaphots
│   │   ├── scooter_service.go ............ Scooter HTTP server
│   │   ├── server.go ........................... Main app
└─-─  ├──   synchronizer.go ............. Connects the HTTP calls and Paxos logic
```

Servers are initialized in the server.go file (and that is where the main function resides). The bulk of multi-paxos is implemented in the multipaxos_service.go file, the synchronizer is implemented in the synchronizer.go file, and the scooter service in the scooter_service.go file.
For gRPC communication we implemented the multipaxos.proto folder which specifies the communication API between servers. We created a Dockerfile for server setup.

Docker

We used Docker containers to host the different applications that together form our distributed system:

- Backend servers - multiple replicas of the image **scooter-server:0.3**
- Client web server - single instance of the image **scooter-spa:0.1**. In production deployment, multiple replicas can be deployed, this will require a load balancer for the client app that we didnt include in our demo
- etcd server, one instance of the image **bitnami/etcd:latest**. We used etcd for servers registration (membership) and leader election. In a production deployment multiple instances should be used and this will require some minimal configuration.
- HTTP Load balancer for the backend's scooter service, we provided 2 options: **traefik:v2.3** and **nginx:latest.** We first used nginx, but figured out that the free version does not have health checks, it works only with a preset list of upstream servers, and when a server goes down it continues to send traffic to it.
  For this reason we changed the LB to use traefik, that fixes this problem by identifying upstream servers based on container

label, uses periodic health checks (using http request to /health) and sends traffic only to healthy servers.

To deploy these servers we used docker-compose (src/docker/docker-compose.yml).

OPTIONAL: We also provided Kubernetes deployment files to host the solution on K8s, and tested that on a local minikube server. The big advantage of k8s over docker-compose in our case is using K8s service as HTTP load balancer that provides built-in upstream servers identification using labels and immediate health monitoring.

The Docker folder tree is:

```
proj.zip
├── src/ .......................................... Your source code
│   ├── docker/ ................................ Docker related files
│   │   ├── .env ................................ Common env vars for backend servers
│   │   ├── docker-compose.yml ..... Docker compose file
│   │   ├── nginx.conf ..................... Nginx configuration
│   │   └── k8s ................................ Folder of Kubernetes yaml manifests
```

The .env file contains environment constants (e.g. lease duration), the docker-compose.yml contains the contents of the compose file, and nginx.conf contains configurations for the nginx load balancer (if used), traefik configuration is included in the docker-compose.yml file.

Testing

We evaluated the correctness of the system using log outputs and the web interface.
Since the system is distributed, we used log output for each server to make sure the expected behaviour is achieved.
We initialized all relevant services with Docker compose.
The web interface has a page that shows all servers. We used it to see who are the current members in the system, and to check that failure detection works; we stop a container of a replication server, and then check that the server is actually taken off from the servers' list.
We manage the scooters in the scooters page. There we can add a scooter, reserve and release it. Using the refresh button we get a pop-up notification specifying the server who handled the request (which changes according to the load balancer); we use it to check that all server states are equal, so this is our way of validating the replication works.

To check server crashes and recoveries, we stop a container and check that it is omitted in the servers page. Then we change the shared state of the system and check that all servers are aligned to the new state. We then restart the server container, and check using the refresh button that that server responded with its state, and that it matches the state of the other servers.

We note that due to the lease duration and load balancing service, it may take a few seconds before changes are registered (when membership is not stable).

Challenges

We encountered three main challenges during our development:

1. **Verification and Debugging:** Since the system is distributed, we could not use traditional debugging tools (such as a debugger). We were not aware of any debugging tools for distributed systems, so we resorted to log printing. This made debugging difficult, and we had to infer the bugs from the log. We mainly tested our implementation on 5 replication servers, but the complexity grows very fast with each added replica.
2. **Crash Recovery and Snapshots:** Making sure that crash recovery worked properly with the periodical snapshots was a challenge. Coordinating the recovery, and making sure that a crashed server updates its snapshot and then all the log entries was hard to debug and test. We had to make sure that the recovered server reached the same shared state, which required a lot of testing via the web interface and log prints.
3. **Unfamiliar Development Language:** For the both of us, it was the first time programming in Go. Go had some interesting features since it is a compiled language, and therefore also had a bit harder learning curve than interpreted languages such as Python.

Conclusion

The assignment sharpened our understanding of distributed consensus protocols. It was the first time the both of us encountered implementing a distributed protocol.
We managed to leverage technologies we saw in class to build a functioning distributed application.

We had to deal and overcome the inherent complexities that arise in a distributed system, which included hours of tedious debugging sessions. We felt the gap between the theoretical algorithms presented in class and their actual implementation. We are **especially proud of our implementation of crash recoveries with snapshots.** Crash recoveries require complex orchestration, and implementing it with the added complexity of snapshot required a lot of work. Furthermore, crash recovery is usually not a topic discussed in the theoretical literature, but is an important real world scenario.

Overall, given the time frame we had for the project, we are very proud of the result we achieved. We had a good time working together, and had a great time working on the project which we found challenging and interesting.