Distributed Algorithms - CW2 - Multi Paxos

Rishabh Jain (rj2315) & Vinamra Agrawal (va1215)

February 21, 2018

1 Introduction

The aim of this coursework was to implement and evaluate a simple replicated banking service that uses the version of Multi-Paxos described in the paper: Paxos Made Moderately Complex by Robbert van Renesse and Deniz Altınbüken. ACM Computing Surveys. Vol. 47. No. 3. February 2015.

2 Part 1 - System Structure

The `Paxos` module in the provided elixir modules initiates the system by spawning the clients, servers and the monitor. Then these components, spawn their own components necessary for the system to work properly (as described in the paper). The spawn diagram (1) here also shows the components each `Server` spawns. Note here that the `Scout` and `Commander` modules are spawned multiple times by the leader depending on the state of the Paxos algorithm.

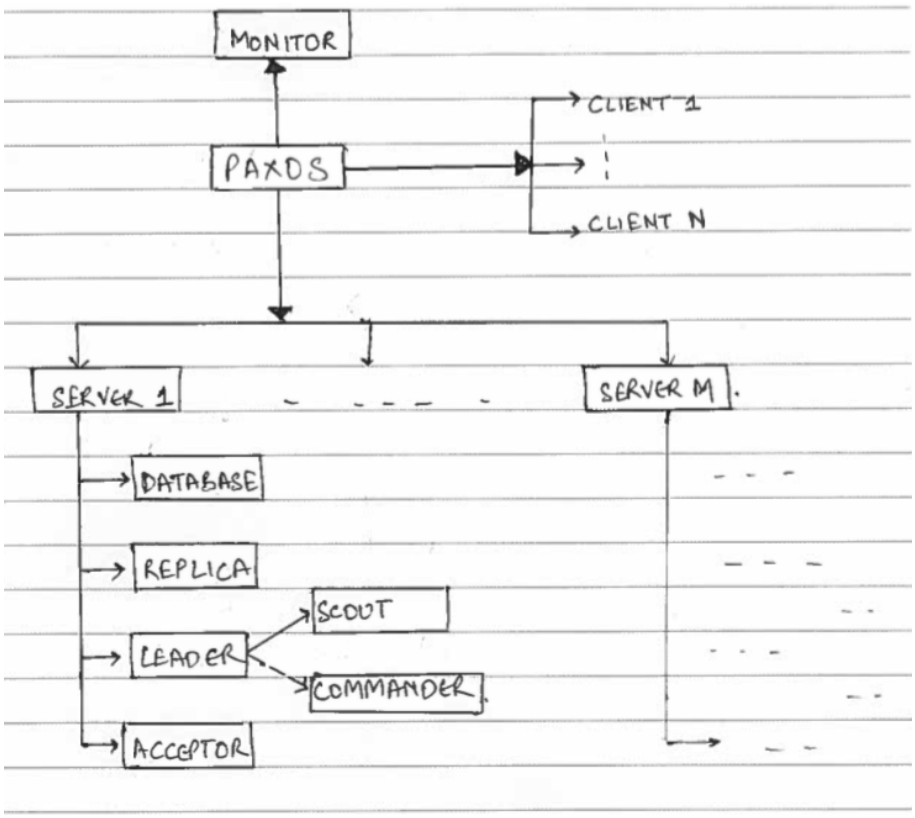

Figure 1: Spawn Diagram

We referred to the Paxos paper to understand the flow of messages in the system between the various modules and how it would work in our specific case. Thus, in the Communication flow diagram (2), we sketched the time line of the algorithm and how each module sends messages to the rest of the modules.

All the clients send multiple `requests` to the replicas and for each of them, they get a `reply` back from it. While that happens, the `replica` also makes sure to `execute` the client requests on the `database` which then informs the `monitor` about the execution.

The rest of the system behaves exactly how it is described in the paper. The Phase 1 messages are labeled as `p1a` and `p1b` (similarly for Phase 2). Note here that the figure shows an ideal case where the `Scout` sends an `:adopted` message to the `Leader`. It can send a `:preempted` message instead. In that case the `Leader` might spawn another `Scout` and move on with the algorithm again.

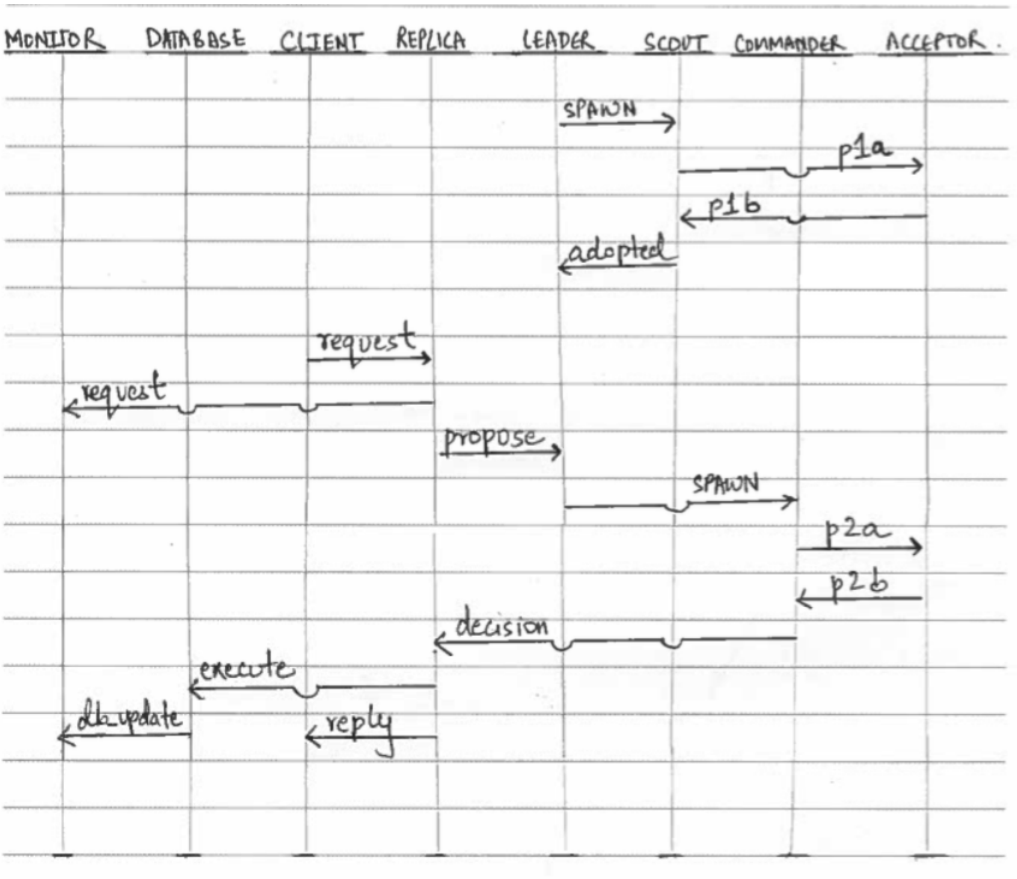

Figure 2: Communication Flow Time-line

3 Part 2 - Implementation and evaluation

Following the explanation given in the paper and referring to the pseudo code provided we were able to write working implementations for `Acceptor`, `Scout`, `Commander`, `Leader` and `Replica`.

After implementation of the system, to get better understanding we implemented we tried some tests better understand the system.

We have also added a `README` file in the folder, describing the different config variables added and how to use them.

3.1 Setup

3.1.1 Local Setup

- Macbook Pro 2016 (macOS High Sierra).
- 2.6 GHz Intel Core i7.
- 16GB RAM.

3.1.2 Docker Setup

- Docker run on same machine as mentioned above.
- Version - 3.4
- Image - elixir:alpine

3.2 Tests and Deductions

3.2.1 Test 1

Clients = 1; Severs = 1; Max Broadcasts per Client = 500; Client message send rate = 1/5ms; Window size = 100

Running our system on the above trivial case gives us perfect results as expected. We were able to execute all the commands in 3 seconds. This is expected as having just one server doesn't require any agreement on the command to execute thus can process commands quickly. The results were similar in both docker and locally.

```
time = 3000  updates done = [{1, 499}]
time = 3000 requests seen = [{1, 499}]

Client 1 going to sleep, sent = 500
time = 4000  updates done = [{1, 500}]
time = 4000 requests seen = [{1, 500}]
```

3.2.2 Test 2

Clients = 5; Severs = 1; Max Broadcasts per Client = 500; Client message send rate = 1msg/5ms; Window size = 100

In this case we are increasing the number of clients, therefore one server now expects to serve many more requests. However still Systems can decide what commands to process quickly as just one server exists. The total time to process all commands was 4 seconds and the results were similar both on docker and locally.

```
time = 3000  updates done = [{1, 2489}]
time = 3000 requests seen = [{1, 2489}]

Client 1 going to sleep, sent = 500
Client 2 going to sleep, sent = 500
Client 3 going to sleep, sent = 500
Client 4 going to sleep, sent = 500
Client 5 going to sleep, sent = 500
time = 4000  updates done = [{1, 2500}]
time = 4000 requests seen = [{1, 2500}]
```

3.2.3 Test 3

Clients = 1; Severs = 5; Max Broadcasts per Client = 500; Client message send rate = 1msg/5ms; Window size = 100

In this case we are increasing the number of servers, therefore each server now processes much less requests as requests get distributed among the servers. However the time it took was really long as for each command they need to first agree to which command to execute. The total time taken was around 16 seconds when run locally. When run on docker, the livelocks occurred more frequently thus the time taken was usually around 2 to 3 minutes.

```
time = 15000  updates done = [{1, 498}, {2, 498}, {3, 498}, {4, 498}, {5, 498}]
time = 15000 requests seen = [{1, 100}, {2, 100}, {3, 100}, {4, 100}, {5, 100}]
Scouts spawned: 3075
Commanders spawned: 371483

time = 16000  updates done = [{1, 500}, {2, 500}, {3, 500}, {4, 500}, {5, 500}]
time = 16000 requests seen = [{1, 100}, {2, 100}, {3, 100}, {4, 100}, {5, 100}]
Scouts spawned: 3101
Commanders spawned: 383070
```

3.2.4 Test 4

Clients = 2; Severs = 3; Max Broadcasts per Client = 500; Client message send rate = 1msg/20ms; Client timeout = 10s; Window size = 100

In this case we increased the time between the each message client sends, however with the current timeout we observe that clients were unable to send all the requests. The clients were able to make around 400 requests when run locally. On docker the system behaved similarly.

```
Client 1 going to sleep, sent = 410
Client 2 going to sleep, sent = 420
time = 10000  updates done = [{1, 604}, {2, 604}, {3, 604}]
time = 10000 requests seen = [{1, 276}, {2, 277}, {3, 277}]
Scouts spawned: 1295
Commanders spawned: 270044
```

```
paxos | Client 1 going to sleep, sent = 466
paxos | time = 10000  updates done = [{1, 611}, {2, 611}, {3, 611}]
paxos | time = 10000 requests seen = [{1, 309}, {2, 311}, {3, 310}]
paxos | Scouts spawned: 1070
paxos | Commanders spawned: 158648
paxos |
paxos | Client 2 going to sleep, sent = 466
paxos | time = 11000  updates done = [{1, 612}, {2, 612}, {3, 612}]
paxos | time = 11000 requests seen = [{1, 310}, {2, 312}, {3, 310}]
paxos | Scouts spawned: 1100
paxos | Commanders spawned: 176210
```

3.3 Live-locks

We also observed in the above tests that there was often a live-lock situation, where it took a long time to take make a consensus on what command should be processed. This happens when we have a promise and preemptive cycle between two leaders.

We can understand this case by assuming two leaders A and B. As leader A get its 'Phase 1' of messages promised by acceptors. Leader B comes with higher proposal number forcing acceptors to preempt on the first leader. However soon enough as Leader B get the required number of promises to start 'Phase 2', Leader A comes back with an updated and higher number which in turn forces acceptors to preempt Leader B. This cycle continues to cause a live-lock and no leader is able to complete the entire cycle and get a consensus.

3.3.1 Solution 1: Random Timeout

The first approach we took was to add a random delay before the proposer tries again. This in most cases, would allow other leaders to form a consensus during this delay. This helps to break the live-lock cycle and we were able to get the results much quickly.

Test 5

Clients = 2; Severs = 3; Max Broadcasts per Client = 500; Client message send rate = 1msg/5ms; Window size = 100

In this test we pass in similar parameters as Test 4 3.2.4. However we notice that this time we were able to process all the requests much quickly even with additional sleep because of the random-timeouts that we implemented. The results were similar locally and on docker.

```
Client 1 going to sleep, sent = 500
Client 2 going to sleep, sent = 500
time = 4000  updates done = [{1, 1000}, {2, 1000}, {3, 1000}]
time = 4000 requests seen = [{1, 332}, {2, 334}, {3, 334}]
Scouts spawned: 36
Commanders spawned: 4533
```

3.3.2 Solution 2: Leader Wait

Although our previous solution worked in most cases, it was still not a full proof solution and could result in live-locks. This solution is more intuitive and prevents live-locks in the above cases.

In this implementation, when we get the preemptive message and then we wait until the `leader` who is `active` or in Phase 2, to complete their cycle. Once we get this confirmation we start the preemptive leader's consensus cycle. This is done via the `waiting` and `leader_resp` messages passed between the leaders. We have also added comments in the `leader.ex` file for a better understanding of the algorithm.

Test 6

Clients = 2; Severs = 3; Max Broadcasts per Client = 500; Client message send rate = 1msg/5ms; Window size = 100

In this test we pass in similar parameters as Test 4 3.2.4. However we notice that this time we were able to process all the commands much quickly. We also observe that lesser amount of Scouts are spawned as one leader waits for other leader to complete its cycle. The results were similar locally and on docker.

```
Client 2 going to sleep, sent = 500
Client 1 going to sleep, sent = 500
time = 4000  updates done = [{1, 1000}, {2, 1000}, {3, 1000}]
time = 4000 requests seen = [{1, 332}, {2, 334}, {3, 334}]
Scouts spawned: 3
Commanders spawned: 3000
```

3.4 Window Size

Clients = 2; Severs = 3; Max Broadcasts per Client = 500; Client message send rate = 1msg/5ms; Client timeout = 10s; Window size = 10

In this case we change the window size of the System to observe its impact. However we observe no considerable change in the output of the System. We get similar results both locally and on docker.

```
Client 1 going to sleep, sent = 500
Client 2 going to sleep, sent = 500
time = 3000  updates done = [{1, 999}, {2, 999}, {3, 999}]
time = 3000 requests seen = [{1, 332}, {2, 334}, {3, 333}]
Scouts spawned: 3
Commanders spawned: 2960

time = 4000  updates done = [{1, 1000}, {2, 1000}, {3, 1000}]
time = 4000 requests seen = [{1, 332}, {2, 334}, {3, 334}]
Scouts spawned: 3
Commanders spawned: 3000
```