Distributed Algorithms - Coursework 2 Report

Chris Hawkes Robert Holland

1 Execution environment

1.1 Single Node

We both run single node tests on the MacBook Pro with 8GB of RAM. It supports 2 Intel i5 cores at 2.7Ghz.

1.2 Docker network

Same as above using Docker for Mac with Version 17.12.0-ce-mac49 (21995).

2 Paxos Structure Overview

Clients send requests to cyclically chosen Replicas (or servers). Requests are then transferred to the proposals queue if $slot\_in < slot\_out + window\_size$ at which point $slot\_in$ is incremented by 1. Following this, a proposal is made to all Leaders. Otherwise, Replicas will wait until a proposal (possibly not their own) has been decided on at which point they increment $slot\_out$ by 1, allowing them to make another proposal.

Active Leaders that receive the proposal will spawn a Commander, tasked with asking Acceptors to accept the proposal. If a majority accepts, a decision is made to perform the operation and so a message is sent to all Replicas to carry out the command for this slot. Each Active leader may spawn many Commanders, one for each proposed slot.

If another Leader's Scout successfully 'wins' one of the Acceptors (by offering them a bigger ballot number) that responds to the aforementioned Commander, that Commander is forced to pre-empt its Leader and the decision isn't made. These Scouts are spawned by inactive Leaders who are attempting to win a majority of Acceptors with a higher ballot number. The previously active (and now pre-empted) Leader becomes inactive, and will spawn Scouts of its own in attempt to regain control.

Scouts are parameterised by the current ballot number of the Leader. If a majority of Accepters adopt the ballot number, the Scout relays that it has been adopted and the Leader assumes active status. Alternatively, a higher, already accepted ballot number could be reported by an Acceptor to the Scout who will then pre-empt the Leader. Consequently, the Leader will update its ballot number in order to beat the reported highest and try again.

The following two figures illustrate the system structure in the two modes of operation. The first is with an *inactive* Leader, who has invoked Scouts and the second an *active* Leader, who has invoked Commanders.

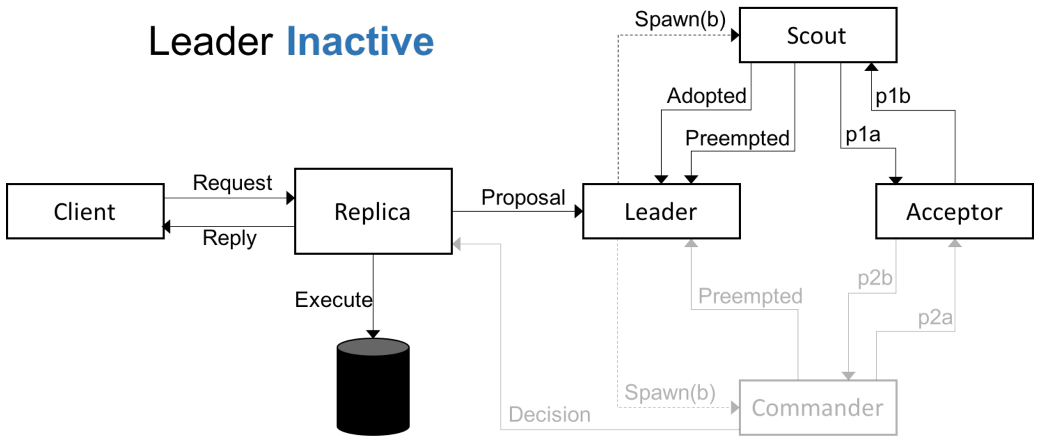

Figure 1: Paxos with an inactive (scouting) leader

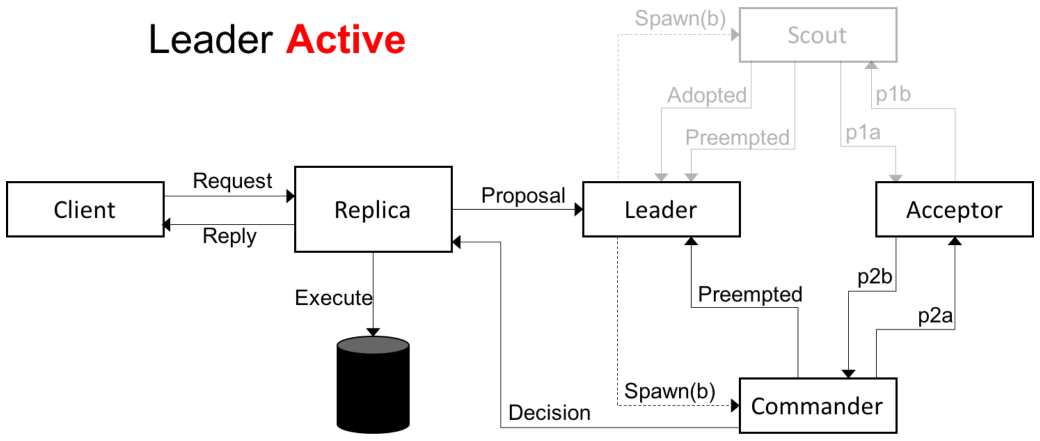

Figure 2: Paxos with an active (commanding) leader

3 Implementation and Evaluation

Through testing Paxos with different parameters, we confirmed numerous hypotheses about how they change performance. In response to these, we also implemented improvements to the system.
We also added two lines to the output that detail the total number of Scouts and Commanders each server has created.
Finally, we define a *base* test below with the following run configurations.

Table 1: Base test configurations

Servers	Clients	Window	Client Messages	Client Message Delay
3	2	1	100	5

When run, the typical output is as follows.

```
time = 1000  updates done = [{1, 59}, {2, 59}, {3, 59}]
time = 1000 requests seen = [{1, 66}, {2, 68}, {3, 66}]
time = 1000 scouts = [{1, 468}, {2, 424}, {3, 386}]
time = 1000 commanders = [{1, 14904}, {2, 15026}, {3, 15074}]
.
.
.
time = 33000  updates done = [{1, 199}, {2, 199}, {3, 199}]
time = 33000 requests seen = [{1, 66}, {2, 68}, {3, 66}]
time = 33000 scouts = [{1, 2288}, {2, 2049}, {3, 1902}]
time = 33000 commanders = [{1, 210229}, {2, 217688}, {3, 218746}]

time = 34000  updates done = [{1, 200}, {2, 200}, {3, 200}]
time = 34000 requests seen = [{1, 66}, {2, 68}, {3, 66}]
time = 34000 scouts = [{1, 2310}, {2, 2069}, {3, 1922}]
time = 34000 commanders = [{1, 213629}, {2, 221688}, {3, 222611}]
```

Figure 3: Base test output

Note that it takes a full 34 seconds to execute all 200 requests locally and 82 seconds on Docker. This is before any improvements were added to the system and the ping-pong argument described in the paper by Robbert van Renesse and Deniz Altınbükenwhere showed this system may not guarantee liveness. This is where no progress may be made as leaders may continue to indefinitely to usurp each other.

Docker run

```
paxos | time = 1000  updates done = [{1, 27}, {2, 27}, {3, 27}]
paxos | time = 1000 requests seen = [{1, 66}, {2, 68}, {3, 66}]
paxos | time = 1000 scouts = [{1, 133}, {2, 123}, {3, 117}]
paxos | time = 1000 commanders = [{1, 1335}, {2, 1635}, {3, 1622}]
paxos |
paxos | time = 2000  updates done = [{1, 32}, {2, 32}, {3, 32}]
paxos | time = 2000 requests seen = [{1, 66}, {2, 68}, {3, 66}]
paxos | time = 2000 scouts = [{1, 220}, {2, 197}, {3, 185}]
paxos | time = 2000 commanders = [{1, 2967}, {2, 3302}, {3, 3263}]
.
.
.
paxos | time = 82000  updates done = [{1, 200}, {2, 200}, {3, 200}]
paxos | time = 82000 requests seen = [{1, 66}, {2, 68}, {3, 66}]
paxos | time = 82000 scouts = [{1, 1371}, {2, 1166}, {3, 1063}]
paxos | time = 82000 commanders = [{1, 99501}, {2, 111751}, {3, 110591}]
paxos |
paxos | time = 83000  updates done = [{1, 200}, {2, 200}, {3, 200}]
paxos | time = 83000 requests seen = [{1, 66}, {2, 68}, {3, 66}]
paxos | time = 83000 scouts = [{1, 1371}, {2, 1166}, {3, 1063}]
paxos | time = 83000 commanders = [{1, 99501}, {2, 111751}, {3, 110591}]
```

Though the Docker run took longer to complete, each server created less scouts and commanders than the local run which may be a sign that even though message sending was slower there was less contention to gain majorities on the docker network. This observation is consistent with all Docker tests we performed.

3.1 Window Size

Window size limits the number of slots that a Replica can propose for at any given time. Limiting size to 1 reduces the number of proposals active Leaders can spawn Commanders to decide upon. This makes it more likely for them to be pre-empted before many decisions can be made, as they have to wait for their replica to be told that a decision was made before it can propose a new client request. Then the active Leader can again create a Commander to try and decide on this new proposal. This makes it more likely that Leaders spend time pre-empting each other, rather than making decisions.

In the worst case this process continues indefinitely and is known as live-lock. Consequently, the decision rate drastically increased.

Table 2: Window size test configurations

Servers	**Clients**	**Window**	**Client Messages**	**Client Message Delay**
3	2	100	100	5

Local run

```
time = 1000  updates done = [{1, 190}, {2, 190}, {3, 190}]
time = 1000 requests seen = [{1, 66}, {2, 68}, {3, 66}]
time = 1000 scouts = [{1, 571}, {2, 477}, {3, 426}]
time = 1000 commanders = [{1, 28229}, {2, 29050}, {3, 29198}]

time = 2000  updates done = [{1, 200}, {2, 200}, {3, 200}]
time = 2000 requests seen = [{1, 66}, {2, 68}, {3, 66}]
time = 2000 scouts = [{1, 591}, {2, 500}, {3, 444}]
time = 2000 commanders = [{1, 31827}, {2, 32448}, {3, 32796}]
```

Docker run

```
paxos | time = 1000  updates done = [{1, 54}, {2, 54}, {3, 58}]
paxos | time = 1000 requests seen = [{1, 66}, {2, 68}, {3, 66}]
paxos | time = 1000 scouts = [{1, 91}, {2, 83}, {3, 73}]
paxos | time = 1000 commanders = [{1, 2018}, {2, 2147}, {3, 2448}]
.
.
.
paxos | time = 6000  updates done = [{1, 200}, {2, 200}, {3, 200}]
paxos | time = 6000 requests seen = [{1, 66}, {2, 68}, {3, 66}]
paxos | time = 6000 scouts = [{1, 220}, {2, 185}, {3, 168}]
paxos | time = 6000 commanders = [{1, 13653}, {2, 16854}, {3, 17199}]
```

This only applies because the probability of a process crashing in this simulation is 0, so there need not be a guarantee of time to reconfiguration.

Since we did not implement a reconfiguration command, and assume servers would never be brought back up after a crash, we find that the window size should be as large as possible.

3.2 Random wait before scouting

To avoid the possible ping-pong effect described earlier it was suggested in the paper that waiting whilst a non-faulty leader was currently active would achieve liveness. We implemented a simpler method that did not involve pinging the active leader but still achieves better liveness behaviour. This was done by applying a random wait (between 1 and 50 milliseconds) to the Leader between being pre-empted and scouting. This allows active Leaders to have more time to collect responses from Acceptors so that more actions are decided upon.
The *base* test (figure 3) includes no random wait and all commands are executed after 34 seconds.
Now by simply adding the random wait we are able to reduce the time taken to under one second. Even with the window size still set at 1.

Table 3: Random wait configurations

Servers	**Clients**	**Window**	**Client Messages**	**Client Message Delay**
3	2	1	100	5

Local run

```
time = 1000  updates done = [{1, 200}, {2, 200}, {3, 200}]
time = 1000 requests seen = [{1, 66}, {2, 68}, {3, 66}]
time = 1000 scouts = [{1, 22}, {2, 16}, {3, 14}]
time = 1000 commanders = [{1, 1418}, {2, 1292}, {3, 1382}]
```

Docker run

```
paxos | time = 1000 updates done = [{1, 200}, {2, 200}, {3, 200}]
paxos | time = 1000 requests seen = [{1, 66}, {2, 68}, {3, 66}]
paxos | time = 1000 scouts = [{1, 15}, {2, 23}, {3, 15}]
paxos | time = 1000 commanders = [{1, 1547}, {2, 870}, {3, 1329}]
```

By the first second, the *base* experiment created ≈ 400 scouts per Leader, but randomly waiting Leaders only produced ≈ 20 in both local and Docker runs.

3.3 Client Request Delay

In this simulation clients send messages at regular intervals to servers in round-robin fashion. We found that reducing the client message delay from 5ms to 0ms made no discernible difference in the local and Docker runs.
We hypothesised that increasing the request delay will slow down the system *only* if it is a limiting factor. It becomes a limiting factor only if the window size is big enough

because active leaders will be unable to decide on as many slots as are available. This is demonstrated below in a test with a large window size and request delay.

Table 4: Client Request configurations

Servers	Clients	Window	Client Messages	Client Message Delay
3	2	100	100	100

Local run

```
.
.
.
Client 1 going to sleep, sent = 97
Client 2 going to sleep, sent = 97
time = 11000  updates done = [{1, 182}, {2, 182}, {3, 182}]
time = 11000 requests seen = [{1, 64}, {2, 66}, {3, 64}]
time = 11000 scouts = [{1, 2626}, {2, 2372}, {3, 2226}]
time = 11000 commanders = [{1, 163445}, {2, 163584}, {3, 168764}]

time = 12000  updates done = [{1, 194}, {2, 194}, {3, 194}]
time = 12000 requests seen = [{1, 64}, {2, 66}, {3, 64}]
time = 12000 scouts = [{1, 2667}, {2, 2431}, {3, 2283}]
time = 12000 commanders = [{1, 169916}, {2, 174328}, {3, 179701}]
```

Docker run

```
.
.
.
Client 1 going to sleep, sent = 96
Client 2 going to sleep, sent = 95
time = 14000  updates done = [{1, 171}, {2, 171}, {3, 171}]
time = 14000 requests seen = [{1, 63}, {2, 64}, {3, 64}]
time = 14000 scouts = [{1, 2072}, {2, 1831}, {3, 1723}]
time = 14000 commanders = [{1, 160229}, {2, 159794}, {3, 161701}]

time = 15000  updates done = [{1, 191}, {2, 191}, {3, 191}]
time = 15000 requests seen = [{1, 63}, {2, 64}, {3, 64}]
time = 15000 scouts = [{1, 2155}, {2, 1876}, {3, 1797}]
time = 15000 commanders = [{1, 174219}, {2, 167482}, {3, 175543}]
```

Note that even with the large window size of 100, it took 12 seconds to complete locally instead of 2 seconds (as found in the window size test). The effect was not so pronounced on docker as messages take longer to send anyway.

3.4 Number of Clients

When testing how the number of clients affected performance, we kept the total number of commands to be executed at 200 by using 10 clients, each sending 20 messages. In this way we can compare well to the *base* test.

Table 5: Constant total messages, increased number of clients configuration

Servers	Clients	Window	Client Messages	Client Message Delay
3	10	1	20	5

```
time = 23000  updates done = [{1, 199}, {2, 199}, {3, 199}]
time = 23000 requests seen = [{1, 60}, {2, 70}, {3, 70}]
time = 23000 scouts = [{1, 2076}, {2, 1838}, {3, 1681}]
time = 23000 commanders = [{1, 164166}, {2, 167609}, {3, 167000}]

time = 24000  updates done = [{1, 200}, {2, 200}, {3, 200}]
time = 24000 requests seen = [{1, 60}, {2, 70}, {3, 70}]
time = 24000 scouts = [{1, 2082}, {2, 1843}, {3, 1687}]
time = 24000 commanders = [{1, 165566}, {2, 168380}, {3, 168400}]
```

We hypothesised that this would be equivalent to reducing the delay between sending messages of each client (see Client Request Delay). Hence, we should observe no difference between increasing the number of clients and the *base* test.
However, throughout the whole investigation we have observed very high variance of the time taken to completion. The result above does take less time to complete than the *base* test, but we did not find this to be the case in all runs.

3.5 Number of Servers

As previously discussed, the more likely you are to be pre-empted the less decisions you are able to make while active. Therefore, if we increase the number of servers from three to 20 the chance that another Leader's Scout causes an Acceptor to accept a higher ballot number, and thus pre-empt the active Leader's Commander, increases.

Table 6: Increased number of servers configuration

Servers	Clients	Window	Client Messages	Client Message Delay
20	2	1	100	5

```
time = 88000  updates done = [{1, 47}, {2, 47}, . . . , {20, 47}]
time = 88000 requests seen = [{1, 10}, {2, 10}, . . . , {20, 10}]
time = 88000 scouts = [{1, 6133}, {2, 5596}, . . . , {20, 3456}]
time = 88000 commanders = [{1, 28576}, {2, 19544}, . . . , {20, 47989}]
.
.
.
time = 100000  updates done = [{1, 47}, {2, 47}, . . . , {20, 47}]
time = 100000 requests seen = [{1, 10}, {2, 10}, . . . , {20, 10}]
time = 100000 scouts = [{1, 6605}, {2, 6076}, . . . , {20, 3658}]
time = 100000 commanders = [{1, 32560}, {2, 23192}, . . . , {20, 53413}]
```

By the 88$^{\text{th}}$ second locally only 47 requests have been proposed and executed no more by the 100$^{\text{th}}$ second. This is an example of live-lock.
We also found that the proportion of Scouts to Commanders was much higher in this test than in the *base* test. For example locally, server 1 had produced 6605 Scouts and 23560 Commanders by second 100 which is approximately one Scout to every four Commanders. In comparison, in the *base* test by second 34 server 1 had 2310 Scouts and 213629 Commanders, or approximately one Scout for every nine Commanders. This is because the chance a given Leader is active is reduced when there are more of them.

Docker run

```
.
.
.
paxos | time = 50000  updates done = [{1, 1}, {2, 1}, {3, 1}, ...]
paxos | time = 50000 requests seen = [{1, 10}, {2, 10}, {3, 10}, ...]
paxos | time = 50000 scouts = [{1, 3900}, {2, 3743}, {3, 3065}, ...]
paxos | time = 50000 commanders = [{1, 2}, {2, 3}, {4, 3}, ...]
```

Clearly with this large number of servers in this configuration with no random waiting we observe a similar live-lock effect where after 50 seconds only 1 update had been done across all of the replicas. Furthermore we can see that all of the servers are finding difficulty in attaining a majority as they all have a very low number of commanders. Instead they all spend most of their time being pre-empted and spawning more Scouts.

4 Coursework Feedback

We enjoyed this coursework, in part because there were no mandatory tests. In coursework 1 we spent much of our time running a large number of mandatory tests that we did not feel helped our exploration.
This time, we had more time to run our own tests and conduct our own investigation which led us to reach more interesting conclusions.
We also enjoyed the fact that alternative algorithms for consensus were suggested (such as raft) that we could look at.