Imperial College London

Distributed Algorithms - 347

Imperial College London

Department of Computing

CW Report

Authors:
Panayiotis Panayiotou - pp3414
Adrian Catană - ac7815

February 23, 2018

General Comments

Most of the comments about the design choices is included within the Elixir code. We also left some code commented out and we indicate within the report when to uncomment it to observe some behaviours (e.g. broadcasting instead of round-robin for clients). We also include a README.md which explains how to run and test the code and what is included. The tests for the system for single node and Docker network were run on the following lab machine configuration:

```
product: Intel(R) Core(TM) i7-6700 CPU @ 3.60 GHz
RAM    : 16 GB
cores  : 8
```

Note that we are also printing the number of commanders and scouts for each server. A nice point is that they helped us to debug a possible deadlock - our code had a loop that was only running for one step (this issue has to do with the way we implemented while loops: special recursive functions checked with if statements, that would just return the mutated state of replicas). We figured this out because our program was advancing in neither the size of the updates, requests, commanders or scouts. That is, our system was not trying to manage the requests anymore.

Secondly, the commanders and scouts are helpful to see when the system livelocks. If there are examples when the updates that we did and the requests that we saw are not increasing, but the commanders and scouts are growing up every seconds, it means that our system is currently dealing with the "Ping-Pong" behaviour.

Basic Implementation

Below we present a simple implementation of multi-paxos, as presented in the paper. The system logic consists of 5 main components: acceptor, scout, commander, leader and replica, together with the servers and the clients. During testing, we will provide details on how the system behaves for different configurations. We vary the number of servers, clients, the window-size, client request sending rate on both local and Docker.

An important design choice that was made was choosing how to implement the while and forever loops presented in the paper, together with maintaining the overall state in replicas. Our idea for the first issue was to make recursive calls for the same function until a certain condition (the loop condition) was met. Then, we solved the "mutability" of the state by always returning tuples of everything that the loops modified.

Figure 1: The diagram shows the messages exchanged between the different components of the Paxos implementation

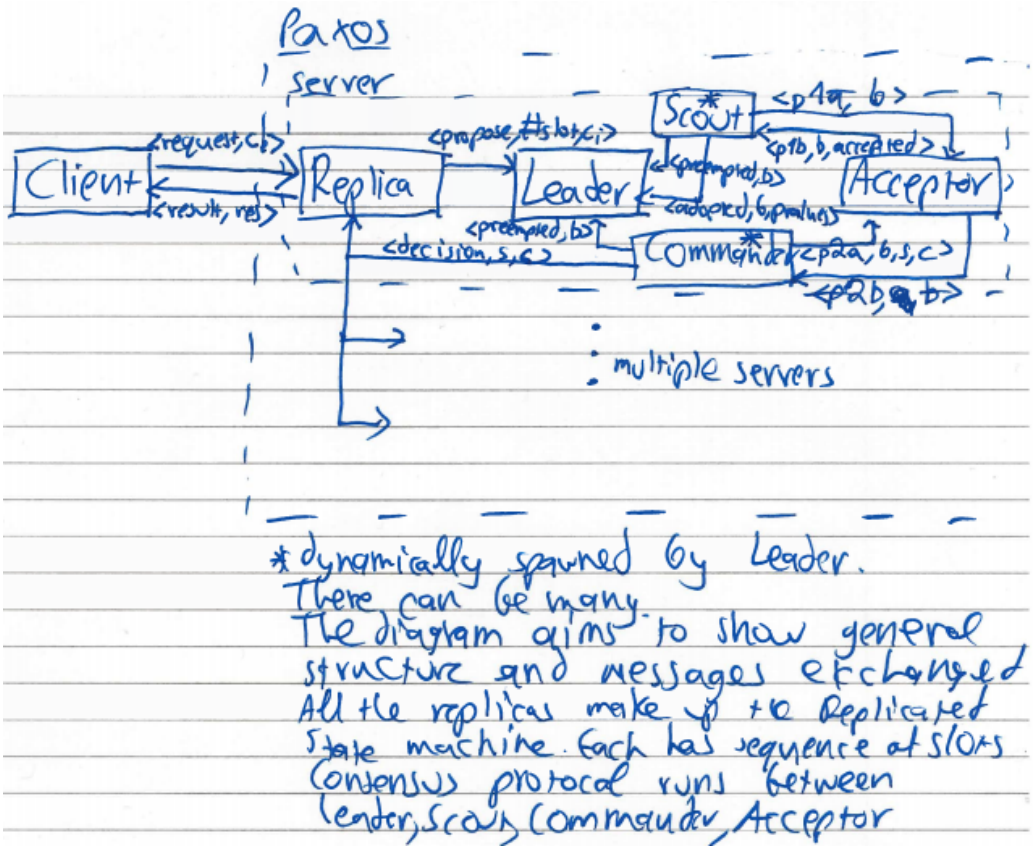
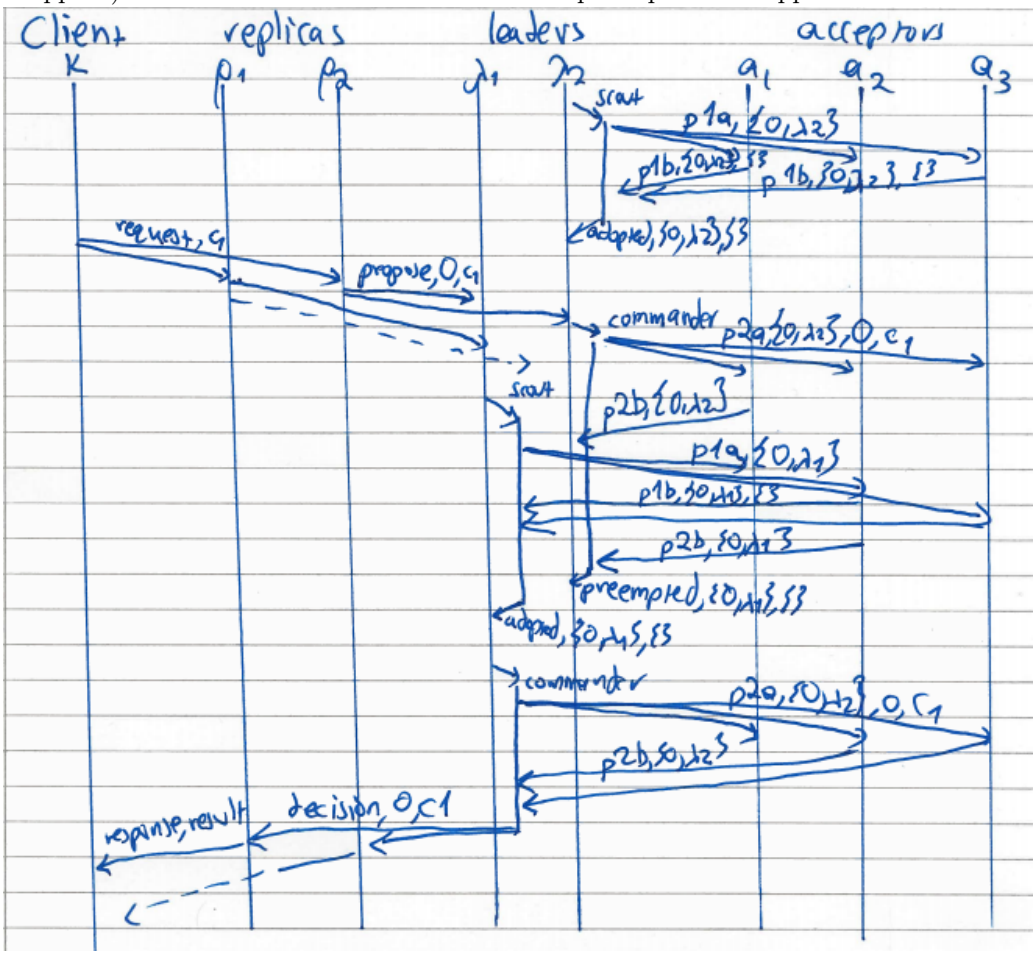


Figure 2: The diagram aims to show how message passing is done between the components to execute a request by a client κ. Time increases as we go down the diagram. In this case we assume that the client is 1 for simplicity of the diagram. We also assume the client κ broadcasts the request to both replicas $\rho 1$, $\rho 2$ (This is to show how the scout, commander are spawned and how proposing (p1a, p1b), committing (p2a, p2b) and preemption happens). We also assume that $\lambda 1 > \lambda 2$ for the preemption to happen



Testing

For testing we try to vary how many commanders and scouts are spawned as well as the window-sizes and the client request sending rate. These can be modified either inside the Makefile (specific configuration, number of servers, number of clients) or configuration.ex (rest of the parameters). We try to also count the number of commanders and scouts spawned and also do some tests on a Docker network using the provided docker-compose.yml.

Testing - Local

Our first testing configuration is the default one, with 3 servers, 2 clients and window-size 100. As we can see, it finishes in 10 seconds, with all the client requests (which are sent round robin to servers, 332, 334, 334) updated in the replicas (1000 updates).

```
time = 1000  updates done = [{1, 208}, {2, 208}, {3, 208}]
time = 1000 requests seen = [{1, 106}, {2, 108}, {3, 108}]
time = 1000 commanders seen = [{1, 8600}, {2, 8858}, {3, 8692}]
time = 1000 scouts seen = [{1, 212}, {2, 215}, {3, 192}]
[...]
time = 10000  updates done = [{1, 1000}, {2, 1000}, {3, 1000}]
time = 10000 requests seen = [{1, 332}, {2, 334}, {3, 334}]
time = 10000 commanders seen = [{1, 25067}, {2, 25174}, {3, 25042}]
time = 10000 scouts seen = [{1, 237}, {2, 241}, {3, 217}]
```

For the case when we set the window to 10 (besides that keep the same configuration as above), it will not modify it significantly. For this case, it was slightly more than 8 seconds, but on average it will be the same as above. The reason for this is the low number of requests, servers and clients.

```
time = 1000  updates done = [{1, 113}, {2, 113}, {3, 113}]
time = 1000 requests seen = [{1, 107}, {2, 108}, {3, 108}]
time = 1000 commanders seen = [{1, 9244}, {2, 9204}, {3, 9251}]
time = 1000 scouts seen = [{1, 198}, {2, 187}, {3, 189}]
[...]
time = 83000  updates done = [{1, 1000}, {2, 1000}, {3, 1000}]
time = 83000 requests seen = [{1, 332}, {2, 334}, {3, 334}]
time = 83000 commanders seen = [{1, 148256}, {2, 149975}, {3, 150775}]
time = 83000 scouts seen = [{1, 467}, {2, 454}, {3, 455}]
```

Trying a system with more servers (11 servers and the configuration same as before) - which means that we have more replicas - we can see that it is not updating the replicas as fast as the previous cases. This is the first case where we are meeting livelocks, i.e. the "Ping-Pong" behaviour. We will see how we solved this issue in the optimizations section. There is no reason we need to test it on Docker, it will give us even worse results.

```
time = 1000  updates done = [{1, 12}, {2, 12}, {3, 12}, {4, 12},
{5, 12}, {6, 12}, {7, 12}, {8, 12}, {9, 12}, {10, 12}, {11, 12}]
time = 1000 requests seen = [{1, 26}, {2, 28}, {3, 28}, {4, 28},
{5, 28}, {6, 28}, {7, 28}, {8, 28}, {9, 28}, {10, 28}, {11, 28}]
time = 1000 commanders seen = [{1, 14122}, {2, 14614}, {3, 15487},
{4, 14946}, {5, 14867}, {6, 15879}, {7, 15921}, {8, 16895}, {9, 17754}, {10, 17041}, {11, 17135}]
time = 1000 scouts seen = [{1, 3125}, {2, 3099}, {3, 2960},
{4, 2737}, {5, 2755}, {6, 2549}, {7, 2341}, {8, 2198}, {9, 2049}, {10, 1935}, {11, 1790}]
[...]
time = 95000  updates done = [{1, 194}, {2, 194}, {3, 194}, {4, 194}, {5, 194}, {6, 194}, {7, 194},
{8, 194}, {9, 194}, {10, 194}, {11, 194}]
time = 95000 requests seen = [{1, 90}, {2, 92}, {3, 92}, {4, 92}, {5, 92}, {6, 92}, {7, 90},
{8, 90}, {9, 90}, {10, 90}, {11, 90}]
time = 95000 commanders seen = [{1, 737219}, {2, 756150}, {3, 760944}, {4, 752132}, {5, 765831},
{6, 787774}, {7, 762723}, {8, 754350}, {9, 782859}, {10, 775191}, {11, 770893}]
time = 95000 scouts seen = [{1, 8906}, {2, 8663}, {3, 8404}, {4, 7870}, {5, 7786}, {6, 7480},
{7, 6895}, {8, 6470}, {9, 6195}, {10, 5823}, {11, 5537}]
```

The purpose of this test (5 servers, 4 clients, same configuration as before) is to try a more extreme example that the previous ones. We increase the number of replicas and also the number of clients (hence the number of requests) to see if the basic implementation can handle everything correctly. We notice livelocks in this situation, i.e. the databases do not get the sent messages even after a long period of time.

```
time = 1000  updates done = [{1, 79}, {2, 79}, {3, 79}, {4, 79}, {5, 79}]
time = 1000 requests seen = [{1, 125}, {2, 128}, {3, 128}, {4, 127}, {5, 126}]
time = 1000 commanders seen = [{1, 28287}, {2, 27765}, {3, 27936}, {4, 28027}, {5, 28223}]
time = 1000 scouts seen = [{1, 701}, {2, 678}, {3, 645}, {4, 565}, {5, 529}]
[...]
time = 114000  updates done = [{1, 681}, {2, 681}, {3, 681}, {4, 681}, {5, 681}]
time = 114000 requests seen = [{1, 400}, {2, 400}, {3, 400}, {4, 400}, {5, 400}]
time = 114000 commanders seen = [{1, 306350}, {2, 304823}, {3, 304090}, {4, 303639}, {5, 303379}]
time = 114000 scouts seen = [{1, 1150}, {2, 1131}, {3, 1086}, {4, 995}, {5, 953}]

time = 115000  updates done = [{1, 681}, {2, 681}, {3, 681}, {4, 681}, {5, 681}]
time = 115000 requests seen = [{1, 400}, {2, 400}, {3, 400}, {4, 400}, {5, 400}]
time = 115000 commanders seen = [{1, 308693}, {2, 307166}, {3, 306433}, {4, 305201}, {5, 304941}]
time = 115000 scouts seen = [{1, 1153}, {2, 1134}, {3, 1089}, {4, 997}, {5, 955}]
```

Something that we have not tried so far is to make many requests to a few number of servers (3 servers, 10 clients, same configuration). Trying out our system shows how slow the updates are done and how much this system is "starving" to make updates, but it is increasing the commanders and scouts on every second.

```
 mix run --no-halt -e Paxos.main 1 single 3 10
time = 1000  updates done = [{1, 374}, {2, 374}, {3, 374}]
time = 1000 requests seen = [{1, 521}, {2, 528}, {3, 525}]
time = 1000 commanders seen = [{1, 7933}, {2, 7942}, {3, 8459}]
time = 1000 scouts seen = [{1, 124}, {2, 126}, {3, 115}]
[...]
time = 935000  updates done = [{1, 1203}, {2, 1203}, {3, 1203}]
time = 935000 requests seen = [{1, 1660}, {2, 1670}, {3, 1670}]
time = 935000 commanders seen = [{1, 970145}, {2, 967317}, {3, 968082}]
time = 935000 scouts seen = [{1, 987}, {2, 984}, {3, 973}]
```

Testing - Docker

An important observation for both the basic and optimized versions is that testing our Distributed System on Docker will take much more time than on the local machines. The reason for this issue is that now we have to transmit messages over links between the containers that our system creates, resulting in an expected overhead.
Trying out the first example from local (3 servers, 2 clients, window 100), the average waiting time is about 4-5 magnitudes greater than local. This is expected, using the observation from above. A good sign is that it is still managing to finish, i.e. update all the replicas with all the requests.

```
Attaching to client1, client2, server1, server3, server2, paxos
paxos | time = 1000  updates done = [{1, 170}, {2, 170}, {3, 170}]
paxos | time = 1000 requests seen = [{1, 99}, {2, 100}, {3, 100}]
paxos | time = 1000 commanders seen = [{1, 2764}, {2, 2694}, {3, 2663}]
paxos | time = 1000 scouts seen = [{1, 123}, {2, 103}, {3, 87}]
[...]
paxos | time = 47000  updates done = [{1, 1000}, {2, 1000}, {3, 1000}]
paxos | time = 47000 requests seen = [{1, 332}, {2, 334}, {3, 334}]
paxos | time = 47000 commanders seen = [{1, 39734}, {2, 43352}, {3, 42143}]
paxos | time = 47000 scouts seen = [{1, 95}, {2, 92}, {3, 83}]
```

Trying the same test as above, but with a window of 20 slows down the system even more. Also, at this stage, after 10 seconds, livelocks happen really often.

```
paxos | time = 1000  updates done = [{1, 70}, {2, 70}, {3, 70}]
paxos | time = 1000 requests seen = [{1, 101}, {2, 101}, {3, 101}]
paxos | time = 1000 commanders seen = [{1, 2158}, {2, 2244}, {3, 2272}]
paxos | time = 1000 scouts seen = [{1, 109}, {2, 92}, {3, 101}]
[...]
paxos | time = 36000  updates done = [{1, 641}, {2, 641}, {3, 641}]
paxos | time = 36000 requests seen = [{1, 332}, {2, 334}, {3, 334}]
paxos | time = 36000 commanders seen = [{1, 28047}, {2, 27894}, {3, 28337}]
paxos | time = 36000 scouts seen = [{1, 180}, {2, 168}, {3, 168}]
paxos |
```

```
paxos | time = 37000  updates done = [{1, 641}, {2, 641}, {3, 641}]
paxos | time = 37000 requests seen = [{1, 332}, {2, 334}, {3, 334}]
paxos | time = 37000 commanders seen = [{1, 28848}, {2, 28695}, {3, 28337}]
paxos | time = 37000 scouts seen = [{1, 181}, {2, 169}, {3, 168}]
[...]
paxos | time = 98000  updates done = [{1, 783}, {2, 783}, {3, 783}]
paxos | time = 98000 requests seen = [{1, 332}, {2, 334}, {3, 334}]
paxos | time = 98000 commanders seen = [{1, 64291}, {2, 64852}, {3, 62780}]
paxos | time = 98000 scouts seen = [{1, 226}, {2, 213}, {3, 211}]
```

Optimizations

The algorithm describe in the paper was not practical. Using the extensions proposed in section 4 of the paper and a few ideas we had we tried to re-engineer it and make it more efficient. We below discuss about the optimizations we attempted.

Custom optimization

An important starting point was making the requests set a :queue (Erlang queue). Thus, we make sure that we enforce the ordering in which we add/remove requests. Then, we also observed that there is no need for the Scout to send all the pvalues to the Leader. In the basic implementation, these pvalues are filtered using pmax function (see definition in the paper), so there is no reason why we cannot filter them before sending the set through the network to the Leader. This way we reduce the amount of data sent. We haven't tested this thoroughly due to time limitations but we are confident is correct. Instead of including it in our direct submission with include a subfolder in the project called *custom_optimisation* that includes this optimization on top of the other ones

Randomness to avoid Livelocks

Maybe one of the most important optimizations, that improved the computation time by almost a magnitude, is adding some randomness sleep time when getting preempted to reduce "Ping-Ponging" and livelocks. A "Ping-Pong" situation is when a leader A gets preempted by a leader B, spawns a new Scout and then eventually preempting leader B. This is possible to be repeated indefinitely if the system doesn't get eventually synchronous. We aim to reduce this livelock effect, where we see no progress even though the states of the leaders change.
Our solution to this blocking issue is based on introducing some "asymmetry", that is, after a Leader is preempted, and before it will create a new Scout, we make the process wait (Process.sleep()) for a random number of milliseconds between 0 and 1000 (we obtained this quantity by extensively testing our system against cases that were previously getting into livelocks - see Basic Implementations for details).
The result was actually better than expected, and we will see and explain the improvements in the Testing section below.

State reduction

The main summary of this optimization is that acceptors only need to maintain most recently accepted value for a slot, as explained in section 4 of the paper. This was actually one of the easiest changes in our codebase. We change accepted from a MapSet of {ballot_number, slot, command} to a Map from {slot} to {ballot_number, slot, command}. Thus, when adding a new entry in accepted, if we already had a tuple for the newly received slot, we simply override its value in the Map.

Collocation

For collocation each replica only sends requests to 1 leader. When the leader is not active we forward the request to the leader that preempted him. This can be thought of traversing edges of who preempted who trying to find the active leader (which is not guaranteed to terminate either). If there is no such leader (because we didn't get active and preempted yet) then it's necessary to broadcast to all leaders. We stopped implementing collocation (even though the initial implementation of collocation can be found at leader.ex:next function) because the following issue arose. When the leader fails, the replica needs to know about it and broadcast the request to everyone. Therefore by naively removing broadcasting in replica.ex:propose function we remove one of the most important properties of our system which is fault tolerance. Also to enable the collocation feature we have to further implement failure detection in a sophisticated way to avoid dropping messages. We concluded this was very abstractly described in the paper and maybe a bit hard to engineer in practice.

Adding a reconfiguration command

We didn't fully finish adding the reconfiguration command but up to this point we just try to handle it uniformly within paxos. Full implementation would also require creating new leaders and acceptors outside the servers.

Testing - Local

The testing below proves that optimizations are extremely needed and helpful. Let's start we the default example, 3 servers, 2 clients, window 100. It is finishing in 4 seconds, that is half the time we got for the same test in the unoptimized version.

```
time = 1000  updates done = [{1, 323}, {2, 323}, {3, 323}]
time = 1000 requests seen = [{1, 107}, {2, 108}, {3, 108}]
time = 1000 commanders seen = [{1, 279}, {2, 564}, {3, 62}]
time = 1000 scouts seen = [{1, 3}, {2, 3}, {3, 1}]
[...]
time = 4000  updates done = [{1, 1000}, {2, 1000}, {3, 1000}]
time = 4000 requests seen = [{1, 332}, {2, 334}, {3, 334}]
time = 4000 commanders seen = [{1, 279}, {2, 2385}, {3, 1545}]
time = 4000 scouts seen = [{1, 7}, {2, 5}, {3, 4}]
```

Using a window of 10, the same comments as above are valid.

```
time = 1000  updates done = [{1, 323}, {2, 323}, {3, 323}]
time = 1000 requests seen = [{1, 107}, {2, 108}, {3, 108}]
time = 1000 commanders seen = [{1, 474}, {2, 277}, {3, 94}]
time = 1000 scouts seen = [{1, 3}, {2, 2}, {3, 1}]
[...]
time = 4000  updates done = [{1, 1000}, {2, 1000}, {3, 1000}]
time = 4000 requests seen = [{1, 332}, {2, 334}, {3, 334}]
time = 4000 commanders seen = [{1, 3119}, {2, 2877}, {3, 2400}]
time = 4000 scouts seen = [{1, 8}, {2, 5}, {3, 4}]
```

We think that this case (11 servers, 2 clients, window 100) shows indeed the power that the optimized system has gained. We do not livelock anymore in this case and we manage to update all the replicas really fast. Our previous implementation was not even close to the final result!

```
time = 1000  updates done = [{1, 304}, {2, 304}, {3, 304}, {4, 304}, {5, 304},
{6, 304}, {7, 304}, {8, 304}, {9, 304}, {10, 304}, {11, 304}]
time = 1000 requests seen = [{1, 26}, {2, 28}, {3, 28}, {4, 28}, {5, 28}, {6, 28},
{7, 28}, {8, 28}, {9, 28}, {10, 28}, {11, 26}]
time = 1000 commanders seen = [{1, 1}, {2, 1}, {3, 1}, {4, 277}, {5, 233}, {6, 129},
{7, 279}, {8, 253}, {9, 1}, {10, 1}, {11, 324}]
time = 1000 scouts seen = [{1, 2}, {2, 2}, {3, 3}, {4, 3}, {5, 3}, {6, 2}, {7, 3},
{8, 3}, {9, 2}, {10, 2}, {11, 3}]
[...]
time = 4000  updates done = [{1, 1000}, {2, 1000}, {3, 1000}, {4, 1000}, {5, 1000},
{6, 1000}, {7, 1000}, {8, 1000}, {9, 1000}, {10, 1000}, {11, 1000}]
time = 4000 requests seen = [{1, 90}, {2, 92}, {3, 92}, {4, 92}, {5, 92}, {6, 92},
{7, 90}, {8, 90}, {9, 90}, {10, 90}, {11, 90}]
time = 4000 commanders seen = [{1, 1}, {2, 1696}, {3, 1}, {4, 1277}, {5, 3645},
{6, 129}, {7, 1120}, {8, 884}, {9, 1}, {10, 1282}, {11, 455}]
time = 4000 scouts seen = [{1, 7}, {2, 7}, {3, 8}, {4, 9}, {5, 9}, {6, 7}, {7, 7},
{8, 8}, {9, 6}, {10, 6}, {11, 6}]
```

The comments from the above test are also valid for this test (5 servers, 4 clients, same configuration).

```
time = 1000  updates done = [{1, 626}, {2, 626}, {3, 626}, {4, 626}, {5, 626}]
time = 1000 requests seen = [{1, 125}, {2, 128}, {3, 128}, {4, 127}, {5, 126}]
time = 1000 commanders seen = [{1, 1}, {2, 1}, {3, 1299}, {4, 1}, {5, 1163}]
time = 1000 scouts seen = [{1, 2}, {2, 2}, {3, 4}, {4, 3}, {5, 3}]
[...]
time = 9000  updates done = [{1, 2000}, {2, 2000}, {3, 2000}, {4, 2000}, {5, 2000}]
time = 9000 requests seen = [{1, 400}, {2, 400}, {3, 400}, {4, 400}, {5, 400}]
time = 9000 commanders seen = [{1, 3006}, {2, 1}, {3, 2508}, {4, 721}, {5, 4822}]
time = 9000 scouts seen = [{1, 5}, {2, 6}, {3, 7}, {4, 6}, {5, 6}]
```

We were actually impressed with the results of this test. With many requests (5000 in total) from 10 different clients, we managed to get our system to the required final state (3 servers) after waiting a bit less than 2 minutes!

```
time = 1000  updates done = [{1, 1156}, {2, 1156}, {3, 1156}]
time = 1000 requests seen = [{1, 505}, {2, 512}, {3, 508}]
time = 1000 commanders seen = [{1, 1159}, {2, 1}, {3, 1561}]
time = 1000 scouts seen = [{1, 2}, {2, 2}, {3, 2}]
[...]
time = 966000  updates done = [{1, 5000}, {2, 5000}, {3, 5000}]
time = 966000 requests seen = [{1, 1660}, {2, 1670}, {3, 1670}]
time = 966000 commanders seen = [{1, 1493338}, {2, 1586096}, {3, 1613692}]
time = 966000 scouts seen = [{1, 447}, {2, 395}, {3, 368}]
```

Testing - Docker

For 3 servers, 2 clients, window 100, we get the same behaviour in Docker as for the local machine.

```
paxos | time = 1000  updates done = [{1, 302}, {2, 302}, {3, 302}]
paxos | time = 1000 requests seen = [{1, 100}, {2, 101}, {3, 101}]
paxos | time = 1000 commanders seen = [{2, 303}, {3, 98}]
paxos | time = 1000 scouts seen = [{1, 1}, {2, 2}, {3, 1}]
[...]
paxos | time = 4000  updates done = [{1, 1000}, {2, 1000}, {3, 1000}]
paxos | time = 4000 requests seen = [{1, 332}, {2, 334}, {3, 334}]
paxos | time = 4000 commanders seen = [{2, 2934}, {3, 2271}]
paxos | time = 4000 scouts seen = [{1, 6}, {2, 5}, {3, 4}]
```

Same comment as above, but for a window of 10.

```
paxos | time = 1000  updates done = [{1, 300}, {2, 300}, {3, 300}]
paxos | time = 1000 requests seen = [{1, 99}, {2, 101}, {3, 100}]
paxos | time = 1000 commanders seen = [{2, 301}, {3, 67}]
paxos | time = 1000 scouts seen = [{1, 2}, {2, 2}, {3, 1}]
[...]
paxos | time = 4000  updates done = [{1, 1000}, {2, 1000}, {3, 1000}]
paxos | time = 4000 requests seen = [{1, 332}, {2, 334}, {3, 334}]
paxos | time = 4000 commanders seen = [{2, 2758}, {3, 2360}]
paxos | time = 4000 scouts seen = [{1, 6}, {2, 5}, {3, 4}]
```

Using 11 servers and 2 clients, we can see that the system on Docker is slower, but is expected to finish since the commanders and scouts keep on increasing.

```
paxos | time = 1000  updates done = [{1, 177}, {2, 178}, {3, 177}, {4, 175}, {5, 177},
{6, 178}, {7, 175}, {8, 178}, {9, 178}, {10, 178}, {11, 177}]
paxos | time = 1000 requests seen = [{1, 17}, {2, 20}, {3, 19}, {4, 19}, {5, 19},
{6, 19}, {7, 18}, {8, 18}, {9, 18}, {10, 18}, {11, 18}]
paxos | time = 1000 commanders seen = [{1, 1}, {7, 182}, {9, 106}]
paxos | time = 1000 scouts seen = [{1, 1}, {2, 2}, {3, 2}, {4, 1}, {5, 1}, {6, 1},
{7, 2}, {8, 1}, {9, 1}, {10, 1}, {11, 1}]
[...]
paxos | time = 114000  updates done = [{1, 813}, {2, 813}, {3, 813}, {4, 813},
{5, 813}, {6, 813}, {7, 813}, {8, 813}, {9, 813}, {10, 813}, {11, 813}]
paxos | time = 114000 requests seen = [{1, 90}, {2, 92}, {3, 92}, {4, 92}, {5, 92},
{6, 92}, {7, 90}, {8, 90}, {9, 90}, {10, 90}, {11, 90}]
paxos | time = 114000 commanders seen = [{1, 22670}, {2, 34403}, {3, 32549},
{4, 35905}, {5, 44091}, {6, 45552}, {7, 49973}, {8, 50211}, {9, 50119}, {10, 29357}, {11, 26913}]
paxos | time = 114000 scouts seen = [{1, 153}, {2, 146}, {3, 145}, {4, 131}, {5, 137},
{6, 125}, {7, 115}, {8, 109}, {9, 103}, {10, 154}, {11, 152}]
```

Tolerating failures

Since one of Paxos core objectives is failure tolerance we conducted a few tests to observe whether our implementation is fault tolerant. As described by the paper, for the system to be tolerant to f failures we need $f + 1$ replicas, $f + 1 leaders$ and $2 * f + 1$ acceptors (to still be able to acquire a majority of

$f + 1$ acceptors in case of f failures happening) We put the failure tests as different configurations in configuration.ex so running the tests on requires changing the variables in the **Makefile** (Note: we kept the window size to 100 for these tests) For killing the processes we added some extra code in paxos.ex (lines 28-52) that kills the number of acceptors/leaders/replicas we specify in configuration.ex after 2 seconds.

Failing acceptors

SERVERS = 7, CLIENTS = 2, CONFIG = 4
Observe we tolerate acceptor failures 3(f) failures, 7(2 * f + 1) acceptors

```
time = 1000  updates done = [{1, 305}, {2, 305}, {3, 305}, {4, 305}, {5, 305}, {6, 305}, {7, 305}]
time = 1000 requests seen = [{1, 42}, {2, 44}, {3, 44}, {4, 44}, {5, 44}, {6, 44}, {7, 44}]
time = 1000 commanders seen = [{1, 308}, {2, 262}, {3, 2}, {4, 308}, {5, 2}, {6, 2}, {7, 234}]
time = 1000 scouts seen = [{1, 3}, {2, 3}, {3, 3}, {4, 3}, {5, 1}, {6, 2}, {7, 2}]

[...]

time = 7000  updates done = [{1, 960}, {2, 960}, {3, 960}, {4, 960}, {5, 960}, {6, 960}, {7, 960}]
time = 7000 requests seen = [{1, 142}, {2, 144}, {3, 144}, {4, 144}, {5, 142}, {6, 142}, {7, 142}]
time = 7000 commanders seen = [{1, 4012}, {2, 1464}, {3, 2161}, {4, 1562}, {5, 2584}, {6, 3106}, {7, 4105}]
time = 7000 scouts seen = [{1, 11}, {2, 11}, {3, 10}, {4, 11}, {5, 8}, {6, 8}, {7, 7}]

time = 8000  updates done = [{1, 961}, {2, 1000}, {3, 1000}, {4, 1000}, {5, 1000}, {6, 1000}, {7, 961}]
time = 8000 requests seen = [{1, 142}, {2, 144}, {3, 144}, {4, 144}, {5, 142}, {6, 142}, {7, 142}]
time = 8000 commanders seen = [{1, 4012}, {2, 2464}, {3, 3161}, {4, 1562}, {5, 2584}, {6, 4106}, {7, 5105}]
time = 8000 scouts seen = [{1, 12}, {2, 14}, {3, 11}, {4, 12}, {5, 9}, {6, 9}, {7, 8}]
```

SERVERS = 8, CLIENTS = 2, CONFIG = 7
Observe that if we go over the limit the system fails fail 4(f) acceptors out of 8(2 * f)

```
time = 1000  updates done = [{1, 280}, {2, 280}, {3, 280}, {4, 280}, {5, 280}, {6, 280}, {7, 280}, {8, 280}]
time = 1000 requests seen = [{1, 36}, {2, 38}, {3, 38}, {4, 38}, {5, 36}, {6, 36}, {7, 36}, {8, 36}]
time = 1000 commanders seen = [{1, 2}, {2, 284}, {3, 93}, {4, 252}, {5, 84}, {6, 2}, {7, 284}, {8, 192}]
time = 1000 scouts seen = [{1, 2}, {2, 3}, {3, 3}, {4, 3}, {5, 3}, {6, 2}, {7, 3}, {8, 2}]

[...]

time = 76000  updates done = [{1, 516}, {2, 516}, {3, 516}, {4, 516}, {5, 516}, {6, 516}, {7, 516}, {8, 516}]
time = 76000 requests seen = [{1, 124}, {2, 126}, {3, 126}, {4, 126}, {5, 126}, {6, 124}, {7, 124}, {8, 124}]
time = 76000 commanders seen = [{1, 2}, {2, 652}, {3, 93}, {4, 252}, {5, 483}, {6, 2}, {7, 1256}, {8, 1073}]
time = 76000 scouts seen = [{1, 7}, {2, 8}, {3, 7}, {4, 8}, {5, 6}, {6, 5}, {7, 6}, {8, 5}]
```

Failing replicas

If we wish to fail replicas we need to change the round-robin transmitting to broadcasting to all replicas. We should also note that the databases associated with replicas we fail don't update anymore. We are only concerned with the databases of the replicas we don't fail. We can change the round robin to broadcasting by uncommenting lines 30-32 in client.ex and commenting out 34-35.
SERVERS = 6, CLIENTS = 2, CONFIG = 3
Observe we tolerate replica failures 5(f) failures, 6(f + 1) replicas

Observe that the last replica (that we don't kill still reaches 1000 updates). Running this with 4 failures shows that 2 replicas reach 1000 updates etc

```
time = 1000  updates done = [{1, 307}, {2, 307}, {3, 307}, {4, 307}, {5, 307}, {6, 307}]
time = 1000 requests seen = [{1, 308}, {2, 308}, {3, 308}, {4, 308}, {5, 308}, {6, 308}]
time = 1000 commanders seen = [{1, 2}, {2, 2}, {3, 2}, {4, 267}, {5, 2}, {6, 593}]
time = 1000 scouts seen = [{1, 1}, {2, 2}, {3, 2}, {4, 2}, {5, 2}, {6, 2}]

[...]

time = 7000  updates done = [{1, 436}, {2, 451}, {3, 436}, {4, 436}, {5, 437}, {6, 1000}]
time = 7000 requests seen = [{1, 597}, {2, 599}, {3, 597}, {4, 597}, {5, 599}, {6, 1000}]
time = 7000 commanders seen = [{1, 4192}, {2, 1034}, {3, 2085}, {4, 4457}, {5, 6270}, {6, 4131}]
time = 7000 scouts seen = [{1, 13}, {2, 13}, {3, 9}, {4, 10}, {5, 11}, {6, 9}]
```

Failing leaders

SERVERS = 6, CLIENTS = 2, CONFIG = 5
Observe we tolerate leader failures 5(f) failures, 6(f + 1) leaders (Note we need to change collocation for this to make more sense)

```
time = 1000  updates done = [{1, 304}, {2, 304}, {3, 304}, {4, 304}, {5, 304}, {6, 304}]
time = 1000 requests seen = [{1, 304}, {2, 304}, {3, 304}, {4, 304}, {5, 304}, {6, 304}]
time = 1000 commanders seen = [{1, 2}, {2, 2}, {3, 231}, {4, 704}, {5, 2}, {6, 297}]
time = 1000 scouts seen = [{1, 2}, {2, 3}, {3, 3}, {4, 4}, {5, 2}, {6, 3}]

[...]

time = 5000  updates done = [{1, 982}, {2, 982}, {3, 982}, {4, 982}, {5, 978}, {6, 982}]
time = 5000 requests seen = [{1, 1000}, {2, 1000}, {3, 1000}, {4, 1000}, {5, 1000}, {6, 1000}]
time = 5000 commanders seen = [{1, 518}, {2, 2}, {3, 231}, {4, 1255}, {5, 596}, {6, 2571}]
time = 5000 scouts seen = [{1, 5}, {2, 4}, {3, 5}, {4, 5}, {5, 4}, {6, 5}]

time = 6000  updates done = [{1, 1000}, {2, 1000}, {3, 1000}, {4, 1000}, {5, 1000}, {6, 1000}]
time = 6000 requests seen = [{1, 1000}, {2, 1000}, {3, 1000}, {4, 1000}, {5, 1000}, {6, 1000}]
time = 6000 commanders seen = [{1, 518}, {2, 2}, {3, 231}, {4, 1255}, {5, 596}, {6, 2906}]
time = 6000 scouts seen = [{1, 5}, {2, 4}, {3, 5}, {4, 5}, {5, 4}, {6, 5}]
```

Failing a lot of components together

SERVERS = 7, CLIENTS = 2, CONFIG = 6
Observe we tolerate multiple failures combined (4 failures out of 7 replicas, 3 failures out of 7 acceptors, 6 failures out of 7 leaders). We use fewer than 6 replicas to observe the consensus between them. (This test also requires broadcasting by clients)

```
time = 1000  updates done = [{1, 302}, {2, 302}, {3, 302}, {4, 302}, {5, 302}, {6, 302}, {7, 302}]
time = 1000 requests seen = [{1, 306}, {2, 306}, {3, 306}, {4, 306}, {5, 306}, {6, 306}, {7, 306}]
time = 1000 commanders seen = [{1, 195}, {2, 2}, {3, 2}, {4, 335}, {5, 223}, {6, 2}, {7, 18}]
time = 1000 scouts seen = [{1, 3}, {2, 2}, {3, 3}, {4, 3}, {5, 2}, {6, 1}, {7, 1}]

[...]

time = 6000  updates done = [{1, 650}, {2, 650}, {3, 650}, {4, 650}, {5, 1000}, {6, 1000}, {7, 1000}]
time = 6000 requests seen = [{1, 650}, {2, 650}, {3, 650}, {4, 650}, {5, 1000}, {6, 1000}, {7, 1000}]
time = 6000 commanders seen = [{1, 1229}, {2, 2}, {3, 2271}, {4, 1143}, {5, 4492}, {6, 3024}, {7, 2796}]
time = 6000 scouts seen = [{1, 10}, {2, 8}, {3, 9}, {4, 10}, {5, 9}, {6, 8}, {7, 6}]
```

Bibliography

[1] - Robbert Van Renesse, Deniz Altinbuken - Paxos Made Moderately Complex

[2] - Paxos System - *http://paxos.systems/*