Distributed Algorithms: Multi-Paxos Report

Daniel Yung (lty16) Tim Green (tpg16)

February 2019

1 Evaluation

When evaluating our multi-paxos algorithm by ran it locally in a single Elixir node with 3 servers and 2 clients.

1.1 Ballot Selection

For our implementation of multi-paxos, ballots are pairs of { `ballot_number, leader_pid` }. This ensures the ballots being unique and have different precedence between leaders. The leader with the bigger pid value will have precedence over other leaders with smaller pid if the ballot_numbers are the same. When the proposal are sent from the replica to all leaders, the leaders in turns spawns a commander which sends a p2b message to all the acceptors with the leaders current ballot number. Even though all leaders have the same ballot_number, only 1 will be accepted by the acceptor which would be the leader with the largest pid. All the other commanders will get a p2b message with a bigger ballot back and inform its leader to be preempted and go into passive mode. The commander with the biggest ballot will now get a majority acceptor message and send the decision to execute it's client's command. The leaders which are preempted in the previous round will send a new scout with a larger ballot the ensure that it's own ballot will be accepted by the acceptor.

1.2 Ordering of the commands

For each command sent by the clients, all the replicas will execute the same commands in the same order. This strict ordering is achieved by the fact that only 1 commander will send a decision to all the replicas because only the commander that have the majority vote by the acceptor will send a decision. Before executing the command at that time slot, the replicas will check if there are any other commands that were other commands clients would like to perform at that time slot. If there are commands that the clients would like to perform and it is not the same as the one the replica received as decision from the leaders, it will be removed from the list of proposals and into the request so it will be proposed another time.

1.3 Live-lock Prevention

A ping-pong situation is possible in the multi-paxos system which causes an infinite loop and prevents any operations from being executed. An illustration of this can be seen in Figure 9 in the Paxos Made Moderately Complex paper [1]. This is made possible when the following happens:

1. Leader 1 spawned which also spawns a Scout with the initial ballot ({ `0, L1_pid` }). The Scout sends { `:p1a, self(), ballot` } message to all Acceptors to initialise the ballot. The Ballot will be accepted by all Acceptors since any value will be greater than the initial $\perp$ ballot number.

2. Leader 2 is spawned with initial ballot { `0, L2_pid` } which also spawns a Scout and sends an initial { `:p1a, self(), ballot` } message to all Acceptors. Since L2 was spawned after L1 it will have a higher PID and hence { `0, L2_pid` } `>` { `0, L1_pid` } will evaluate to true (due to Elixirs lexicographic ordering of objects), causing L2's ballot to be used in the Acceptor.

3. A new proposals comes from the client, through L1 which spawns a Commander. This Commander sends a `:p2a` message to all Acceptors. The Acceptor returns its current ballot ({ `0, L2_pid` }) through a `:p2b` message which is returned to the Commander. Since { `0, L1_pid` } `!=` { `0, L2_pid` } a decision is not made and a `:preempted` message is sent to L1.

4. The L2 ballot is greater than the L1 ballot so L1 is made inactive and its ballot is increased to { `1, L1_pid` }. A Scout is spawned with the new ballot. This sends a `:p1a` to all Acceptors which increases the global ballot to { `1, L1_pid` }. Since the ballot was accepted, an `:adopted` message is sent to its Leader which then spawns a Commander with the new ballot.

5. During this, another command is send to L2 which spawns a Scout and sends a `:p1a` message to all Acceptors with ballot { `1, L2_pid` }. This is greater than L1's ballot so the Acceptors now take it as the current ballot.

6. Now, when L1's Commander sends a `:p2b` message to Acceptors the ballot has changed and so it will send a `:preempted` message to its Leader L1. Now, no Commander except L1 can get majority and the Leaders will go back and forth increasing their ballots.

To solve this problem we added a `Process.sleep(timeout)` command after a `:preempted` message is received in the Leader. Doing so allows the Leader with the majority to continue its cycle to make a decision without being preempted. Importantly, the sleep command prevents a Scout from being spawned in the minority Leader. This means a `:p1a` message will not be sent and a new (higher) ballot will not be made available to all Acceptors.

2 Experiments

For the following experiments, we turned on debug mode to show more of the internal states of the multi-paxos system. When running debug mode we added a new feature that prints out the amount of commands executed by the clients to show if there are any disparities in number of commands executed by each clients.

2.1 Varying Timeout

When implementing the timeout in our multi-paxos system, we found that choosing the right timeout for the leader when it receives a preempted message had a big impact on the overall lag of the system. For the following experiment we fixed the number of clients and servers to be 2 and 3 respectively and the client send their requests in a round robin fashion.

2.1.1 Experiment 1 Fixed timeout

2.1.2 Results

Time(s)	**Timeout(milliseconds)**		
	5	**10**	**30**
2	0	1	454
4	0	0	1112
6	1	0	1654
8	240	303	1170
10	548	512	583
20	1463	1184	1083
30	3053	2747	2595

Table 1: The total lag for each timeout at a given time

2.1.3 Analysis

In the table above we tested 3 different timeout value. Timeout value of 5 and 10 milliseconds results were not very different to each other however 30 milliseconds gives us a more interesting result.

We can see that by increasing the fixed timeout we effectively increase the lag at the beginning (because most of the leaders in passive mode sleep for longer time) but the lag increases in a slower rate, eventually the lag is smaller than the lower timeout. This is due to the fact that after running a while, the program starts to build up message queue and slows down turnover rate for each decision and increases the likely hood of being stuck in a ping-pong situation that does not progress the algorithm. So the high timeout actually benefits the other leaders to finish up their message queue while a small timeout actually causes the live lock situation.

2.1.4 Experiment 2 MIAD

In TCP connection, to avoid network contention and collision, different hosts usually adopts AIMD where they increasingly more packets, then once a collision has been detected, it will back off multiplicative. In the multi-paxos case, contention is detected by a preempted message, so by referencing the method proposed the Paxos Made Moderately Complex paper [1], leaders should also adopt a similar solution, once a preempted message has been received. The Leader should sleep for a timeout depending on how many preempted messages has been sent before. If a leader has been receiving a few preempted messages, then its timeout should be quite large to ensure the other leaders can finish their proposals.

In this experiment, each leader do not have a static timeout but rather a MIAD (Multiplicative Increase Additive Decrease) timeout. Leaders are initialise with timeout of 0 milliseconds then every time it gets a preempted message it will sleep for `1 + timeout * rate`. While every adopted message sent by the scout will decrease the timeout by a fixed amount.

For this experiment we focus just on the rate of change and kept the fixed decrease amount of timeout to be 5 milliseconds.

On every preempted message:

$$timeout = 1 + round(timeout * rate)$$

On every propose message:

$$timeout = timeout - 5$$

2.1.5 Result

Time(s)	**Rate**					
	1.005	**1.05**	**1.15**	**2**	**3**	**4**
2	0	0	0	0	0	0
4	0	0	0	0	2	0
6	0	0	0	0	2	2
8	2	0	0	1	1	3
10	28	3	5	10	10	10
15	194	172	61	281	330	494
20	696	725	594	810	1025	1012
25	1378	1365	1344	1453	1524	1587
30	2205	2198	2157	2312	2395	2590
35	3084	3070	3047	3244	3217	3423
40	3968	4066	4108	4326	4344	4530

Table 2: The total lag for each rate at a given time

2.1.6 Analysis

In our results we can see that no matter the rate, after 40s there are roughly 4000 operations that are lagging behind and have yet to be completed. However as we can see that increasing the rate also has some effect of the initial lag in the system. With the full data set we collected we can see some spikes that causes the lag to go up and down, a small example is at rate 3 where the lag was 2 at 6 seconds, it went back down to 1 after 2 seconds. The spikes shows us that the algorithm is working properly where the leaders are sleeping enough to let other leaders finish their cycle to make a decision and execute the command.

If the timeout rate is too small, in the case of 1.005, we see that the lag spikes starts very soon after 10 seconds. Because the leaders are not sleeping long enough for the other leaders to finish their proposal which causes the ping-pong situation. But in the case of 3 and 4 where

the rate causes the timeout to be too high, when there are more request building up and more leaders get the preempted message, the timeout becomes too high that sleep is too long and causes less commands to be executed.

In the Paxos Made Moderately Complex paper [1], it suggested that we choose a factor larger than 1 to multiply the timeout with, however does not suggest how much larger. We an see from our results that it should not be too large as it would also cause the productivity of the system to go down, we propose a timeout that is around 1.05 - 1.15 as this has the best result in our experiment.

Finally compared to fixed timeout we found that this multiplicative back off algorithm for timeout has a much better lag performance. This is due to the fact that it increases the timeout when many preempted message are being sent to a leader, therefore it leaves more time for other leaders to finish their proposals, but not too much time wasted as every successful adopted message decreases the timeout for the next time it is preempted.

2.1.7 Experiment 3 AIMD

The paper Paxos Made Moderately Complex [1], although it describes a multiplicative increase and additive decrease, it references a TCP-like algorithm AIMD (Additive Increase Multiplicative Decrease). So we decided to attempt to see if it is better than the one described in the paper. For this experiment we attempt to vary the different fixed increase in timeout every preempted message. Every time the leader gets a proposal, the timeout will be divided into 2 where we will not change for this task.

On every preempted message:

$$timeout = timeout + fixed\_value$$

On every adopted message:

$$timeout = round(timeout/2)$$

2.1.8 Result

Time(s)	**Fixed_Value(milliseconds)**					
	1	**2**	**5**	**20**	**30**	**50**
2	0	0	0	0	0	18
4	1	2	1	0	0	13
6	147	1	2	0	0	0
8	134	2	1	1	1	28
10	26	34	12	2	2	170
15	470	500	87	125	182	742
20	1128	1039	585	634	755	1244
25	2028	1603	1287	1174	1884	2060
30	2891	2214	2171	1837	2919	2799
35	3848	3137	3084	2730	2931	3392
40	4719	3995	3957	3614	4041	4283

Table 3: The total lag for each additive increase

2.1.9 Analysis

For AIMD, the results in general seems to be similar to the MIAD counter part. Hence AIMD is also better than fixed timeout algorithm.

From our table, we find that if the increase in timeout is too low such as 1 and 2 milliseconds, this timeout increase is not enough for other leaders and still causes the live lock which explains the large spike in lag for 1 millisecond increment after 4 to 6 seconds. However as we increase the increment up to 20, we can see it has a good affect on the overall system. Similar to MIAD, the timeout will adjust itself depending on whether or not a certain host is receiving a lot of preempted message or adopted message, and reduces the chance of having the leaders contending each other on the ballot number. While a timeout too large such as 30 - 50 causes the system to lag almost immediately due to the fact that leaders sleep for too long at the beginning even with low contention in the algorithm.

Looking at the results we can see the best result come from 20 milliseconds. Decreasing the timeout to 5 milliseconds has a more similar affect on the lag than increasing the timeout to 30

milliseconds. Thus we can conclude that the ideal range for the increment should be between 5 - 20 milliseconds.

2.2 Different client sending procedure

2.2.1 Experiment

We varied the methods of message sending in the system to analyse the different results.

2.2.2 Result

All results are shown after 10 seconds of execution.

```
:round_robin

time = 10000  updates done = [{1, 2882}, {2, 2882}, {3, 2870}]
time = 10000 requests seen = [{1, 1110}, {2, 1108}, {3, 1108}]
time = 10000    total seen = 3326 max lag = 456
time = 10000        scouts = [{1, 0}, {2, 0}, {3, 0}]
time = 10000    commanders = [{1, 0}, {2, 0}, {3, 0}]
time = 10000        clients = [{1, 4315}, {2, 4319}]

:broadcast

time = 10000  updates done = [{1, 3327}, {2, 3327}, {3, 3327}]
time = 10000 requests seen = [{1, 3327}, {2, 3327}, {3, 3327}]
time = 10000    total seen = 9981 max lag = 6654
time = 10000        scouts = [{1, 0}, {2, 0}, {3, 0}]
time = 10000    commanders = [{1, 0}, {2, 0}, {3, 0}]
time = 10000        clients = [{1, 4992}, {2, 4989}]

:quorum

time = 10000  updates done = [{1, 3182}, {2, 3182}, {3, 3182}]
time = 10000 requests seen = [{1, 2219}, {2, 2217}, {3, 2218}]
time = 10000    total seen = 6654 max lag = 3472
time = 10000        scouts = [{1, 0}, {2, 0}, {3, 0}]
time = 10000    commanders = [{1, 0}, {2, 0}, {3, 0}]
time = 10000        clients = [{1, 4776}, {2, 4770}]
```

2.2.3 Analysis

As can be seen in the results, using the `:broadcast` method for message sending produces similar results to `:round_robin` in terms of updates done but sees approximately 3x more requests seen and has a significantly higher lag. This result makes sense and can be attributed to the increased number of messages send by the clients.

The `:quorum` method also gives significantly more lag than the simple `:round_robin`. Again, this makes sense as round robin only sends one message per client, whereas quorum sends a message to half the servers for every client.

References

[1] R. v. Renesse, D. Altinbuken, “Paxos Made Moderately Complex”, 2015.

Appendices

Appendix A Diagrams

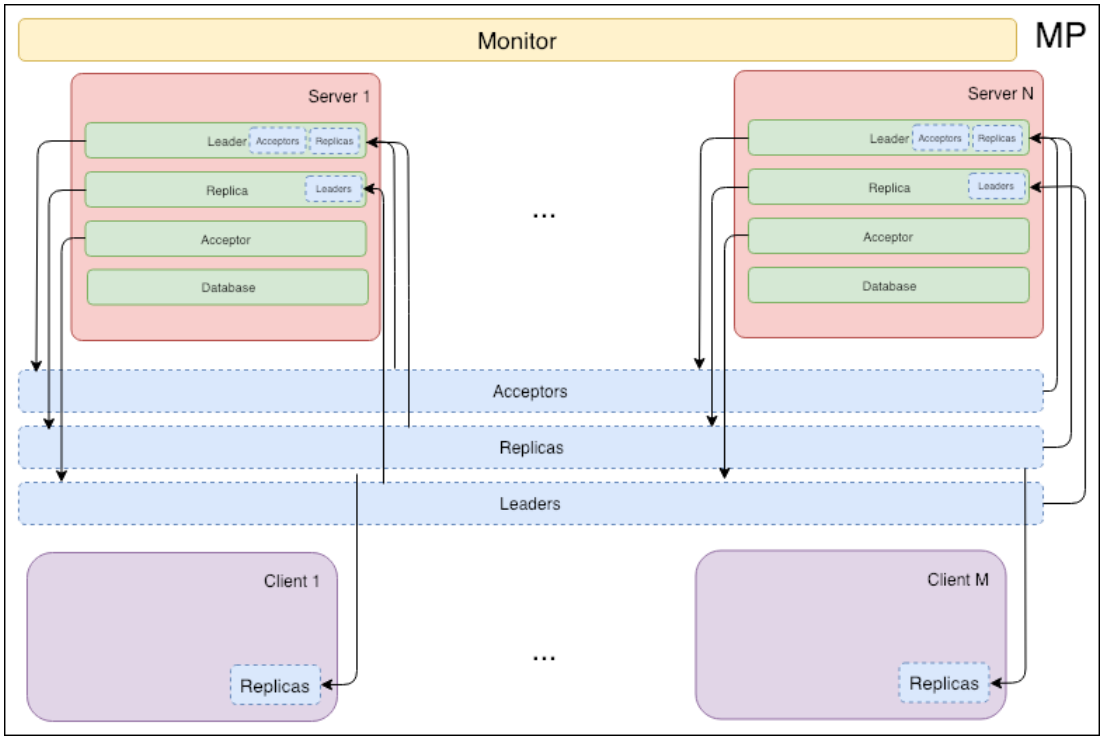

Figure 1: Multi-paxos system overview

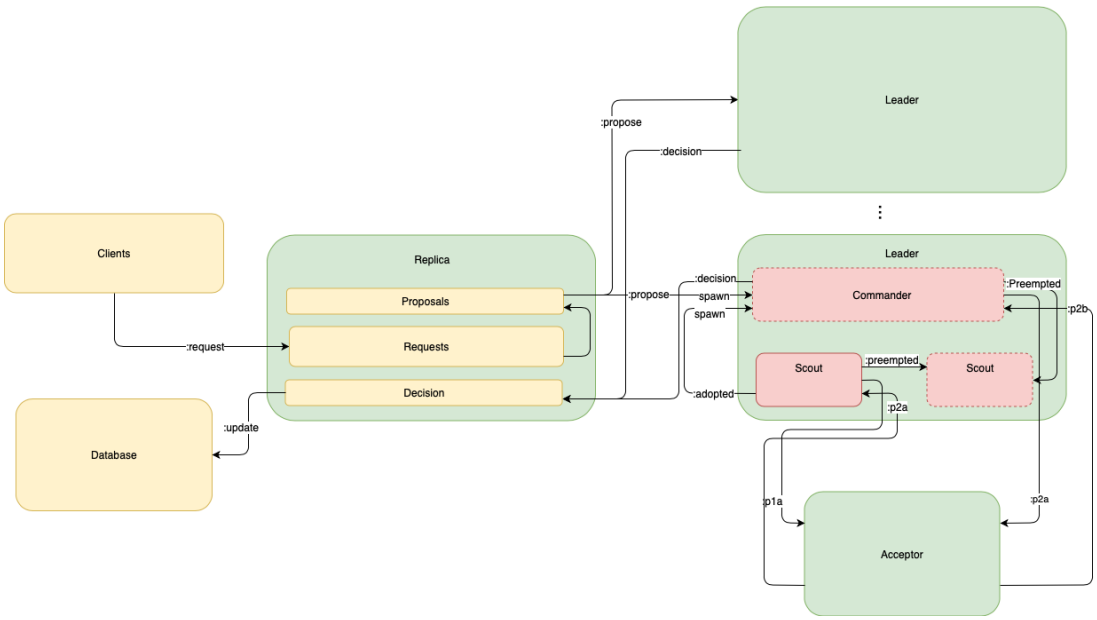

Figure 2: Multi-paxos messaging sequence overview