CO374 Distributed Algorithms

Imperial College London

Department of Computing

Coursework 2 - Multi-Paxos

Authors:
Leszek Nowaczyk CID: 01253901
Alessandro Serena CID: 01188591

Date: February 25, 2019

Contents

1 Introduction

The purpose of this assignment was to implement and evaluate a simple replicated banking service that uses Mulit-Paxos to achieve consensus on the sequence of banking operations.

2 Machine Set Up

All of the local test were ran on a DOC lab machine with the following specs:
- Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz
- 16GB RAM

Docker tests were ran on apache cloudstack VM with the following specs:
- 4 core Intel(R) Xeon(R) CPU E5-2690 0 @ 2.90GHz
- 8GB RAM

3 Part 1 - System Structure

The simulate a distributed banking system, a top level Multi-Paxos node was created. This node spawns new Client nodes, which want to execute transactions on the system and server nodes which act as concurrent servers for the banking system. Multi-Paxos also spawns a monitor process which prints information about the state of the system. Each Server spawns 4 processes - database, replica, leader and acceptor. Additional scout and commander processes are spawned dynamically within the leader process.

The Paxos run starts with a client sending a client_request message to replicas (to how many replicas depends on the `client_send` parameter). Upon receiving this message the replica forwards (adding extra information) the message to the Monitor process and sends a propose message to the leader in that server. The leader then spawns a scout to make a p1a message proposal to the acceptor and receives a p1b message back with the highest ballot number value the acceptor has accepted. If the ballot number it receives is lower then what the scout proposed, the scout sends a preempted message to the leader. Otherwise it sends an adopted message to the leader. The leader then spawns a commander and that sends a p2a message to see if it can get a majority of acceptors to accept a particular command for a slot. if the commander has in the meantime accepted a higher ballot number from some other scout, the p2b message will cause the commander to send a preempted message to the leader. Otherwise if the commander gets a majority of p2b messages in his favour it sends the command as a decision to all replicas, which in turn send an execute message to the corresponding database in that server. For monitoring purposes database send a message to the monitor. Scouts and commanders also send messages to the monitor whenever they are spawned or have finished for debugging purposes. Finally, the monitor also sends a message to itself as specified in the config file to print the status of the system. The full sequence of messages that lead to a decision can be seen in Figure 2.

The system structure and message exchanges can be seen in Figure 1.

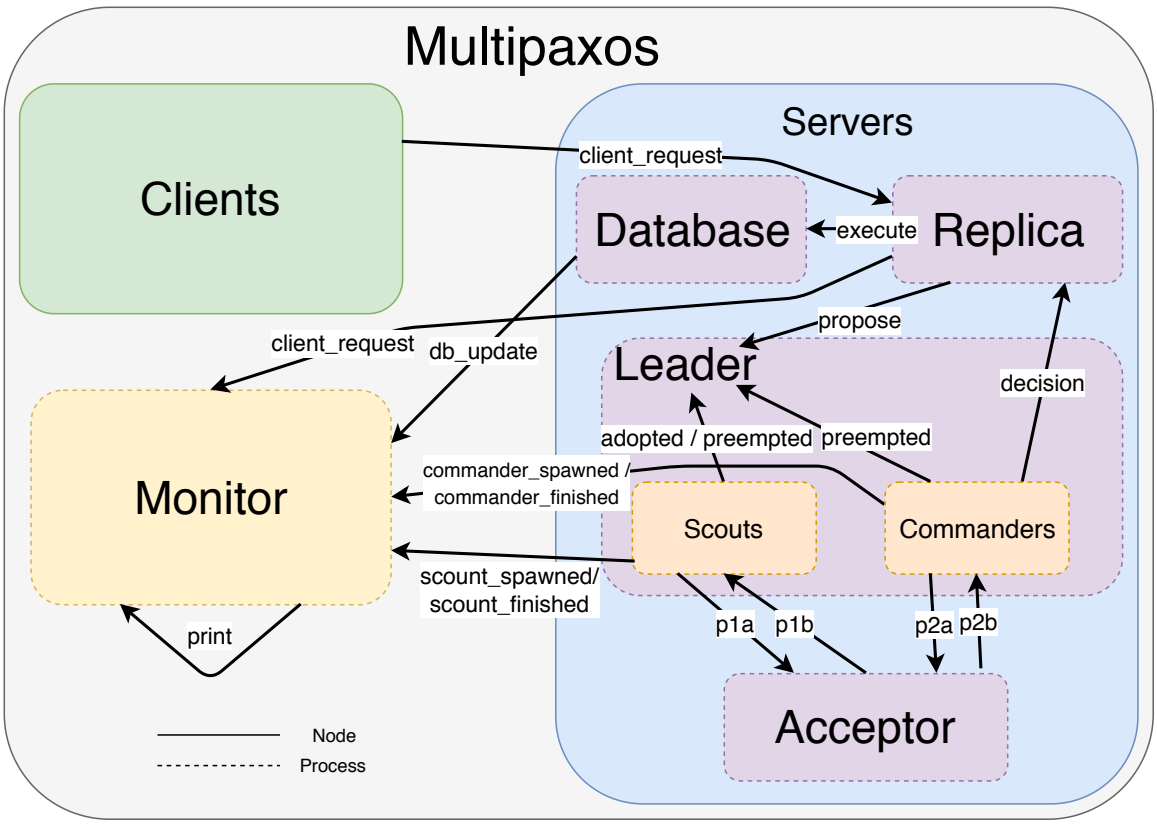

Figure 1: Multi-Paxos Diagram

4 Part 2 - Evaluation

After having implemented all the requested modules, we have proceeded to testing the system.
Each test reported below aims to inspect the correctness of implementation of certain system parameters.[1]

4.1 Avoid live-locks

The first tests that we run were to debug and to check that the regular operation of the algorithm was correctly implemented. We noticed that the system will often encounter a live-lock (i.e. when the components are sending messages to each other but no consensus is being reached). Another phenomenon that we studied was that, when the system noticeably slows down and it seems that a live-lock might have occurred, a high number of commanders is usually spawned by a certain server. At this point the system generally performs many transactions.
In order to solve the live-lock problem, we modified `leader` to wait for a random delay (the

[1]For the full local test results see results_local.txt file and for the docker results see results_docker.txt file in the root directory of the sumbission.

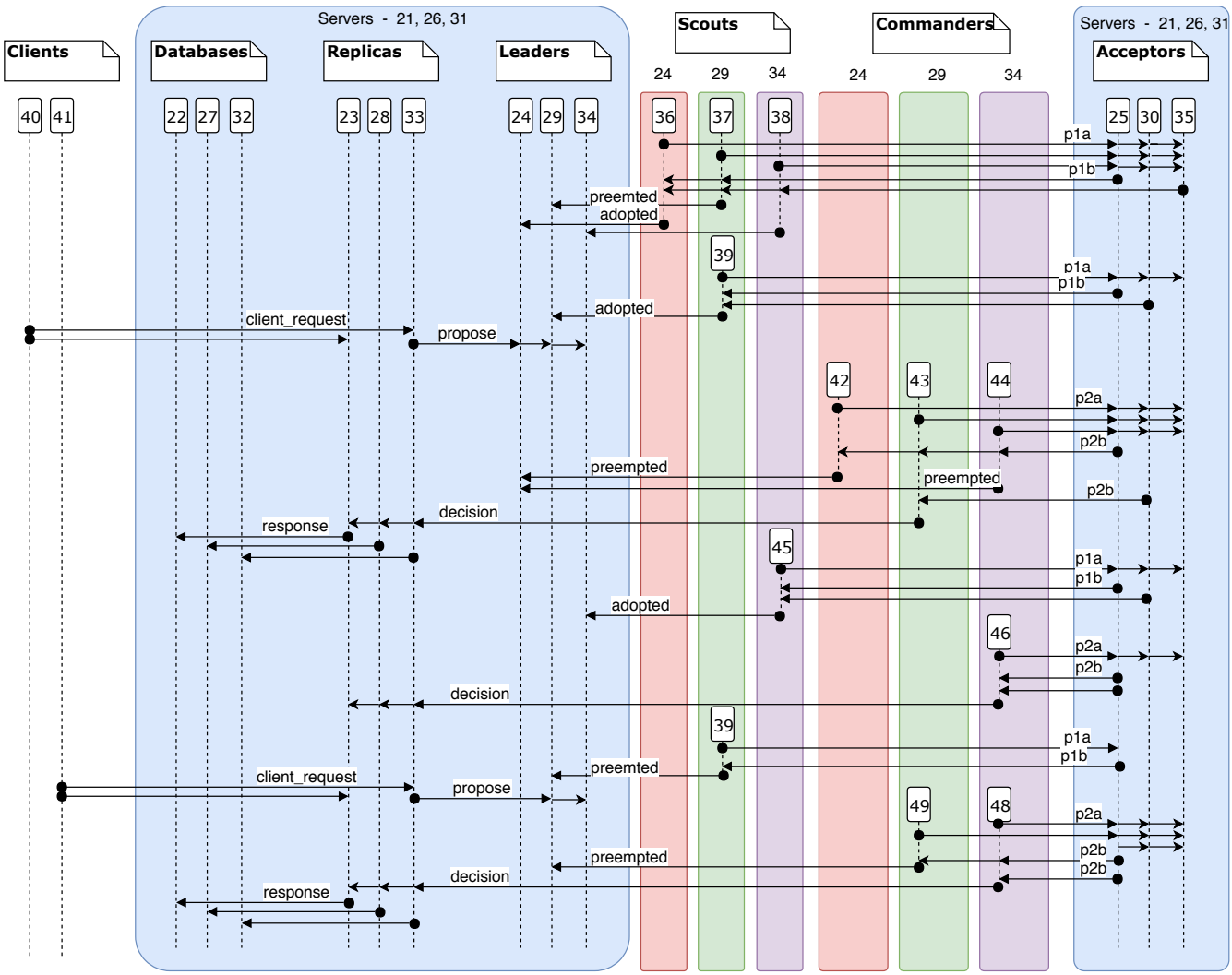

Figure 2: Message Diagram

upper bound of the random number generation can be specified in `configuration.ex`) when it receives a `:preempted` message from its `scout` or `commander`.

4.2 `Client_send`

Since the goal of each client in Multi-Paxos is to update the *state* of the system (in this case the values held in the databases) accordingly to its command, it is in the clients' best interest that the command request is successfully received by at least one of the replicas.
A client sends `client_requests` to one or more replicas, depending on the `client_send` parameter contained in `configuration.ex`. It is sent only to one replica (round robin), to the majority of them (quorum) or to all the replicas (broadcast).
Theoretically, having clients to send their requests to as many replicas as possible should improve the system reliability because we reduce the probability of requests not being considered by the any of the servers. In practice, the sending method implemented by the clients

does not affect the system reliability because we are not using unreliable links between components and processes are not crashing, implying therefore that all messages will always be received.
Following what described above, from the tests we can see that reliability does not improve and that actually round robin yields the fastest execution. This seems counter-intuitive unless we consider that when each client is communicating its (slot, command) pair only to one replica, there are less messages waiting in the queue to be read and, more importantly, there will not be multiple replicas clashing with each other when proposing the same command for the same slot, wasting round trips.

4.3 Number of clients and servers

Next, we decided to test the scalability of the algorithm and of our implementation by varying the number of servers and clients in the system.
In particular, one of the tests was simulating a possible scenario of application for Multi-Paxos: a distributed banking database made of 4 servers, and 150 agents (i.e. clients) that can generate transactions and send them to the bank database.
The experiment ran successfully, with the servers being able to reach consensus on all the transactions.

```
4 SERVERS - 150 CLIENTS
   debug_level: 0,   # debug level 0
   docker_delay: 1_000,  # time (ms) to wait for containers to start up
   max_requests: 100,  # max requests each client will make
   client_sleep: 3,  # time (ms) to sleep before sending new request
   client_stop: 60_000,  # time (ms) to stop sending further requests
   client_send: :quorum, # :round_robin, :quorum or :broadcast
   n_accounts: 100,  # number of active bank accounts
   max_amount: 1_000,  # max amount moved between accounts
   print_after: 1_000,  # print transaction log summary every print_after
      red↪ msecs
   crash_server: %{},
   window: 10, # when window = 1, it is equivelent to not having a window
   delay: 100000 # random delay from 0 to 'delay' when the leader is
      red↪ preemted (0 means no delay)
   ------------------------------------------------------------------
   time = 5000 updates done = [{1, 9758}, {2, 9733}, {3, 9610}, {4, 9630}]
   time = 5000 requests seen = [{1, 7500}, {2, 7500}, {3, 7500}, {4, 7500}]
   time = 5000 total seen = 15000 current lag = 5390, max lag = 9086
   time = 5000 scouts = [{1, 0}, {2, 0}, {3, 0}, {4, 0}]
   time = 5000 commanders = [{1, 0}, {2, 0}, {3, 0}, {4, 0}]

   time = 14000 updates done = [{1, 13015}, {2, 13032}, {3, 13027}, {4,
```

```
   red↪ 12703}]
time = 14000 requests seen = [{1, 7500}, {2, 7500}, {3, 7500}, {4,
   red↪ 7500}]
time = 14000 total seen = 15000 current lag = 2297, max lag = 9086
time = 14000 scouts = [{1, 0}, {2, 0}, {3, 0}, {4, 0}]
time = 14000 commanders = [{1, 0}, {2, 0}, {3, 0}, {4, 0}]

time = 27000 updates done = [{1, 15000}, {2, 15000}, {3, 15000}, {4,
   red↪ 15000}]
time = 27000 requests seen = [{1, 7500}, {2, 7500}, {3, 7500}, {4,
   red↪ 7500}]
time = 27000 total seen = 15000 current lag = 0, max lag = 9086
time = 27000 scouts = [{1, 0}, {2, 0}, {3, 0}, {4, 0}]
time = 27000 commanders = [{1, 0}, {2, 0}, {3, 0}, {4, 0}]
```

4.4 Sending rate

By testing different sending rates we have been able to test the system adaptability and performance for various different situations.
The general pattern that we have observed is that with high rates of client requests, the system starts rapidly and then slows down, whilst still not encountering a live-lock. This happens because message queues get filled with many messages, and thus, ballot values that get proposed, actually get read much later.

```
10 REQUESTS x 5000 CLIENTS - 1 ms SLEEP

   debug_level: 0,   # debug level 0
   docker_delay: 1_000,  # time (ms) to wait for containers to start up
   max_requests: 10,  # max requests each client will make
   client_sleep: 1,  # time (ms) to sleep before sending new request
   client_stop: 60_000,  # time (ms) to stop sending further requests
   client_send: :round_robin, # :round_robin, :quorum or :broadcast
   n_accounts: 100,  # number of active bank accounts
   max_amount: 1_000,  # max amount moved between accounts
   print_after: 5_000,  # print transaction log summary every print_after
      red↪ msecs
   crash_server: %{},
   window: 10, # when window = 1, it is equivelent to not having a window
   delay: 1000000 # random delay from 0 to 'delay' when the leader is
      red↪ preemted (0 means no delay)
   ---------------------------------------------------------------------
   time = 30000 updates done = [{1, 28628}, {2, 28547}, {3, 29759}]
   time = 30000 requests seen = [{1, 14025}, {2, 13965}, {3, 19962}]
```

```
time = 30000 total seen = 47952 current lag = 19405, max lag = 19405
time = 30000 scouts = [{1, 0}, {2, 0}, {3, 0}]
time = 30000 commanders = [{1, 0}, {2, 0}, {3, 0}]

time = 60000 updates done = [{1, 41561}, {2, 41557}, {3, 41559}]
time = 60000 requests seen = [{1, 15000}, {2, 15000}, {3, 20000}]
time = 60000 total seen = 50000 current lag = 8443, max lag = 19405
time = 60000 scouts = [{1, 0}, {2, 0}, {3, 0}]
time = 60000 commanders = [{1, 0}, {2, 0}, {3, 0}]

time = 90000 updates done = [{1, 50000}, {2, 50000}, {3, 50000}]
time = 90000 requests seen = [{1, 15000}, {2, 15000}, {3, 20000}]
time = 90000 total seen = 50000 current lag = 0, max lag = 19405
time = 90000 scouts = [{1, 0}, {2, 0}, {3, 0}]
time = 90000 commanders = [{1, 0}, {2, 0}, {3, 0}]
```

Appendices

A Test Results

For the full local test results see results_local.txt file and for the docker results see results_docker.txt file in the root directory of the sumbission.