60009 Distributed Algorithms
Multi-Paxos Coursework Report

Jin Xian Yap (jxy18), Emily Haw (eh4418)

18th February 2021

1 Overview

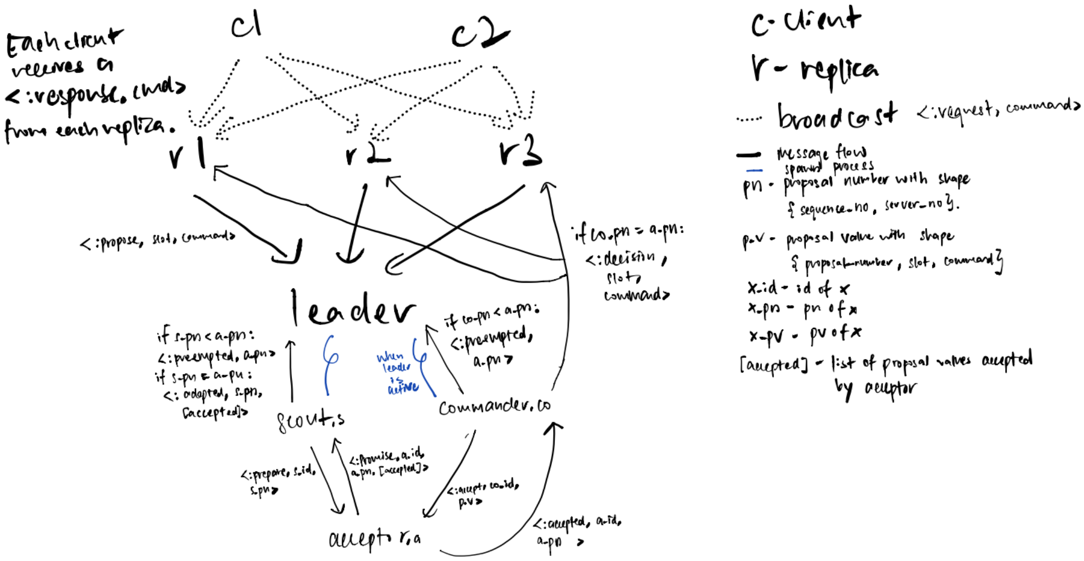

Figure 1: Multi-Paxos system structure and communications between modules

2 Design and Implementation

Replica

Each replica maintains a sequence of slots that will be filled in with commands issued by clients. Upon receiving a {`:CLIENT_REQUEST, command`} message, the replica assigns the command to the lowest-numbered slot - `slot_in` and increments the `slot_in` counter, and sends a {`:PROPOSE, slot, command`} message to all leaders.

A replica also receives {`:DECISION, slot, command`} messages, which is sent by a commander, where a majority of acceptors has accepted that `command` is assigned to `slot`. This decision is added to a map of decisions with key `slot` to value `command`.

The replica then iterates through the decisions map, starting from the current `slot_out`. This ensures that if there is a decision for a previously executed slot, it will be ignored since `slot_out` is incremented every time a decision is executed. The replica checks if for `slot_out` there is a proposal in the proposals map. If so, it is removed from the proposals map. Take `command_proposed` to be the corresponding value to `slot_out` in the proposals map. If the `command` in decisions is not equals to `command_proposed`, this means that this replica's proposal was preempted, so `command_proposed` is added back to the set of requests to be re-proposed later.

The replica checks again that a `command` has not been executed before, by checking for similar commands with lower slot number than `slot_out`, then performs the `command`, by sending its database {`:EXECUTE, transactions`} and notifying the client with {`:CLIENT_REPLY, cid, true`}, where `command` = {`client_process_id, cid, transactions`}.

Acceptor

Each acceptor receives two types of messages, `:PREPARE` and `:ACCEPT`, and keeps track of the highest proposal number it has seen, and the list of proposal values it has accepted internally. `:PREPARE` is sent from Scouts while `:ACCEPT` is sent from Commanders. Each `:PREPARE` message received contains a proposal number `pn`, which it then compares to its current proposal number. It adopts `pn` if it is higher. It also sends a `:PROMISE` message that contains its proposal number, and the values it has accepted back to the scout. Each `:ACCEPT` message received contains a proposal value of the form {`proposal_number, slot, command`}. If `pn` is equivalent to its current proposal number, this value is accepted and appended to the list of accepted values. It also sends each commander an `:ACCEPTED` message which contains its current proposal number, which is the highest it has seen so far.

Scout

A scout is spawned by its leader to try to get the leader's proposal number accepted by a majority of acceptors. It sends {`:PREPARE, self(), proposal_number`} to every acceptor, then waits for promises from acceptors. The scout receives {`:PROMISE, acceptor_id, pn_returned, p_accepted`}, where `pn_returned` is the highest proposal number seen by that acceptor, and `p_accepted` is the set of all proposal values it has accepted before. If `pn_returned` is equivalent to `proposal_number`, then `acceptor_id` is removed from a `waitfor` list, which is initialized to include all acceptors, and adds `p_accepted` to its list of proposal values `p_values`. If the number of acceptors in the `waitfor` list is less than half the total, then the scout sends to its leader {`:ADOPTED, proposal_number, p_values` then terminates itself. If the scout receives a `:PROMISE` where `pn_returned` is not equivalent to `proposal_number`, this means it has been preempted, and the scout stops waiting, sends the leader {`:PREEMPTED, pn_returned`} and terminates.

Commander

A commander is spawned by the leader if the leader is currently in active mode. Each Commander sends its acceptors a {`:ACCEPT, self_id, proposal_value`} message and waits for each acceptor to reply with {`:ACCEPTED, acceptor_id, proposal_number`}. It checks if any acceptor it receives a reply from has adopted a proposal number that is higher than the proposal number of its leader. If this is the case, it sends a `:PREEMPTED` message to the leader which contains this higher proposal number. If not, it waits until it has received replies from a majority of its acceptors, before sending a `:DECISION` message to all the replicas to fix `command` at the slot `s`, where `proposal_value` = {`proposal_number, s, command`}.

Leader

Each leader receives a {`:PROPOSE, slot, command`} message from replicas that includes the command proposed for a particular slot number. Internally, it also keeps track of proposals, which is a map of slot numbers to commands proposed for that slot, and its current proposal number, which is a tuple with its current sequence number and its server number.

Each leader functions in two different modes, active and passive. In passive mode, it has received a `:PREEMPTED` message, indicating that at least one of its acceptors has adopted a proposal number that is higher than its current proposal number, p. This implies that there is a possibility that it would not be able to get a majority of acceptors to adopt or accept its proposals. It would then increase its current proposal number by increasing its current sequence number. After some time, it repeats its attempts to request an acceptor to adopt its proposal number.

When it receives an `:ADOPTED` message, this means that a majority of acceptors are now able to accept its proposals, and the leader now switches to an active mode. For each slot, it gets the command with the highest proposal number and merges this with its current map of proposals. It spawns Commanders for each of these proposals, enabling decisions to be made for each slot.

A leader also updates its proposals map when it receives a `:PROPOSE` message from replicas proposing a command for a particular slot. If the slot has not been previously added to proposals, we add the slot and the command to it. We also spawn Commanders for any new proposals added if the leader is currently active.

3 Debugging and Testing

To ensure that nodes and processes were being spawned correctly, we made use of the existing Monitor component that would output a notification whenever a process (such as a Replica, Leader) has been spawned. Aside from compiler output due to syntactic errors, we were able to track logical errors in our code primarily by tracking the life-cycle of a request, starting from when it is first received by a replica, until it gets decided and is finally executed. Through this process, we were able to identify which component in the pipeline was causing errors.

During initial tests, we used a very simple configuration (using a client sleep time of 500ms) to first make sure that requests can be proposed and decided correctly without too much preempting of proposal numbers among servers. Once we have made sure of that, we set the client sleep time back to 2ms to examine the live-lock that would arise, and then took steps to alleviate the situation. It was particularly difficult to test for live-lock situations, because there would be a very large number of new scouts and commanders being spawned, such that it is impossible to analyse the output in a meaningful way.

Since the system kept crashing prematurely on our own laptops, we mainly ran our tests on DoC lab machines in Linux(Ubuntu), which has a Intel Core i7-6700 3.40GHz processor, 4 cores and 16GB RAM.

4 Evaluation

Without Live-lock Mitigation

Prior to introducing a timeout for when a Leader spawns a Scout with a new proposal number after being preempted, live-lock occurs very frequently in the system. In the simple case where client sleep time is 6ms and clients stop sending requests at 6000ms, the system becomes live-locked at just 126 requests out of a total of 293 requests. In a later evaluation below under the same configuration with the timeout implemented, the system successfully executes all requests received, indicating that the timeout is effective. (refer to `outputs/livelocked.txt`)

Solving Live-lock

To mitigate the effects of a live-lock, we have introduced a timeout before a leader spawns a new Scout with a new proposal number. This timeout is set as the difference between the adopted proposal number and its current proposal number multiplied by 2ms, and is capped at 1000ms, so that it does not become too long. This would allow the other leader with the current accepted proposal number some time to achieve a decision for some request, instead of being itself preempted again. Further evaluations on the system were done after this timeout has been implemented.

Varying Request Frequency

We primarily varied the `client_sleep`, `client_send` and `client_stop` configurations to evaluate our system under different levels of load and frequency of requests. `client_sleep` is the amount of time a client waits before sending new requests, `client_send` is the number of servers that a client sends a request to, and `client_stop` is the time at which a client stops sending further requests.

- **client_sleep**
  We varied the `client_sleep` configuration to measure how well our system performed under load, while keeping `client_send` as broadcast and `client_stop` as 60000ms (which means the clients never stop during our evaluation duration of 15000ms). A smaller `client_sleep` value implies that the clients wait a shorter amount of time before sending a new request, thus corresponding to more requests being sent by the clients during the evaluation period. Our system was able to perform relatively better when `client_sleep` was 100ms (refer to `outputs/client_normal_sleep_100.txt`) as compared to when it was 2ms (refer to `outputs/client_under_load_sleep_2.txt`). At the end of 15000ms, the percentage of requests completed for the former was 17.76%, with 136 requests completed and 766 requests seen compared to that of the latter which was at 13.4%, with 3088 completed and 23052 seen requests . This was to be expected since the higher load would lead to higher probabilities of a live-lock occurring due to the large amount of requests, and hence proposals being sent to acceptors.

- **client_stop**
  Next, `client_stop` was varied to observe how our system performed when the clients stop sending requests after 6000ms (refer to `outputs/client_stop_after_6s.txt`), as compared to when the clients continuously sent requests for the full 15000ms.
  (refer to `outputs/client_under_load_sleep_2.txt`)
  If a live-lock was encountered then later resolved, the system managed to catch-up by completing the remaining requests that were undecided, thus reducing max lag, given the clients stops sending further requests after 6000ms. However, in the version of the output that we have included, there was some form of live-lock in place that prevented the system from doing so.

- **client_send**
  There were three different variables, `:broadcast`, `:round_robin` and `:quorum` that were varied for this field. `:round_robin` resulted in a higher percentage of requests completed as compared to `:broadcast`. 75% of requests seen were executed in `:round_robin` as compared to 20% of requests in `:broadcast`. This was because at each round, only one server, instead of all servers, receives a client request, reducing the probability of the system live-locking since servers do not waste time proposing repeated commands for the same request which may have already been executed.
  (refer to `outputs/broadcast_sleep_100_stop_6000.txt`,
  `outputs/round_robin_sleep_100_stop_6000.txt`,
  `outputs/quorum_sleep_100_stop_6000.txt`)

Server Crashes

We evaluate our multi-Paxos implementation on a range of scenarios involving crashing servers with a simplified configuration, namely a client sleep time of 100ms and clients stop sending requests after 6000ms. This is to ensure that the system behaviour can be easily observed without the interference of other factors such as live-lock. There are 5 servers and 5 clients.

- **1 server crash**
  Server 1 is set to crash after 500ms. During the first 500ms, the system successfully executes 9 database updates. However, the system is still able to execute further updates after server 1 has crashed, since majority (3 servers) can still be easily achieved out of the 4 remaining servers, successfully executing 227 out of 294 requests at the end.
  (refer to `outputs/1_server_crash.txt`)

- **2 servers crash**
  In this scenario, server 1 crashes after 500ms, and server 2 crashes after 5000ms. During the first 500ms, the system successfully executes 5 database updates. Like before, the system can still proceed to execute further requests, reaching 240 database updates at the time server 2 crashes. The system is able to continue and finish executing the remaining requests to complete all 295 requests sent from the clients. This indicates that the system is capable of tolerating up to f failures for $2f + 1$ servers. In a later test (not included in the output folder) where the system is given a heavier request load by having clients issue requests every 2ms, the three remaining requests appear to be live-locked indefinitely after the second server crashes. This is likely because it becomes more difficult to achieve a majority acceptance among the remaining acceptors, as all three of them must agree on the same proposal number.
  (refer to `outputs/2_servers_crash.txt`)

- **3 servers crash**
  Server 1 crashes after 500ms, and server 2 and server 3 both crash at 5000ms. As before, the system is able to achieve consensus and execute requests after server 1 crashes. However, after server 2 and 3 both crash, the system is unable to execute further requests. This is because the scouts and commanders spawned will never receive replies from 3 out of 5 servers, so there will not be a majority, hence this system cannot tolerate more than f failures for $2f + 1$ servers.
  (refer to `outputs/3_servers_crash.txt`)

2 Servers

In the case of 2 servers, gaining majority acceptance means both acceptors in the system must agree. Our evaluations show that a 2-server system is still capable of achieving consensus and executing requests when the `client_sleep` time is set to 100ms.
(refer to `outputs/2_servers_5_clients_easy.txt`)

When the `client_sleep` time is set to 2ms, the system is also able to progress, does not live-lock, and at the end of 15000ms has executed more requests than when using 5 servers in the system. (refer to `outputs/2_servers_5_clients_hard.txt`) This is because with just 2 servers, there is much less contention for proposal numbers, so they can be accepted easily, and requests decided quickly.

Clearly, though, this system is not fault-tolerant at all. When one of the two servers crash, there will never be consensus, hence the system fails and will never execute subsequent requests received. (refer to `outputs/2_servers_5_clients_1_crash.txt`)