DAD Project - Boney, Paxos, Two-Phase Commit and Perfect Channels

David Belchior - 95550
davidbelchior@tecnico.ulisboa.pt

João Aragonez - 95606
joao.aragonez@tecnico.ulisboa.pt

Vasco Correia - 94188
vasco.cardoso.correia@tecnico.ulisboa.pt

Instituto Superior Técnico - MEIC-A

1. Introduction

This project consists of a distributed system composed by a set of bank servers, which cooperate using a primary-backup replication protocol.

To achieve this while ensuring coherence in the choice of the primary server for each given time slot, this set relies on a Boney service, composed by a set of servers which run Paxos, a partially fault-tolerant consensus algorithm [1].

Additionally, to ensure a fault-tolerant message propagation to backup servers, each slot's leader runs a two-phase commit algorithm.

Finally, to guarantee an in-order delivery of the messages across all communication layers, an implementation of perfect channels was required.

2. Boney

Boney is an adapted version of Chubby, a coordination service reliant on Paxos to achieve a consensus in a concurrent system and, when used multiple times, ensure total order.

For our project, Boney was used with the goal of deciding a primary bank for each slot and, as such, avoid having more than one self-claimed bank leader running commands, reducing additional complexity regarding the two-phase commit algorithm (which will be described later in the report).

2.1. Time slots

Time slots are used in the Boney servers as a simplified way to simulate machines perceiving others as dead (in these case, frozen, i.e., unable to process any receiving messages) without needing to create network and/or machine failures.

2.2. Paxos algorithm

Paxos, described by Leslie Lamport in [1], is a consensus algorithm based on majorities to make progress.

In our approach, each Boney server is both a proposer, an acceptor and a learner, containing a separate service for each to provide modularity. If one were to create a partial instance of a Boney server (i.e., with only one or two of the aforementioned roles), they would only need to specify a different configuration for each role to all servers.

For a more in-depth description of the algorithm, we recommend reading the original paper or a different approach to it [2] for a better understanding.

2.2.1 "Prepare" and "Promise"

If a machine is alive and no other machine with a lower ID than its is perceived alive during a given time slot, such machine will wait until the first Bank client requests a CompareAndSwap operation. This is ensured by a dictionary that keeps track of the first value proposed for each slot (slotsHistory), whose values are set in BoneyProposerService.cs and processed in the Paxos function of Boney.cs.

Once the first request for a slot is received, the proposer will send a "Prepare" message, whose answer will depend on the request's timestamp (initially, the Boney's server ID) and the acceptor's most recent timestamp. If the former is higher or equal to the latter (meaning it's "more recent"), the request is accepted, the read_timestamp is updated, and a reply is sent with a "Promise" message containing the acceptor's current value ("bottom" if it has no value yet, which we defined as -1 to be lower than any real value, allowing it to be discarded) and when it was written (a write_timestamp); otherwise, a "NACK" message will be sent (write_timestamp is sent as -1 to identify the message type).

If the proposer received a majority of "Promise" messages without receiving a "NACK", it will start sending

"Accept" messages, which we'll get more in detail below; else, it will stop the current iteration of the algorithm, increment its timestamp by the number of known Boney servers, and start a new iteration.

2.2.2 "Accept" and "Accepted"

The "Accept" message is comprised of its timestamp and the value to request acceptance. Notice that both the "Propose" and the "Accept" messages include the corresponding slot for a correct access to the slot's information by the acceptors.

The acceptor then has a similar job to when it receives a "Prepare" message: if the proposer's timestamp is higher or equal to its own timestamp, it's marked as "Accepted", i.e., the response's status is marked as "true", the slot's information is updated and the "Accepted" message is propagated to the learners; otherwise, its status is marked as "false" ("NACK") and the proposer will increment the timestamp just like in the first stage.

Each learner, once it receives a message with a newer timestamp than the one it currently has, restarts a counter, which is incremented each time another message with that timestamp is received. Once it reaches a majority of "Accepted" messages with the same timestamp, the result of the CompareAndSwap operation will be sent to all Boney servers, and the chosen leader will begin accepting Two Phase Commit messages.

2.2.3 Potential problems

In the next paragraphs, we'll describe some possible situations that may happen during the execution of the algorithm but don't risk correction.

Even if two leaders attempt to propose their own value and they're both successful (for example, in a situation where a server with timestamp 1 achieves a majority of "Promise" messages and, before it's able to send "Accept" messages, a server with timestamp 2 also achieves a majority of "Promise" messages), the first server will necessarily receive at least one "NACK" reply from the "Accept" messages, since the second server replaced the initial read_timestamp with its own, forcing it to stop.

Another possible situation involves a second leader attempting to run an instance of the algorithm for a given slot after consensus was already reached. Since it will necessarily find the committed value, even if its "Accept" messages are accepted by a majority, causing the learners to send a new CompareAndSwap result message, the result's value is exactly the same as the first one, so, although redundant, the second server's run is still correct.

2.2.4 (Potential) optimisations

If a bank server issues a CompareAndSwap request for a slot that has already been decided, the proposer module can reply immediately with the decided value, thanks to a dictionary which keeps each slot's CompareAndSwap result (-1 until consensus hasn't been reached), not needing to start a new instance of Paxos.

A potential optimisation would involve using the same dictionary to avoid running into situations like the second one in the last section, allowing a proposer to stop running the Paxos instance once it finds out consensus has already been reached.

3. Bank

The bank is a service, running over a Primary-Backup service to guarantee consistency, that is responsible for keeping the state of the clients account and also for replying directly to this clients commands.

The run-time of the bank is divided into predefined time slots that will be used to determine the primary bank(with the help of Boney), as well as if a server is (not)suspected and/or (not)frozen.

3.1. Two Phase Commit

As each client can send commands concurrently, the bank implements the two phase commit protocol to guarantee that each of the received client commands is assigned with a unique sequence number and that this sequence is known through out all the bank servers.

3.1.1 Primary

The bank's primary server (only 1 per slot, due to the CompareAndSwap operation) is the responsible for assigning the unique sequence numbers to each command received and to propagate that command with the chosen sequence number to the backup servers.

When a command is received by the primary, it waits until no other command is being processed (using locks). In other words, all commands are processed sequentially. Then, it starts by assigning a tentative seq to that command(seq = last_seq + 1), sending a tentative message to the all servers (except for itself, where the tentative is stored directly) and waiting for a majority of responses.

After it has the majority of responses, it sends a commit to all servers (except for itself again, executing the command in the account) and again waits for majority of responses to proceed to the next command.

3.1.2 Backup

The bank's backup servers' main purpose is to keep the service fault tolerant. All commands received by one of this servers are "rejected" since the primary is the responsible for processing them.

When a tentative message is received, it is only accepted if the sending server has been the leader from the slot it sent the message until the current slot (this includes, naturally, when the leading server sent the message in the current slot).

Also, if there is a tentative for a given sequence and a server receives another tentative for the same sequence this message is only accepted if the slot contained in it is newer then the slot in the stored one.

Once a commit message is received, if the server which received it hasn't received a previous commit message for it,the commit will happen in this stage.

3.1.3 Clean Up

When a slot transition occurs, the primary bank server can change and it needs to assure that the job of the previous primary is finished and that it has all the commands that were previously committed. That is the Clean Up routine's purpose.

The routine starts by sending a ListPendingRequests message to all servers(except for itself, where it can do it directly) with the purpose of obtaining each servers received tentative commands that aren't committed in the primary.

With a majority of responses (including the sending server's value), the server orders the commands by their original sequence number in an ascending order, then, in case of a draw, in a descending order relative to the original slot (i.e., prioritising the most recent slot).

For all the filtered commands, the new primary is going to run a new instance of the Two Phase Commit protocol, using the same sequence number assigned to this commands.

Once all commands are finished, the new primary can start processing the new commands received as mentioned previously.

4. Client

Each client of the bank service tries to execute commands, broadcasting the command message to all bank servers. The replies received from the backup servers are ignored and the one received from the primary will determine if the command execution was successful, showing the obtained result, or unsuccessful, prompting an error to the user to retry the command.

This last situation can happen due to cases such as: the primary is still being decided; the clean up routine hasn't finished or the clean up routine/the Two Phase Commit protocol couldn't obtain a majority of responses, due to the majority of servers being frozen.

5. Perfect Channels

To ensure each connection creates a perfect channel, we must combine a FIFO channel with a guaranteed message delivery.

For the FIFO channel, we implemented a "stop-and-wait"-like algorithm: each client sends messages with a SEQ value starting at 1, while each server keeps an ACK value starting at 0 (representing the last successfully ACK'd message). When a client's stored ACK in a server is equal to the message's SEQ + 1, it's accepted and the ACK is incremented. In both cases, the ACK is sent within each message's reply, allowing the client to check if ACK higher or equal than SEQ and, if the condition isn't verified, repeat the request.

Since gRPC only ensures an "at-most-once" semantic, an infinite loop with a timeout for each gRPC call is needed to make sure each message is delivered successfully. It's value is defined by a constant baseline plus a random value.

6. Puppet Master

This project can be automatically run using a configuration file and an instance of a Puppet Master, which recognises the operating system (OS) it's being run on, parses the configuration file, and launches the referred processes with their necessary arguments (host, port, and, in the case of the client, an additional configuration file with the commands it will run) using the appropriate executable file, depending on the OS.

References

[1] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2), May 1998.

[2] L. Lamport. Paxos made simple. November 2001.