Boney Bank Report

Diogo Nogueira 95558
Eduardo Claudino 95567
Tomás Pereira 95682

Abstract

The Boney Bank project consists of developing a simplified distributed banking system that keeps the state of a single bank account and performs a variety of operations such as deposits, withdrawals, and balance reads. The challenge of this project is to achieve fault-tolerance and consistency. Fault-tolerance can be achieved through the use of multiple replicas, though that creates a consistency problem, we must make sure all the replicas have the same state (though we tolerate temporary inconsistencies). Our work can be tested through a configuration file that simulates network failures.

1. Introduction

To solve the proposed problem, the execution time is divided into time slots that dictate the network's behavior. The network itself is divided into three types of processes: Bank Clients, which we refer to as Customers, Bank Servers and Boney Servers. Boney processes are in charge of running the Paxos consensus system. This is achieved by "separating" the Boney process in two: the façade of Boney receives *CompareAndSwap* service requests from the Banks, which are then passed to a **Paxos** object, which communicates to the other processes in order to achieve algorithmic consensus. Meanwhile, Banks provide the bank service itself used by Customers to interact with the account, stored in **BankStore**. The whole project is implemented using Google's gRPC services in Microsoft™'s C#.

To handle concurrency requisites, our project utilizes two main systems. Firstly, there's the **PuppetMaster** that ensures all processes are initialized knowing their respective process id and also know the agreed upon startup time. Here we can also define if the Customer processes are started with command line input ("cmd" flag), or with scripted input from the file "customer_script" (using the "script" flag). Then also, there's the **Config** class, which all main classes use, which parses all the configuration parameters and processes them for later use by other processes. By convention, an **X** on the time wall parameter in the config file means all processes will start 20 seconds after the PuppetMaster.

2. Network Simulation

As mentioned before, to simulate multiple network configurations and behaviors, the configuration file is accessible to all the project's processes. The **Timeslots** object dictates each process's behaviour in each time period, or slot, of the expected run-time. This structure is also accessible through the Config class.

2.1. Freezer

A freezer thread in each process simulates temporary faults in the network. That said, using the configuration file it is possible to schedule a process to freeze during a time slot. At the beginning of each time slot, this thread checks if its process is frozen in the current slot, and so, sets that corresponding value in the process' Perfect Channel.

2.2. Perfect Channel

The Perfect Channel implements a gRPC Interceptor, overriding the methods for *UnaryServerHandler* and *BlockingUnaryCall*, for outgoing and incoming messages, respectively. For outgoing messages, the Perfect Channel waits for a signal to resume gRPC calls. This process infinitely retries till the recipient is available. For incoming messages, if the Perfect Channel is frozen, the service throws a *RPCException* to notify that the server is *Unavailable*. This way, the Perfect Channel handles both the case of a frozen sender and the case of a frozen recipient. As a small detail, the Perfect Channel also ensures FIFO for outgoing messages by the recipient, this measure may improve performance if a large queue of messages is being sent to a frozen server because only the message at the top of the queue will be performing remote calls.

3. Boney

The service provided by the Boney servers allows the Banks to elect a leader for each time slot. For this purpose, the Boneys implement a Paxos service between each other. Paxos is utilized after a Compare and Swap request reaches a Boney server, so that it maybe reach consensus between a majority of leader election requests.

3.1. Compare and Swap

At the beginning of a time slot, each Bank broadcasts a Compare and Swap request in a new thread to all the Boney Servers. Upon receiving this request, a Boney process asks its Paxos internal Paxos service for the result of the consensus for that time slot. When a reply is received, the Compare and Swap service then finally answers the Bank.

3.2. Paxos

The implementation of the Paxos algorithm is based on *ManualResetEvents*. This C# synchronization structure can be *Set* or *Reset* to either let through or block threads respectively. In the setup phase, a Paxos instance creates two threads: a *MagicFailDetector* and a *Proposer*. The first thread uses the configuration file to decide the Paxos leader in each time slot, it then uses this information to either block or start the *Proposer* thread. This second thread runs the proposer routine of the Paxos algorithm. The activity of the *Proposer* is controlled, as mentioned before, by the *MagicFailDetector*, but also by another event: when a Compare and Swap request reaches a Paxos instance, from a Boney service, a proposed value is set and this unlocks the second gate controlling the *Proposer*.

After starting the *Proposer*, the thread that requested consensus is blocked and when consensus is reached, a Learner thread, created by gRPC for the *Commit* service, unblocks the requesting thread which in turn blocks the proposer and answers the Boney. In regards to the *MagicFailDetector*, it follows a simple protocol: a process never suspects itself and it elects the process with the smallest ID from a list of plausible (non-suspected) candidates, which includes itself.

4. Bank

The *Bank* layer is responsible for processing operation requests from Customers. To do this it implements a Primary-Backup replication protocol. To choose the primary process for each timeslot, the Banks use the Boney service, which then use the Paxos algorithm implementation.

4.1. Leader Election

The Primary-Backup protocol works in the following way: for each timeslot, a Primary process is chosen, and all the other processes will be a Backup. The Primary process is responsible for assigning a sequence number to each operation in queue, and comunicating that decision to the other processes, through a 2-Phase Commit. To elect the Primary server for a given timeslot, all the Bank processes send a request to the Boney layer to retrieve which one is the chosen. When sending a request, the Bank servers suggest a process they think would be a good option, following the following criteria:

1. The previous Primary, if it is not currently suspected to be frozen;

2. The Bank process with the lowest id which is not suspected.

After the Bank processes send the request, the Boney layer is responsible for reaching a consensus within itself (see . Paxos), and communicate the outcome to the requesting Bank servers.

4.2. Primary Backup

The Primary-Backup protocol involves two types of processes:

- Primary: responsible for assigning a sequence number to each request and inform all other Banks using 2-Phase Commit;

- Backup: responsible only for keeping an updated state of the system, executing requests commited by the Primary.

When a Primary server receives an operation request from a Customer, it first saves it, and then sends a *prepare()* request to all Banks. When receiving a *prepare()* request, the receiving process first checks if the sending process is really the Primary. If true, it acknowledges the request. After gathering a majority of replies, if no answer was negative, the Primary process will then send a *commit()* message to all Bank processes. This message will trigger all the processes to assign a sequence number to the operation and put it in the *commited* queue. This way, as soon as the operations are contiguous, each Bank process will start executing them. When an operation is executed, it is moved to the *executed* list, and a trigger is sent to signal that a response is available to be sent back to the Customer.

5. Structures

Most of the relevant structures used in this project can be found in the **Common** directory. Some can be process specific, such as a Bank *Operation*.

5.1. Operation

An *Operation* object comprises all relevant information about a Customer request. An *Operation* also has a trigger that is signaled when the operation is executed on the *BankStore*. This is useful for the requesting threads because this way they can just wait on that trigger, knowing they will the signaled when the result is available. When that happens, the result is sent back to the Customer.

5.2. Commit History

A simple *Learner* specific structure that keeps track of already completed rounds of consensus. Also establishes the correlation between generation and value given in said generation.

5.3. Server Info

Server Info encapsulates the gRPC channel and keeps track of the given address. Subclasses can be found for both the Boney Server Infos and the Bank Server Infos, in the required directories.

6. Conclusion

In this project, we implemented Paxos and 2-Phase Commit algorithms in order to tackle redundancy and consistency problems. These protocols work the best when each process involved has a good way to determine which processes are frozen or not.