Boney Bank: an approach to reliable fault-tolerant system using Paxos

Guilherme Carvalho, Hugo Escobar, João Sá
Prof. Luís Rodrigues, Prof. João Garcia
Instituto Superior Técnico
Desenvolvimento de Aplicações Distribuídas

Abstract

Boney is a 3-tier application that acts as a Bank service. It brings an introduction to the implementation of a Paxos communication system and Primary-Backup replication protocol. The 3 tiers are composed by clients, bank servers and boney servers. The clients make requests to the bank servers, and the bank servers communicate with the boney servers to know who is the primary server(who they need to follow). The processes in the system(bank+boney servers) are connected by perfect channels, that ensure that all messages in the system are not lost and are eventually delivered. To simulate server crashes, the application runs on a slot based timer. On slot change, a process may "freeze" to simulate a crash either on a bank server or on a boney server.

1. Introduction

The challenge taken on this project was to make a real implementation of the material given in the theory classes: a functional Primary-Backup replication protocol on top of a consensus achieving system utilizing Paxos protocol. The basic operations of the application are induced by the clients, the bank has multiple servers in which one is the primary (elected by the boney servers). All servers receive the requests from the client but the primary is the one assigning the order in which requests are executed.

Utilizing a "time" slots scheduler to force the change (or not) of primary servers the bank can replicate and continue functioning even with n/2 - 1 server failures. When a "time" slot finishes, the bank servers communicate with the Paxos protocol to know who is the new primary for that slot. To control the whole system, a "Puppet Master" was also incorporated, thus allowing an easier setup and testing. The first step was to get the Paxos protocol up and running, this included the 3 stages of communication intrinsic to it: proposers, acceptors and learners. The Primary-Backup servers operate through sequence numbers for the received requests to maintain consistency among all replicas.

The final objective is that the bank is a distributed system that can be used on different machines and still work, and as so, the whole application was developed in C# utilizing gRPC for the communications between processes.

2. Puppet Master

The sole purpose of the **Puppet Master** is to centralize the initialization of all the processes and save some manual work in testing and running the application. It reads the initializing configuration file(a file with all the needed information about the setup of the application: number of servers for Boney, Bank and Clients as well as the time slots information about the state of the servers), and runs all the needed processes to put the application functional. The puppet master code is located inside the "PuppetMaster" project in a file named "Program.cs". If the user needs to call a new process all he has to do it add a line to the main method: "processesList.Add(p.run(n, true));" where n is the processID. The boney will automatically check the type of process it needs to launch. The first extra feature that was added was terminal hidding when launching a process, this way we can test only a layer of the code without the other layer terminals popping out. For this all that is needed to do is add the "true" flag when calling p.run() method. The second and last extra feature is the possibility to shut down all processes instantly by sending a command "exit" on the puppet master terminal.

2.1. Configuration File

The Configuration file is a global file that can be accessed by all the processes and contains all the information to correctly simulate a possible situation on the bank application. It identifies the number of processes of each tier, the number of slots where the application will run, a time given to each slot, the time at which the simulation starts and the individual configuration/characteristics of each process for every time-slot. The individual configuration of each process includes: the identifier of the process, information about it being frozen or not for the corresponding time-slot and if the other processes of the same ”group” should see him as suspect of not(of failure).

3. Bank Client

Bank clients have no information on how the bank works and if they are or are not frozen. The available requests can either be a withdrawal, a deposit, or a balance consult. The request for a specified operation is sent to all known bank servers. The replies that come from the bank are then shown with the new balance. The requests are identified that specifies the client and the number of the request(¡clientID, operationID¿). All replies from the bank server are printed with information regarding if the server replying is the primary or secondary. The client process has two ways of parsing input. The first one is reading from a file that is given in the configuration file. If no file is given in the configuration file with commands the user is able to input anything they want.

3.1. Client Parsing

On our approach we opted to give all processes access to the configuration file instead of passing the information from the puppet master to be less ”heavy”. The clients parse 2 types of files. Firstly, the global configuration file is read to get information of the existing bank servers on the system so that further on they can be contacted and receive the requests. It then goes on a second parse of a new file that has a list of the commands that the client will reproduce. The 4 used identifiers are: D(for deposit), W(for withdrawal), R(for read balance) and S(to induce a wait operation and idle the client from requests).

3.2. Client Initialization/Running

Once a client is initialized by the puppet master, it is expected that the main system composed by the bank servers is already up and running with no problems. From then on, all known bank servers of the system(read in the parse file) are stored locally and a loop is started to read and reproduce the actions from the input file. Every request gets at least an answer unless the majority of the bank servers are ”crashed”(situation where we consider the bank application is down). After sending a request, the client awaits 1 answer before continuing to the next operation. Other replies are not needed to advance because of consistency assurance we have from the bank servers. There were some difficulties when parsing the client input because if there were too many requests at the same time, for example 300 requests, the server starts having handling problems and throws network exceptions.

4. Bank Server

Bank servers utilize 2 communication protocols since they are the only ones that communicate with 2 tiers of the application at the same time(Clients+Boney). The operation is based on Primary-Secondary servers that receive the client requests. Although all servers receive the requests, the primary server is the one that organizes and sequences the commands and then sends the correct order to the other servers. If a secondary server receives a request to execute a command from a client, it stores it and only executes it upon being told to do so by the primary for the current slot. The primary server is re-elected every new time-slot and every bank servers communicate with the Boney system to know who is the new primary. Bank Server Code is located in ”BankServer” project inside ”program.cs”.

4.1. Bank Parsing

All instances must store the addresses of all other bank instances and Boney instances to further communicate. Besides that, they also know for how long they are supposed to run, when the application starts, the size of the slots and the state of itself and a suspect state of the other servers for all slots. This code is writting inside the method ”parseConfigFile”.

4.2. Leader Election

Every time a a server advances a slot it can change his state to frozen(simulating a temporary crash) or normal(if previously frozen). As so, all communicate with the Boney servers to discover who is the new leader for that slot. The Boney tier is a service that runs a Paxos algorithm with the sole purpose of deciding who will be the next primary server for the asked time-slot. After the primary is elected, a cleanup command is executed to ensure that all the operations proposed by the previous primary server are committed and in the correct order.

The Leader will send a request for all bank servers for their list of received but not executed operations. Then it will start by committing those before continuing for the newer operation requests.

4.3. Server Replication

Because the bank clients send their requests to all the servers but only the primary can approve the operations, the replication of information between the primary and the backup servers follows a 2-stage process. At first the primary server does a tentative for a sequence number to all the backup servers that answer with an acknowledgment. In our case, it will almost always be an OK since we are implementing perfect channels. After receiving a majority of OK's(acknowledgments) the primary then proceeds to commit the sequence number connected to an operation to all other servers. The backup servers can only accept a commit if it comes from a primary server correspondent to the slot, hence the need for a cleanup(no commands stay in a not finished state). If it accepts the commit, it will execute the operaton in question and add it to the "executedOperations" list.

5. Boney Server

The Boney server exists to resolve a consensus: who will be the primary server each slot. To achieve so, it uses an implementation of a Paxos algorithm that requires three main structures to simulate: a proposer, an acceptor and learner, one for each boney instance/server. When a new time-slot is started, the bank servers need to discover who is the new leader, so they all propose to the boney servers the server with the lowest id that is not suspected to be frozen(using the suspect flags) and the first request to be accepted by a boney server is stored as the primary server for that slot. Although our problem is electing a líder, in theory anything could be consensed with our paxos implementation. The boneys also change slots so they can change state from normal to frozen. The main boney implementation is in "BoneyServer" project inside "program.cs" but all paxos algorithm is inside "paxos.cs". All the boney servers keep a list of previously consensed líders to answer other bank server requests.

5.1. Boney Parsing

Each instance opens communications to all other boney instances and bank servers and stores all the information given in the configuration file: number of slots, timer of slots and state of all servers on all slots. This code is writting inside the method "parseConfigFile".

5.2. Paxos Algorithm

Paxos is used to resolve the consensus problem on who is the primary server for each time slot. To achieve it, every server has a proposer, and acceptor and a learner. Utilizing the theoretical message protocol, when the first request from a bank server is received, the boney server that received it, makes a propose to the other boney servers and awaits a majority of promises from their acceptors. Once a quorum of promisses arrives, the accept is sent to the acceptors that reply with an acknowledge and at the same "teach" the learners with the new value. The learners save the history of all the primary servers for every slot to prevent recalculating requests from processes that get unfrozen from past slots. Since the code was written directly for boney server and the only consensus needed are the liders, we made an optimization to isolate requests of each slot. For instance, everything regarding slot 3 consensus is saved in postion 3 of an array while position 4 belongs to slot 4 information. This way, we can have a better paralelism in the event of multiple slots consensing at the same time.

6. Conclusion

There are some final remarks to be made regarding the primary-backup implementation. First in our case, the Tentative request/reply are not really needed since we implemented perfect channels we can be sure that all the servers have the same sequence numbers and received the same information. Second, the way this primary-backup algorithm works is very inefficient since all clients need to contact all servers for all write requests, a better implementation would be to write to one of them and then this one would replicate to all of them, this way we would not have all the servers loaded all the times.

The first problem we encountered was when a frozen server is unfrozen he will be bombarded by many requests and when there are a lot it starts to slow down, our guess is that it is the slow down mechanism from http since grpc uses HTTP/2.

We finished this project with the code working for all test we have made, even the harder ones for instance: when a bank server makes commit for an operation and changes slot before the commit arrives. Although we are confident that our implementation is working for most cases, we can never be sure we tested every scenario since this is an heavily threaded implementation and threads tend to carry many bugs and crashes that the programmer never expects.