Boney Bank

Guilherme Gonçalves, Mafalda Ferreira, Marina Gomes
Instituto Superior Técnico
Av. Rovisco Pais 1, 1049-001 Lisbon
Design and Implementation of Distributed Applications
Group 11

Abstract

Banking applications are commonly used to perform simple day-to-day tasks and often require reliable and available systems that guarantee the continuous correctness of their functionality. The scope of this project is to present a fault-tolerant banking application, Boney Bank, having a Bank built on top of transactional processing protocols and replication mechanisms supported by a coordination service, Boney.

1. Introduction

Large distributed systems operate under a heavy workload, processing and executing requests from a large number of users. Particularly, in bank applications, clients depend on a reliable and available system to perform everyday tasks. These systems must remain operational under machine failures and ensure data is accessible and recoverable at any time. In order to ensure the availability and durability of data, systems operate under multiple servers maintaining copies of the same data, and replication mechanisms are responsible for data propagation across multiple nodes. Therefore, if one node fails, the remaining nodes can still provide data access and execute banking operations. However, this approach has two main challenges: maintaining data consistency and ensuring total ordering and atomic execution of operations. Hence, replication protocols such as Primary-Backup alongside Two-Phase Commit support these properties, having one server responsible for leading the ordering, execution, and propagation of transactions.

In this paper, we present a distributed fault-tolerant bank application that makes use of both replication and transactional protocols previously mentioned. Boney Bank is a reliable distributed bank application that ensures data durability and availability under machine failures by storing data across multiple nodes using the Primary-Backup replication protocol. To ensure the correctness of the protocol, the Bank uses Boney as a coordination service running on top of successive Paxos consensus to elect the primary that will lead the replication protocol. The primary uses a Two-Phase Commit protocol to define a total order of requests and atomically execute them while preserving data consistency.

2. Architecture

Boney Bank is built under a three-tier system: Client, Bank, and Boney. Clients execute read, deposit and withdrawal operations through the Bank API, and the Bank replicates requests throughout available nodes, ordering and atomically executing them. Both Bank and Boney run over a sequence of time slots and, at the start of each time slot, the Bank contacts the Boney service to run an instance of Paxos consensus that elect the new primary for the current slot.

This section will cover several components of the Boney Bank application, as presented in Figure 1. Firstly, we will briefly discuss failure detection and communication mecha-

nisms between processes using perfect channels. Then, we will cover the architectural approach in the Boney and Bank systems.

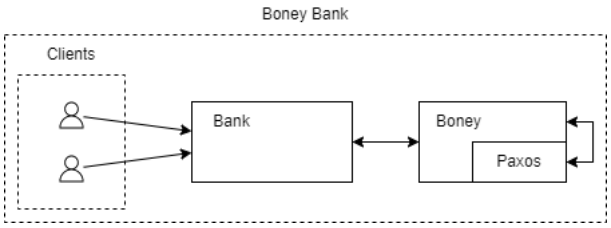

Figure 1. Overview of the Boney Bank architecture.

2.1. Failure Detection and Perfect Channels

Processes of Boney Bank rely on a failure detector that updates the view concerning the state of other nodes for each time slot. Processes guess whether or not other processes are suspected to be frozen ($suspected$ and $notsuspected$). Each process stores its state regarding being frozen or not ($frozen$, $notfrozen$), but the remaining ones can only suspect, incorrectly or not, about the state. Frozen processes represent an abstraction for unavailable nodes and, accordingly, must not execute or respond to new requests. In our solution, we force a frozen process to throw a gRPC exception with an unavailable status to signal other processes about its own frozen state.

Before moving to the architectural description of Boney Bank, we need to comprehend how the communication protocol takes place between the existing processes. Server processes communicate using perfect channels, ensuring ordered messages are never lost during the failure of a receiver, being eventually delivered as soon as it becomes available, i.e., no longer frozen. In Boney Bank, perfect channels are implemented using an interceptor for each communication channel. An interceptor is a gRPC component that interacts with the request and its context from incoming and outgoing messages. In particular, we intercept outgoing messages at the client side and queue them as soon as they arrive at the interceptor and before delivering them to the server. As soon as the message at the top is successfully transmitted to the server, it is then dequeued, and the call is returned to the client. After being dequeued, the following message can finally be transmitted. However, while the message at the head of the queue is still not successfully delivered, for example, due to a frozen server, the remaining messages must wait in the queue. Hence, we can establish a total order of messages.

2.2. Compare And Swap Requests

For primary backup to happen, the bank server's will need to elect a primary for each slot. For that reason, these servers will resort to the Boney service to act as their coordination service, through compare and swap requests and replies. For Boney's servers, everything starts with a a compare and swap request and the resulting values (i.e, bank leader for that slot) will be stored in a dictionary, where the key is the slot number and the value the bank leader id that corresponds consensus' output.

2.3. Synchronization Logic and Supporting Data structures

At first, the thread that received the request will queue the proposed leader value in the priority queue and block until a consensus has been reached for that slot. It is worth mentioning that, to ensure that only the first compare and swap request for each slot triggers the corresponding consensus, the bank leader for that slot will be automatically assigned with the value -1 (this way, a simple check if the dictionary contains the leader, prevents running multiple consensus for the same slot). The priority queue will be a standard c# priority queue, where the priority will be the slot value, thus allowing to execute each consensus in order and one after the other. Also, there will be a thread responsible for running the consensus that remain in the queue and will interact with the other threads in a producer-consumer logic (i.e, threads will put the consensus proposed values in the queue and one thread will remove the values from the queue). When a commit is performed, either as a leader or non-leader, the threads waiting to reply to a compare and swap request are notified of the occurrence.

2.4. Paxos

After placing a value in the queue, the thread responsible for running the consensus will be notified and will start handling the remaining consensus. First, that thread will check if its corresponding process is the leader and one of three things can happen if the process is not frozen for that slot, otherwise nothing will happen. If the process was the leader, it will start the Paxos algorithm. If it doesn't suspect that the leader is frozen, it will do nothing, since the leader eventually will start the Paxos algorithm. Finally, if it suspects the leader is frozen, it will increment the current leader until the value corresponds to a process that is not suspected of being frozen or the process' own id (case that it will start the Paxos algorithm as the leader).

As mentioned above, the Paxos algorithm starts when a process learns that it is the leader for that slot. However, before doing anything, the process will check if there was a leader change for that slot. If it did happen, the process will start with the prepare part of the Paxos algorithm (i.e, will ask the other servers for values for that slot, to check for previous commits), then will move on to proposing the value for that slot and finally committing the value. If no leader change happen, it will start with the propose and then commit the value. To ensure we don't wait endlessly for the replica's replies to prepare,propose and commit, we will only wait for a majority of replies. In another words, we just need to wait for n//2 (where n is the total number of servers and // represents integer division) replies, since we count the leader as one.

Since we only wait for a majority before proceeding and processes can become frozen (thus not replying until it stops being frozen), we can receive replies outside of its corresponding Paxos' stage(i.e, prepare, propose, commit) and/or consensus instance. For that reason, each reply will know the consensus instance (by sending the slot number in the message) and Paxos' stage, and the server will know at which stage and consensus instance it is. By calling a specific method for prepare (i.e, prepareHandler), we know that that reply corresponds to prepare. For both propose and commit, when we call the method that will handle the reply, one of its arguments will tell whether that reply corresponds to propose or reply.

2.5. Bank

Boney Bank provides the functionalities of a Bank application to the end users. Clients can perform multiple operations, such as reading balance and deposit and withdrawing a specific amount to and from the user's account. These functionalities are provided by the Bank's API:

```
balance = deposit (value)
(value, balance) = withdrawal (value)
balance = readBalance ()
```

Servers run over a successive set of slots, and at the beginning of each slot, servers vote for the most appropriate primary that they do not suspect to have been frozen. Using the coordination service, they perform a $compareAndSwap$ command to retrieve the leader assigned to that slot. The new primary will lead the primary backup protocol, further described.

2.6. Primary Backup and Two-Phase Commit

The bank service operates under multiple machines that store replicated data. The service uses primary-backup replication to ensure data durability and resilience to loss. Thus, if one node fails, data can still be recoverable and accessible through available nodes. In the Primary-Backup protocol, one node is selected as primary, responsible for assigning sequence numbers for each client's request and establishing an order of requests, propagating their changes to backups. Backups also receive requests from clients but are not allowed to decide on new sequence numbers. They must wait until the primary node sends an order to execute requests, which can be achieved through a Two-Phase Commit (2PC). This transactional protocol ensures atomic writes and data consistency. To perform this, replicas store requests according to their state during the protocol:

- $waitingCommands$ - list containing client requests waiting for 2PC.
- $tentativeCommands$ - dictionary with requests associated with a proposed number from the primary and waiting for a commit order.

- $commitedCommands$- dictionary concerning requests already committed with assigned sequence number.

Two-Phase Commit is accomplished within two phases: $tentative$ and $commit$. The prior concerns the proposal of sequence numbers by the primary and the latter regards the final commit of a request. In the primary server, one thread is responsible for executing the 2PC for each arriving client request and wait for following requests. Threads responding to clients can only queue the requests and wait for them to be processed in order before responding to the client.

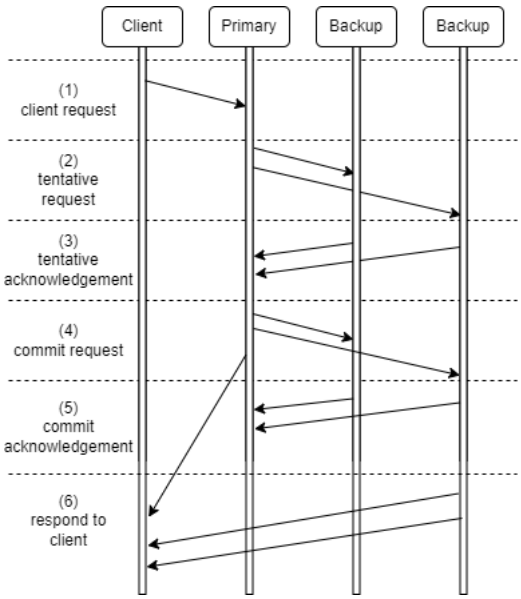

Figure 2. Execution example of a successful Two-Phase Commit protocol.

Figure 2 presents the protocol which is described as follows:

1. A client sends a request to all replicas.

2. As soon as the primary receives the request, it stores the request in $waitingCommands$ waiting to be processed. The thread performing the Two-Phase Commit protocol eventually retrieves the request from $waitingCommands$, stores it into $tentativeCommands$, and sends a $tentative$ to all backups in order to propose a sequence number.

3. Whenever backups have already ordered, committed, and executed previous requests and have already received the corresponding request from the client, having the request in $waitingCommands$, they may then move it from $waitingCommands$ to $tentativeCommands$ and respond to the primary with an acknowledgement message. Otherwise, they must block the call and wait for the client request.

4. Afterward, when a majority of backups have accepted the $tentative$ order, the primary then proceeds to commit the request, moving it to $commitedCommands$ and sending a $commit$ order to all nodes.

5. When backups receive the commit order, they move the request to $commitedCommands$ aswell with the associated sequence number and commit and execute the command. The primary does not wait for the majority of commit acknowledgements and continues with the Two-Phase Commit for newly available user requests.

6. Finally, after the request is stored in $commitedCommands$ and executed, processes respond to the client. Since the primary does not block until the majority of replicas has responded with the acknowledgement of the $commit$ order, it can respond to the client as soon as the $commit$ request is sent to the backups.

In both $tentative$ and $commit$ procedures, if backups receive a request from a primary not assigned to the current slot, they must reject the message and signal the leader to abort the protocol. However, if the request comes from a previous slot, but the sender is still the leader until the current slot, the request shall be accepted. Therefore, backups do not discard old but correct commit requests that can still be accepted and avoid forcing the leader to resend the exact requests for the new slot. This simple solution improves the Bank's efficiency and avoids flooding the system with unnecessary messages. It is important to note that a primary can propose new sequence numbers to commands that are not yet committed but were proposed by other previous leaders. If a backup receives a $tentativeRequest$, it must

adopt the new sequence number and remove the original associated one from $tentativeCommands$. This last step is crucial to avoid racking up the $tentativeCommands$ with invalid sequence numbers that would be later incorrectly committed by a new leader during the cleanup procedure, as we describe next.

2.7. Cleanup

Primary-Backup protocol supports the failure of primaries during the lifetime of the system. Once a primary fails, another replica takes over its function. In particular, in the Boney Bank system, slots define the start of periods when primaries might change due to the failure detector that identifies which servers are suspected to be frozen. Accordingly, the $compareAndSwap$ command may elect a new primary for a given slot. If new primaries present a stale state with missing committed commands and immediately proceed with 2PC, it will result in the incorrect assigning of sequence numbers. To solve this, we use a particular procedure, $cleanup$, that runs before a new primary moves into the usual 2PC protocol. The $cleanup$ procedure ensures that new primaries can always collect missing commands and reach a stable server version with every recent transaction execution.

Whenever a new primary distinct from the previous one is selected for the new slot, it runs the $cleanup$ procedure. Otherwise, if the primary remains the same, it simply continues its execution to perform Two-Phase Commit. The primary sends a $listPendingCommands$ message with the $lastKnownSequenceNumber$ to all backups. The $lastKnownSequenceNumber$ represents the last sequence number the primary knows to have been committed, and backups should only collect commands with a higher sequence number and, consequently, missing in the primary node. Backups then return a collection of pending commands, i.e., $tentativeCommands$ that were not yet committed by a previous leader and possible $commitedCommands$ that were not committed in the current primary server. The primary collects all these commands alongside its own $tentativeCommands$ waiting to be committed. Finally, after collecting all the requests, the primary stores them in a sorted list, ordering them according to their sequence number. This will force the original ordering of requests that might have been previously commited and avoid inconsistencies. Finally, the primary will commit all collected requests using the Two-Phase Commit protocol. Nevertheless, the new primary must wait to receive the request from the client and, consequently, having it stored in $waitingCommands$, before proposing and committing the collected command. After finishing the cleanup procedure, the primary may return to the program's normal execution, assigning new sequence numbers to new clients' operations.

3. Conclusion

The development of the Boney Bank applications eased the comprehension of how transactional distributed systems can preserve data consistency and availability throughout node failures. Furthermore, replication mechanisms such as the Primary-Backup protocol yields the need for consensus mechanisms to elect the primary. Thus, implementing the Boney coordination service was a crucial and necessary approach to building a reliable banking application.