APPLICATION FOR MANAGING CLIENTS IN A BANK

Giovanni Vuolo, Surbhi Sonkiya
(EIT Digital Master School)

1. Project Overview

This document presents a distributed banking application developed for CRUD (Create, Read, Update and Delete) operations. The application uses Zookeeper to have a coordination service between multiple servers and clients. The application is capable of providing following services through a simple text-based interface:

a) **Create**: Add a new client in the bank by providing account number, name of the client, and its balance.
b) **Read**: Read any of the existing client information.
c) **Update**: Modify the balance (deposit or withdraw) for an existing client.
d) **Delete**: Remove any of the existing client.
e) **State**: Retrieve the information of all clients in the bank.

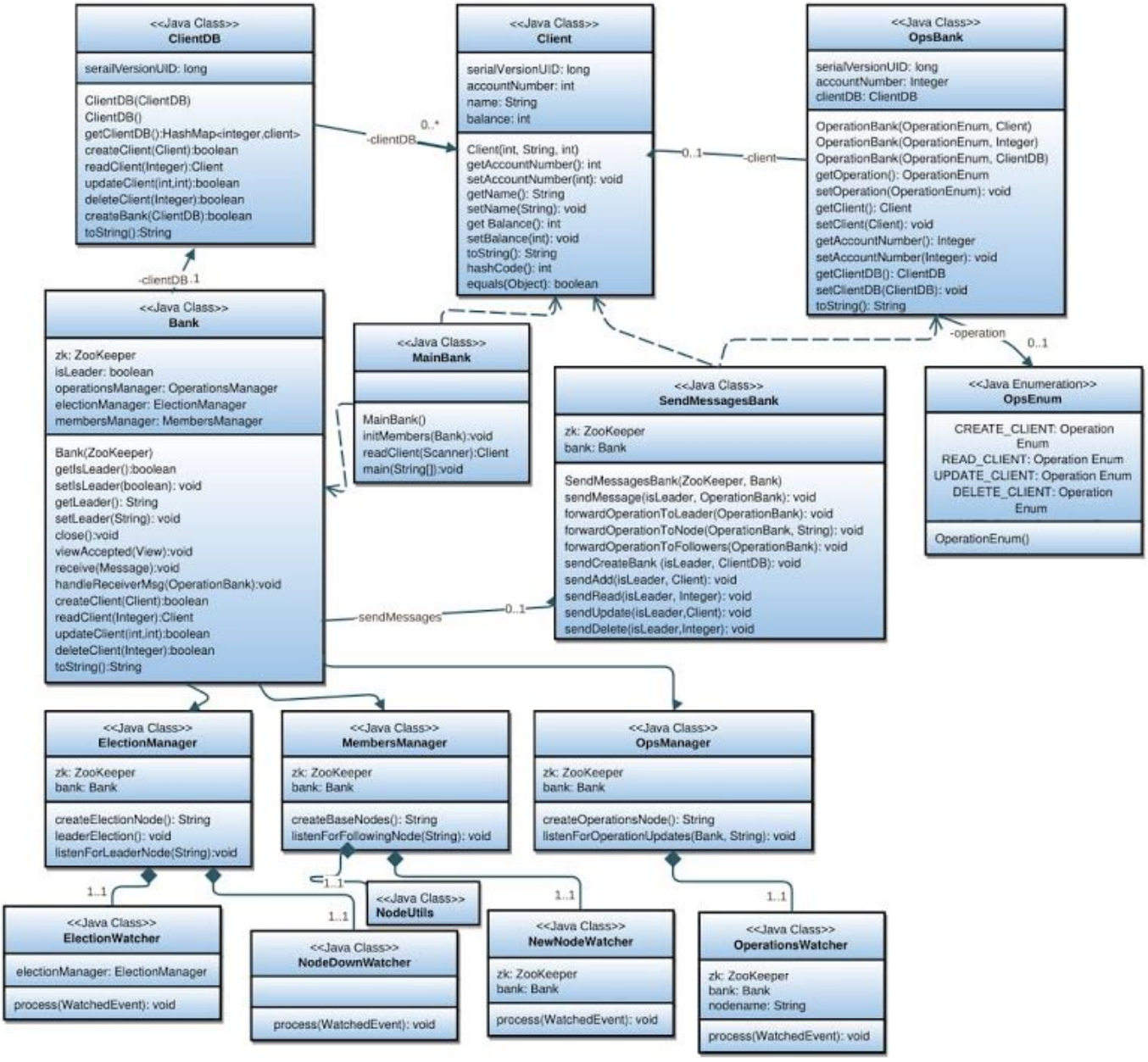
In the distributed environment, lack of coordination between systems is one of the crucial risk that needs to be taken care of. In the banking environment, if this is not watched, it might lead to financial losses for the bank and its client. Therefore, deployment of a coordination service was a must and hence, the zookeeper.

Zookeeper helped solve the major problems that an application face in a distributed environment such as fault tolerance, consistency, and availability. These are explained below.

a) **Fault Tolerance** - In order to make the system fault tolerant, it will be distributed in several virtual server. Therefore, if one fails another server will replace it and handle the requests. This will be handled by the Zookeeper ensemble.
b) **Consistency** - As far as it goes for the consistency, Zookeeper guarantees it will have it. In fact, the two guarantees of a Zookeeper system are Consistency and Partition Tolerance. In particular, the atomic broadcast and the leader election through quorum will assure a consistent view of the system. Finally every, write operation on the database will have to go through the leader, which will broadcast it to the follower.
c) **Availability** - For the system to be able to guarantee its availability at all the time, three different server nodes would be running at all times. So, the client can access the service at any time even with a slight delay due to the priority given to the consistency.

2. Update of the design of the system

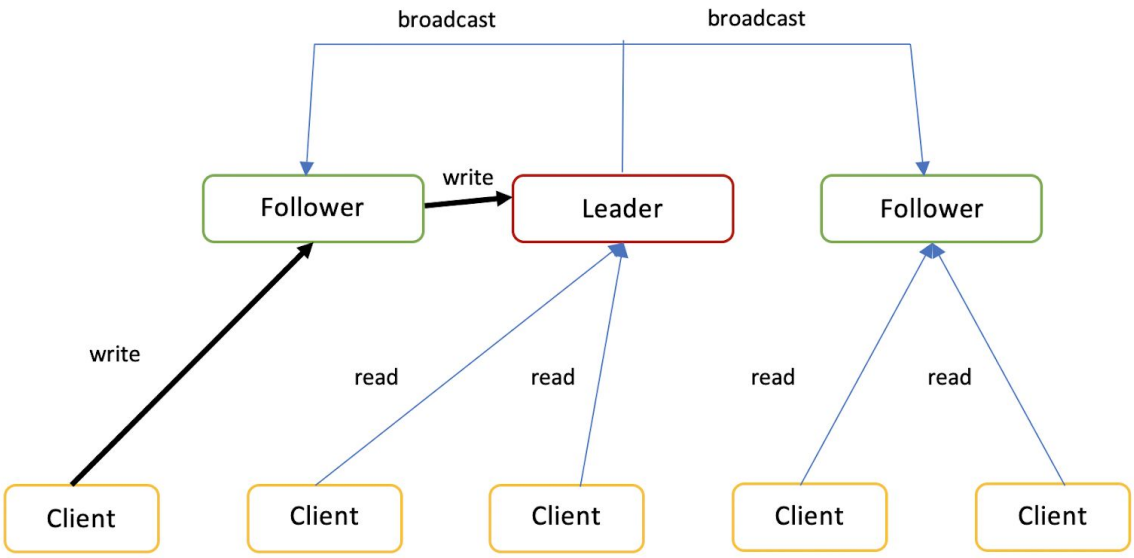
Class diagram shows the structure of the system. Below presented is the updated class diagram for this application. Zookeeper watchers are added in this diagram.


<<Java Class>>
ClientDB
serialVersionUID: long
ClientDB(ClientDB)
ClientDB()
getClientDB():HashMap<integer,client>
createClient(Client):boolean
readClient(Integer):Client
updateClient(int,int):boolean
deleteClient(Integer):boolean
createBank(ClientDB):boolean
toString():String
0..*
-clientDB
<<Java Class>>
Client
serialVersionUID: long
accountNumber: int
name: String
balance: int
Client(int, String, int)
getAccountNumber(): int
setAccountNumber(int): void
getName(): String
setName(String): void
get Balance(): int
setBalance(int): void
toString(): String
hashCode(): int
equals(Object): boolean
0..1
-client
<<Java Class>>
OpsBank
serialVersionUID: long
accountNumber: Integer
clientDB: ClientDB
OperationBank(OperationEnum, Client)
OperationBank(OperationEnum, Integer)
OperationBank(OperationEnum, ClientDB)
getOperation(): OperationEnum
setOperation(OperationEnum): void
getClient(): Client
setClient(Client): void
getAccountNumber(): Integer
setAccountNumber(Integer): void
getClientDB(): ClientDB
setClientDB(ClientDB): void
toString(): String
-clientDB
1
<<Java Class>>
Bank
zk: ZooKeeper
isLeader: boolean
operationsManager: OperationsManager
electionManager: ElectionManager
membersManager: MembersManager
Bank(ZooKeeper)
getIsLeader():boolean
setIsLeader(boolean): void
getLeader(): String
setLeader(String): void
close():void
viewAccepted(View):void
receive(Message):void
handleReceiverMsg(OperationBank):void
createClient(Client):boolean
readClient(Integer):Client
updateClient(int,int):boolean
deleteClient(Integer):boolean
toString():String
<<Java Class>>
MainBank
MainBank()
initMembers(Bank):void
readClient(Scanner):Client
main(String[]):void
<<Java Class>>
SendMessagesBank
zk: ZooKeeper
bank: Bank
SendMessagesBank(ZooKeeper, Bank)
sendMessage(isLeader, OperationBank): void
forwardOperationToLeader(OperationBank): void
forwardOperationToNode(OperationBank, String): void
forwardOperationToFollowers(OperationBank): void
sendCreateBank (isLeader, ClientDB): void
sendAdd(isLeader, Client): void
sendRead(isLeader, Integer): void
sendUpdate(isLeader,Client): void
sendDelete(isLeader,Integer): void
-operation
0..1
<<Java Enumeration>>
OpsEnum
CREATE_CLIENT: Operation Enum
READ_CLIENT: Operation Enum
UPDATE_CLIENT: Operation Enum
DELETE_CLIENT: Operation Enum
OperationEnum()
-sendMessages
0..1
<<Java Class>>
ElectionManager
zk: ZooKeeper
bank: Bank
createElectionNode(): String
leaderElection(): void
listenForLeaderNode(String):void
<<Java Class>>
MembersManager
zk: ZooKeeper
bank: Bank
createBaseNodes(): String
listenForFollowingNode(String): void
<<Java Class>>
OpsManager
zk: ZooKeeper
bank: Bank
createOperationsNode(): String
listenForOperationUpdates(Bank, String): void
1..1
1..1
1..1
1..1
1..1
<<Java Class>>
ElectionWatcher
electionManager: ElectionManager
process(WatchedEvent): void
<<Java Class>>
NodeDownWatcher
process(WatchedEvent): void
<<Java Class>>
NodeUtils
<<Java Class>>
NewNodeWatcher
zk: ZooKeeper
bank: Bank
process(WatchedEvent): void
<<Java Class>>
OperationsWatcher
zk: ZooKeeper
bank: Bank
nodename: String
process(WatchedEvent): void

3. Describe the system components, interactions and interfaces

System Architecture and Software Design:

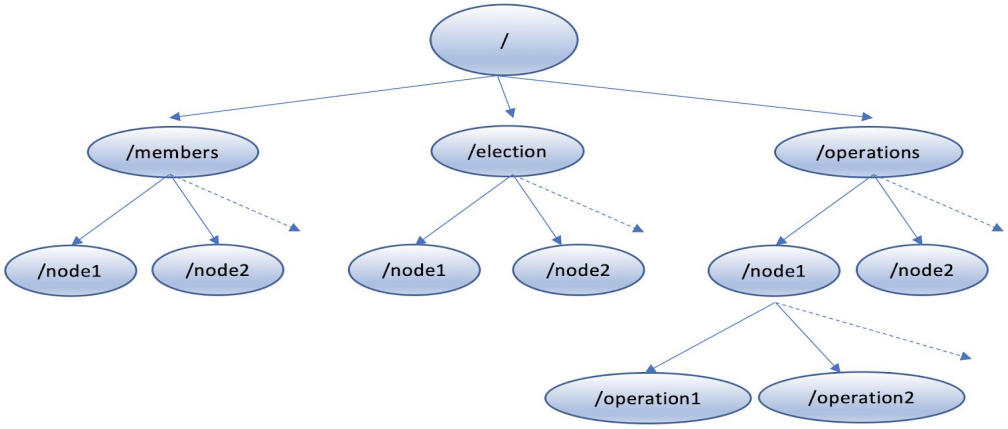
The system architecture diagram below shows how Zookeeper handles the requests from the clients. On top, we have three servers with one leader and two followers, as to guarantee consistency (leader handling the write requests), fault tolerance and availability (having three server running at the same time).



Data structure of the ZNodes of Zookeeper

This is a basic data structure of Zookeeper and shows which kind of nodes we will have in our application. Members for each process running. Election, which will handle the leader election, most probably

choosing the server with lowest id. Operation, which will contain all the information about the operation a client can execute on our system.



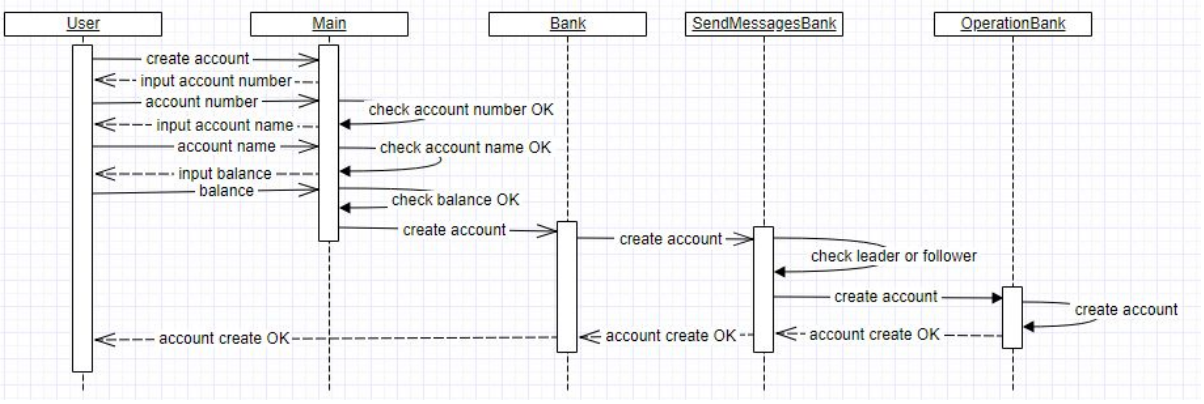
UML Diagrams

Below we present only Sequence diagrams because the updated Class diagram has already been presented in Section 2.

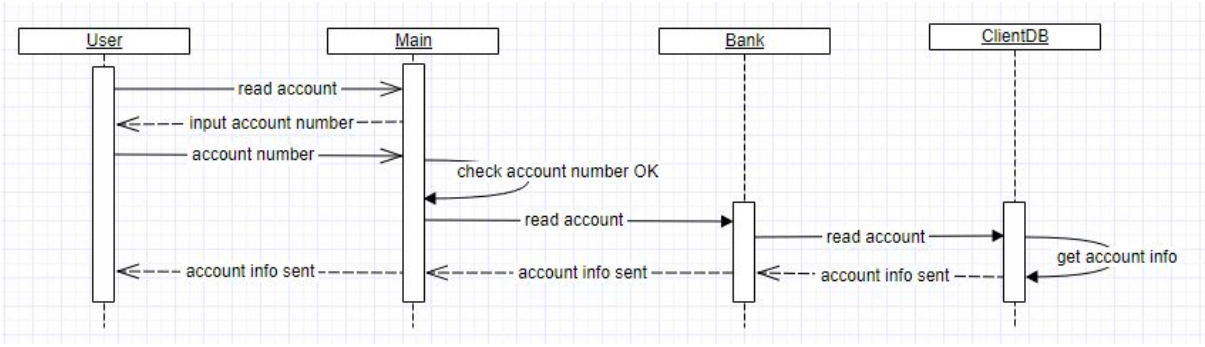
Sequence Diagrams

Sequence diagrams shows the interaction between various actors involved in CRUD operations of this application.

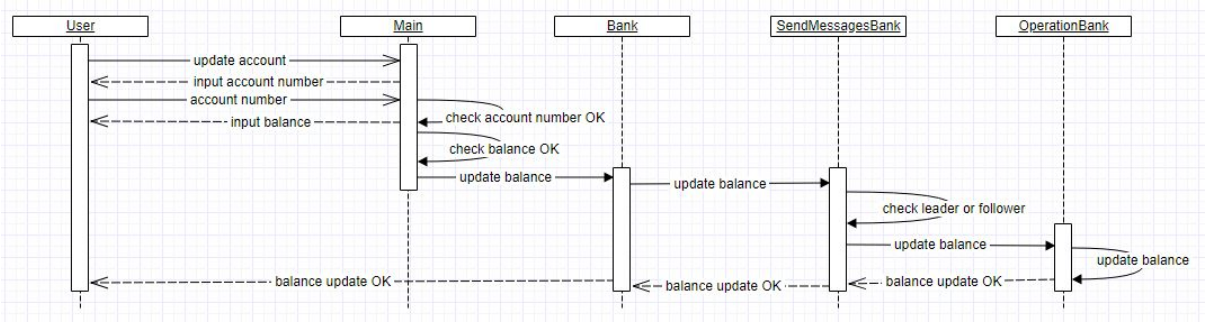
Create



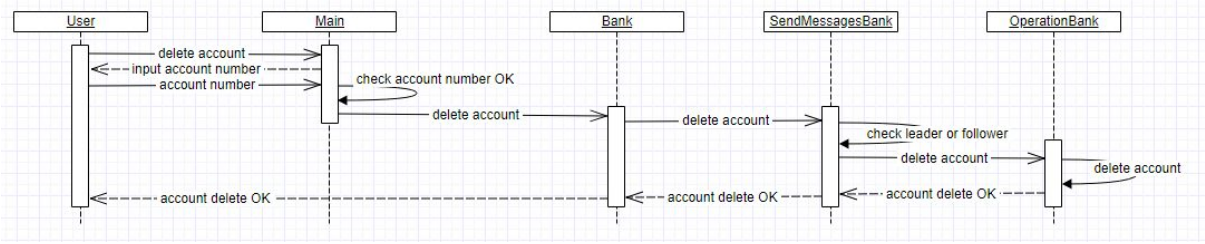
Read



Update



Delete



4. Description of the use of ZooKeeper for consistency

In Zookeeper, as observed through testing of the application, reads are faster than writes. It is obvious because reads operation retrieves client information from the server it is connected to, may it be a follower or a leader. However, if a write operation is received by a follower, then it sends that message to the leader and then leader sends write message

to all the followers. This takes additional time to complete the operation on all servers and ensure that data on all servers is consistent. Eventually, the client will see the same result on whichever server it connects to.

To ensure consistency in this application, Zookeeper watchers are implemented. They take care of broadcasting messages about leader election, node failure, operations, etc. Some of the properties are described below.

a. **Sequential consistency**: The updates are registered and executed in the same order as they were sent.
b. **Eventual consistency**: As explained by Werner Vogels, '*Eventual Consistency is a specific form of weak consistency; the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value*'. This explains exactly how Zookeeper behaves in reality. Due to network latency or failures, every node may not be updated at the same time, but eventually every operation will be executed on every node in the order it was received. If it is important that all the clients read the same value at any given instant and there exist no latency, then a sync() method must be called from Zookeeper API before every operation within each node.
c. **Atomicity**: No possibilities of partial updates, either an update completes or fails.
d. **Timeliness**: Within a defined threshold of time, client will be able to see the view or receive an outage error.
e. **Reliability**: A successful update operation cannot be rolled back unless and until an update operation is launched again to overwrite the previous update. However, this has two consequences:
    i. It cannot account for some failures like communication errors, timeouts, where client would never receive a message that the

update has been failed. Though, Zookeeper inherently tries to minimize these kind of errors as much as possible.

ii. While recovering from a server failure, meanwhile if a client happens to see the update either via executing a read or an update operation, then those updates cannot be rolled back.

5. Testing procedure

a. Modify ZooKeeper config file in all three servers and update the myid file in the data directory.
b. Start three Zookeeper servers.
c. Run the MainBank class as java application in three different machines.
d. Input operation number from client and see results.
    i. 1) Create a client
    ii. 2) Read a client's balance
    iii. 3) Update a client's balance
    iv. 4) Delete a client
    v. 5) Show bank DB

Below are the screenshots of all the operations.

Leader Election:

Server 1 is elected as leader. When a new node joins the ensemble, leader forwards the operations (initial database) to the new node.

```
New node: /members/node-0000000012
Client: [1, Surbhi Sonkiya, 1000]
Foward to new node: /operations/node-0000000012
Client: [2, Giovanni Vuolo, 2500]
Foward to new node: /operations/node-0000000012
Client: [3, Raffaele Perini, 6500]
Foward to new node: /operations/node-0000000012
Client: [4, Zsolt Dargo, 4000]
Foward to new node: /operations/node-0000000012
```

Server 2 is displaying node Server 1 as the leader.

```
The process node-0000000011 is the leader
>>> Enter desired op:
1) Create -> to create a client.
2) Read -> to show balance of a client.
3) Update -> to update balance of a client.
4) Delete -> to delete a client from database.
5) BankDB -> to show list of client in bank's database.
6) Exit -> to exit application
```

Operation 1: Create Customer

String Check: Server 1 trying to create a new customer. The operation does not proceed because the user inputs integer value for name.

```
1
>>> Enter account number (int) = 5
>>> Enter name (String) = 3
The input must be a string
null
```

Server 1 creates new customer

```
1
>>> Enter account number (int) = 5
>>> Enter name (String) = Pippo
>>> Enter balance (int) = 3000
Processing... [Operation = CREATE_CLIENT, client=[5, Pippo, 3000]]
Broadcast operation to followers: [Operation = CREATE_CLIENT, client=[5, Pippo, 3000]]
Operation List (Watcher): [0000000000]
Processing... [Operation = CREATE_CLIENT, client=[5, Pippo, 3000]]
Broadcast operation to followers: [Operation = CREATE_CLIENT, client=[5, Pippo, 3000]]
```

Server 2: Reading the bank database (operation 5) after a new customer has been created by Server 1.

```
Operation List (Watcher): [0000000003, 0000000002, 0000000004]
5
[1, Surbhi Sonkiya, 1000]
[2, Giovanni Vuolo, 2500]
[3, Raffaele Perini, 6500]
[4, Zsolt Dargo, 4000]
[5, Pippo, 3000]
```

Operation 3: Update customer

Server 1: updates the customer

```
3
>>> Enter account number (int) = 5
>>> Enter balance (int) = 4000
Processing... [Operation = UPDATE_CLIENT, client=[5, Pippo, 4000]]
Broadcast operation to followers: [Operation = UPDATE_CLIENT, client=[5, Pippo, 4000]]
Operation List (Watcher): [0000000001, 0000000002]
Processing... [Operation = CREATE_CLIENT, client=[5, Pippo, 3000]]
Broadcast operation to followers: [Operation = CREATE_CLIENT, client=[5, Pippo, 3000]]
Processing... [Operation = UPDATE_CLIENT, client=[5, Pippo, 4000]]
Broadcast operation to followers: [Operation = UPDATE_CLIENT, client=[5, Pippo, 4000]]
```

Server 2: Reading the bank database (operation 5) after a customer has been updated by Server 1.

```
>>> Enter desired op:
1) Create -> to create a client.
2) Read -> to show balance of a client.
3) Update -> to update balance of a client.
4) Delete -> to delete a client from database.
5) BankDB -> to show list of client in bank's database.
6) Exit -> to exit application
Operation List (Watcher): [0000000005, 0000000006]
Operation List (Watcher): [0000000007]
Operation List (Watcher): [0000000008]
5
[1, Surbhi Sonkiya, 1000]
[2, Giovanni Vuolo, 2500]
[3, Raffaele Perini, 6500]
[4, Zsolt Dargo, 4000]
[5, Pippo, 4000]
```

Operation 4: Delete customer

Server 1 deleted the customer

```
4
>>> Enter account number (int) = 5
Processing... [Operation = DELETE_CLIENT
Broadcast operation to followers: [Operation = DELETE_CLIENT
Operation List (Watcher): [0000000003, 0000000005, 0000000004]
Processing... [Operation = CREATE_CLIENT, client=[5, Pippo, 3000]]
Broadcast operation to followers: [Operation = CREATE_CLIENT, client=[5, Pippo, 3000]]
Processing... [Operation = DELETE_CLIENT
Broadcast operation to followers: [Operation = DELETE_CLIENT
Processing... [Operation = UPDATE_CLIENT, client=[5, Pippo, 4000]]
Broadcast operation to followers: [Operation = UPDATE_CLIENT, client=[5, Pippo, 4000]]
```

Server 1 reads bank database (operation 5) after deleting account number 5.

```
5
[1, Surbhi Sonkiya, 1000]
[2, Giovanni Vuolo, 2500]
[3, Raffaele Perini, 6500]
[4, Zsolt Dargo, 4000]
```

Operation 5: Read bank database

```
5
[1, Surbhi Sonkiya, 1000]
[2, Giovanni Vuolo, 2500]
[3, Raffaele Perini, 6500]
[4, Zsolt Dargo, 4000]
```

Operation 6: Exit from application

Server 1 selected operation 6. Exit from application.

```
6
Session finished
```

Server 2 is elected as leader when server 1 goes down.

```
$$$$$ You are the leader! $$$$$
Node id: /election/node-0000000037
```

6. Group view: Future improvements in design

There is scope for improvements in this application. This application was developed focusing on the deployment of the various properties

like fault tolerance, consistency, availability and replication that an application must posses in a distributed environment. This objective has been achieved, however, there could be few additions:

a. **User interface**: Currently, the user interface could be improved to make it more user friendly. Additionally, a web app or a mobile app could also be developed.
b. **Control flow**: A possibility to go back to the main menu if user selected one of the options by mistake. For example, user intends to do an update and instead of selecting option 3, he/she selected option 4. In this case, there should be a provision to go back to main menu without executing the selected option.
c. **External Database**: Integrate external database for better management of customer data.

7. Source Code

The source code for this project is present in the below Github link. It is a public repository, presently shared between the two authors of the project.

Link: https://github.com/surbhisonkiya/Banking

8. Problems Encountered

a. **Accessing the right node in the right order.** Struggled to understand why, when forwarding operations to followers, these operation where in the wrong order. The simple solution was to order them through the watcher, but the problem was arriving in understanding of the watchers and how to work with them.

b. **Forwarding operations to a new node.** Sometimes the leader did not react to the arrival of a new member and when it did, it forwarded only a few operations or not at all. The solution was to trigger an operation from the new node to “wake” the leader.

References:

1) https://zookeeper.apache.org/doc/current/index.html
2) https://zookeeper.apache.org/doc/r3.1.2/zookeeperProgrammers.html#ch_zkGuarantees
3) http://my-zhang.github.io/blog/2014/04/11/consistency-model-in-zookeeper/