Ira A. Fulton Schools of Engineering
Arizona State University
Engineering | School of Computing, Informatics, and Decision Systems Engineering

CSE 531: Distributed and Multiprocessor Operating Systems

Name: Thien Tran

gRPC Project

Problem statement

A bank asks to build a system (referred as the system onward) where it can interact with its clients and allow each branch to communicate with one another.

Goal

Explanation

This assignment project uses the user requirement discovery framework in the agile project management methodology. The framework starts with an epic theme which indicates the true need of the bank. The theme links to 5 business requirements. They allow system designers and stakeholders to find out logic gaps, determine acceptance criteria and effectively control the project scope. Furthermore, they give directions to software developers on the technical, technology and specification requirements in the implementation process (Appendix 1).

Epic theme (business goal)

The goal of the assignment 1 is to implement a system where multiple clients and multiple branches can communicate with one another.

Business requirements

As a client, they want to deposit an amount of money (business requirement 1, breq 1)

As a client, they want to withdraw an amount of money (breq 2)

As a client, they want to query the amount of money (breq 3)

As a branch, they want to increase the balance across to other branches (breq 4)

As a branch, they want to decrease the balance across to other branches (breq 5)

Technical goals

Overview

Although many different implementation designs and algorithms can result in the same functionalities of the system, the implementation focuses on the distributed system implementation. Because the 4 design goals of a distributed system - namely openness, scalability, resource sharing and distribution transparency, align with the bank's requirements.

3 pillars of a distributed system are: synchronization, coordination and communication. The project will categorize the bank's requirements into these 3 important pillars when designing the system. Each pillar provides the basic framework of assumptions, considerations, algorithms and architectural parts to build the system.

Synchronization

- A transaction made on one branch is updated to the other branches (technical requirement 1, treq 1).
- Clients and branches can function regardless of their hardware and software specifications (treq 2).

Coordination

- A client contains a list of events they want the bank to do (treq3).
- A deposit transaction from a client incurs a propagated deposit function call from the acting branch to the other branches (treq4)
- A withdraw transaction from a client incurs a propagated withdraw function call from the acting branch to the other branches (treq5).

Communication

- A client can only interact with a bank which has the same unique ID which is stored internally in the client stub. Each id can only communicate with the branch having the same corresponding id (treq6).
- When a bank is created, it initializes a list of the identifications (ID) of the other branches (treq 7)
- When a bank is created, it initializes a list of the stubs of the other branches (treq8).

Assumptions and constraints

- No transaction occurs concurrently.
- The number of branches and clients are predetermined.
- The branches are only created at the beginning.
- No update occurs on the list of branches and their stubs after they are created.

Set up (Technology Requirements)

- Built on: Windows 10 (19042.867)
- Tested on: Windows 10 (19042.867) and Ubuntu 20.10
- Programming language: Python v3 (3.8.5)
  Libraries:

Package	Version
grpc	1.31.0
grpcio	1.36.1
grpcio-tools	1.36.1
protobuf	3.15.6
MessageDelivery_pb2	1.31.0 (grpc generated)
MessageDelivery_pb2_grpc	1.31.0 (grpc generated)
logging	0.5.1.2
time	3.8.5 (Python built-in)

Implementation process

Choice of project management methodology

This assignment is small, and its components are easy to iterate over multiple phrases. It is suitable for the agile methodology. The initial choice plans out 4 iterations namely: client communication, client transaction, branch intra-communication and branch transaction (Appendix 1 A,B,C,D)

Overall design

As previously mentioned, meeting the 3 technical goals' satisfactory conditions in the previous section are based on the 4 design goals (resource sharing, scalability, openness and distribution transparency) while preserving the 2 properties of the system: autonomy of nodes and system coherency. Each of the design goals satisfies a few out of the above goals and their requirements.

Autonomy of nodes

Admission control

A closed system is a preferred choice since a client is only assigned to a specific branch (treq6), and each branch keeps a specific list of addresses of the others (treq 7-8).

Organization of nodes

2 types of communication occur in the project, branch-to-client communication and branch-to-branch communication(breq 1-3 and breq 4-5).

Branch-to-client communication (Appendix 2)

The project implements the client-to-branch communication as a simple client-server organization. Each client's machine has an id. Each id will only communicate with the branch having the same corresponding id.

The client's machine will contain an application which only sends and receives the RPCs to the branch with the corresponding id (treq 6).

Branch-to-branch communication

The branch-to-branch communication is implemented as a decentralized structured organization . Each branch is treated as an equal peer to another (breq 4-5) . They all have the same access rights. Each branch has a list of id to generate a stub list (treq 7).The stublist (branch.stublist) which serves as an address topology for address look-ups (treq 8).

System coherency

Openness

Separate function from policy (Appendix 3,4,5)

This project uses the process-oriented programming approach. It promotes the interoperability, portability, extensibility of the system. Further, project 2 and 3 of the course may relate to this one, a high probability exists such that the system or at least their parts will be open to the other systems or components in project 2 and 3.

Distribution transparency

According to the technical goals, the project only aims at the following: access transparency and replicate transparency. The requirements do not detail the specifications for location, relocation, migration and failure transparency. Lastly, the requirements do not mandate concurrent processing which means the concurrency transparency is not needed.

Access transparency

The project implements Protobuf3 as an interface definition language and gRPC as the transmission protocol. Instead of the traditional message passing, gRPC allows either a client or a branch to directly call a procedure of any machine in which another branch's application resides. Protobuf3 messages carry binary bits of the parameters needed and carry back the result responses.

Replicate transparency

Whenever a client makes a transaction, the eventHandler function calls for the Propagate_withdraw and Propagate_deposit to update the global balance variable branch.balance of the other branches (treq 4-5).

Scalability

The requirement specifications do not include how they want the system to be scaled. The design does account for a few functions for scalability. Although they are unimplemented, they serve the purpose for future references.

Resource sharing

The project's implementation allows a branch application either to fit in its own machine or to function a separate process in a distributed resource machine.

Explain Implementation results

Iteration 1: Client communication (Appendix 1.A)

To create gRPC protocols, the project uses the syntax of protobuf 3. Service client_request is created to handle all the communication in the project.The service takes the clientrequest message object as the input and the response message object as the output.

clientrequest message class consists of id, type and multiple event classes. It dictates its credential id, the type of sender either a client or a branch, and the events it wants to execute on the receiver side. The event class in turn contains id, interface and money. It tells the receiver the types of actions to be executed.

The response message class consists of id and multiple recv classes. It tells the receiver the results of the request sent and whether the actions are executed successfully.

gprc-tools uses the file to generate MessageDelivery_pb2 and MessageDelivery_pb2_grpc.

The initialization of the client object is predefined by the assignment instructions. The object stores an id and a list of events (P1, appendix 3).

Based on client.id and a predefined channel, the createStub function creates the address of the branch (P2, appendix 3). Since no instruction specifies the address of branches, the project uses the channel (localhost:80050 + id) assigned to each branch's address for the purpose of testing.

The executeEvent function takes in the branch's address (stub) and the events to be sent (P3, appendix 3). It sends the events to the requestHandler function to package the events in gRPC format (P4, appendix 3). In turn, executeEvent receives back the wrapped events in gRPC format. It then creates a gRPC request and sends the request to the corresponding branch's address.

The requestHandler processes the events received and wraps them into the gRPC format object (P3, appendix 3).

In the second part of iteration 1, the object branch is defined. It initializes 4 variables: id, balance, branches and stublist (P1, appendix 4). The id, balance and branches are user inputs while stub is created based on the branches variable and a predefined channel pattern: localhost:80050 + id.

The client_request function generated from MessageDelivery_pb2 transforms the request in gRPc format back into the python readable format (P1, appendix 5).

The serve function creates a server object as a listener thread which waits for the requests. The listener takes in the servicer object (the branch) itself. The listener port created using the channel above.

Iteration 2: client transaction (Appendix 1.B)

The request is then sent to the clientHandler to separate the group of events into each individual event (P2, appendix 5). During iteration 1, the clientHandler function sends each individual event to eventHandler (P4, appendix 5). Each event will return a setReceiveMessage message. Each message is wrapped into the return_message. The return_message is then returned back to the client_request.

The eventHandler (P4, appendix 5) is responsible for matching the event to each corresponding function (deposit, withdraw or query). It sends the parameter needed to the corresponding function for servicing. It returns the setReceiveMessage message to the clientHandler.

The corresponding functions (deposit, withdraw and query) (P7,8,9, appendix 5) increase, decrease or query the balance. Each of these functions calls the setReceiveMessage to report the successful status of the event. They return the message back to the eventHandler.

Iteration 3: branch communication (Appendix 1.C)

createStub function uses the list of branches from the object to create a stubList of branch addresses starting from local localhost:80050 to (localhost:80050 + N) with N is the number of branches (P2, appendix 4).

branchRequestHandler during iteration 3 only requires the stub input (P3, appendix 5). It ensures the branch object can send a request to another other branch. It creates a request object and sends a predefined event to another branch for functionality testing.

Iteration 4: branch transaction (Appendix 1.D)

Propogate_Withdraw and Propogate_Deposit functions are added at the end of the withdraw and the deposit functions so that these new functions update the balances from the other branches (P11 and P12, appendix 5). Propogate_Withdraw and Propogate_Deposit functions iteratively send a request to increase or decrease to each branch address.

A new functionality is added to the clientHandler (P2, appendix 5). It separates the client request from the branch request. The branch request is then passed on to the branchEventHandler for processing (P3, appendix 5).

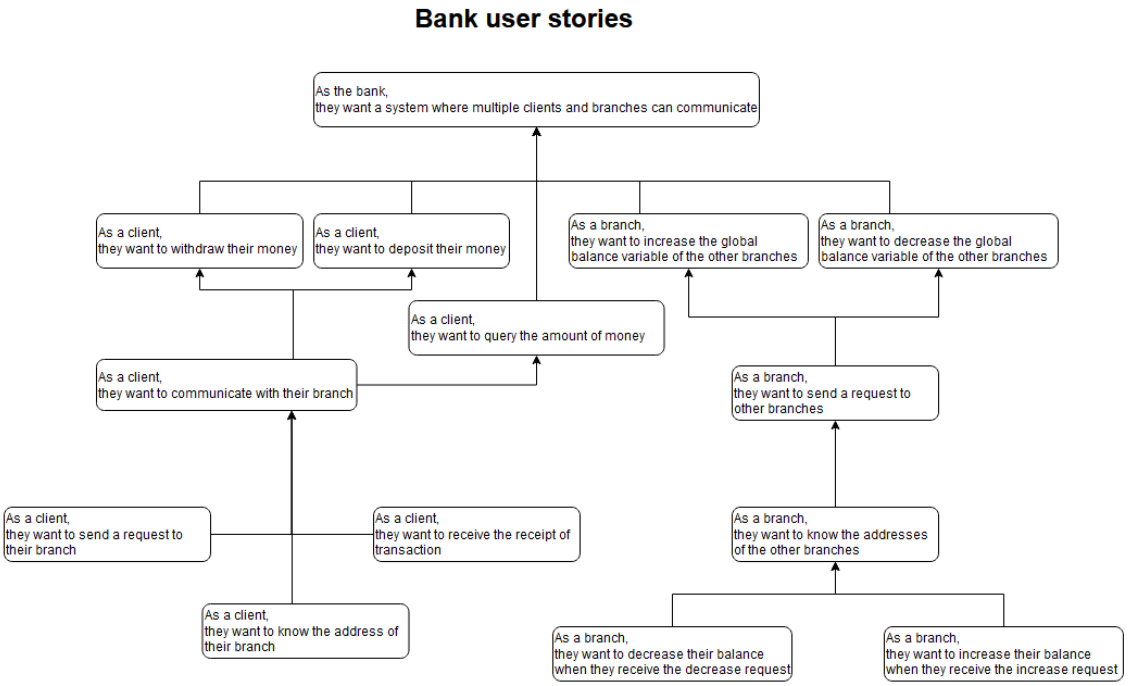
Similar to the eventHandler handling the client request, branchEventHandler sends the request to either branchDeposit or branchWithdraw and receives a setReceiveMessage message if the result is successful (P5 and P6, appendix 5). branchDeposit and branchWithdraw are similar to the deposit and withdraw functions.

Unimplemented features

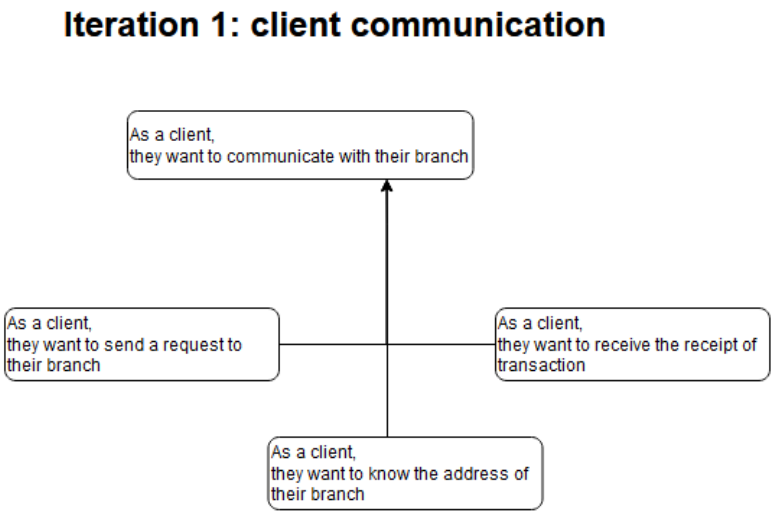
As confirmed by the GSA that no testing shall be carried out, the following files are reserved for internal use only: branch1.py, branch2.py, testenv.py and testenv2.py

Appendix

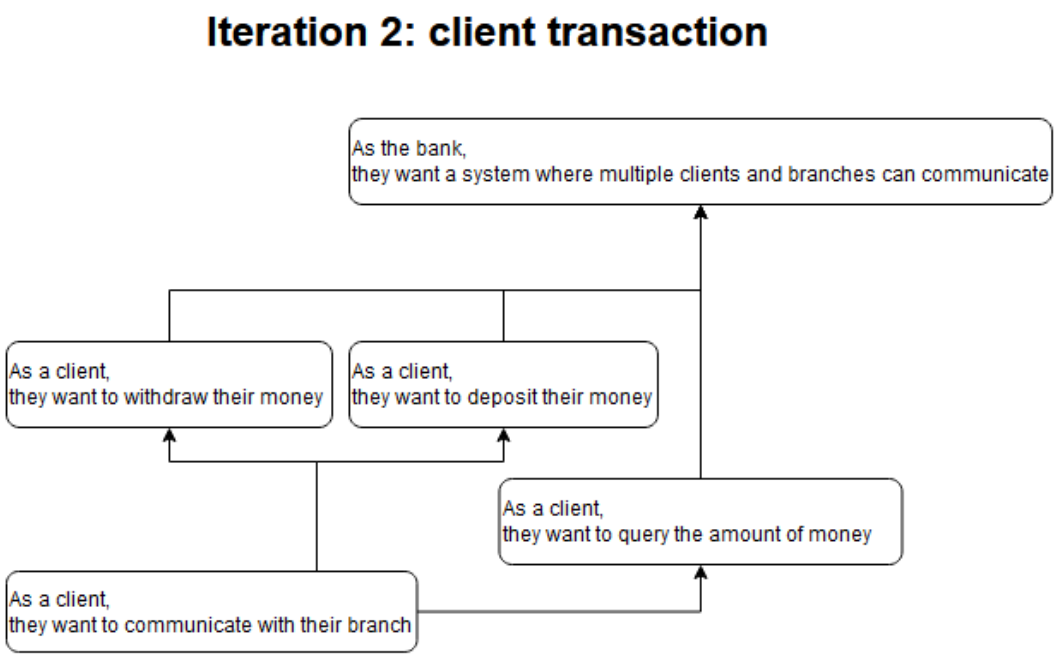
Appendix 1



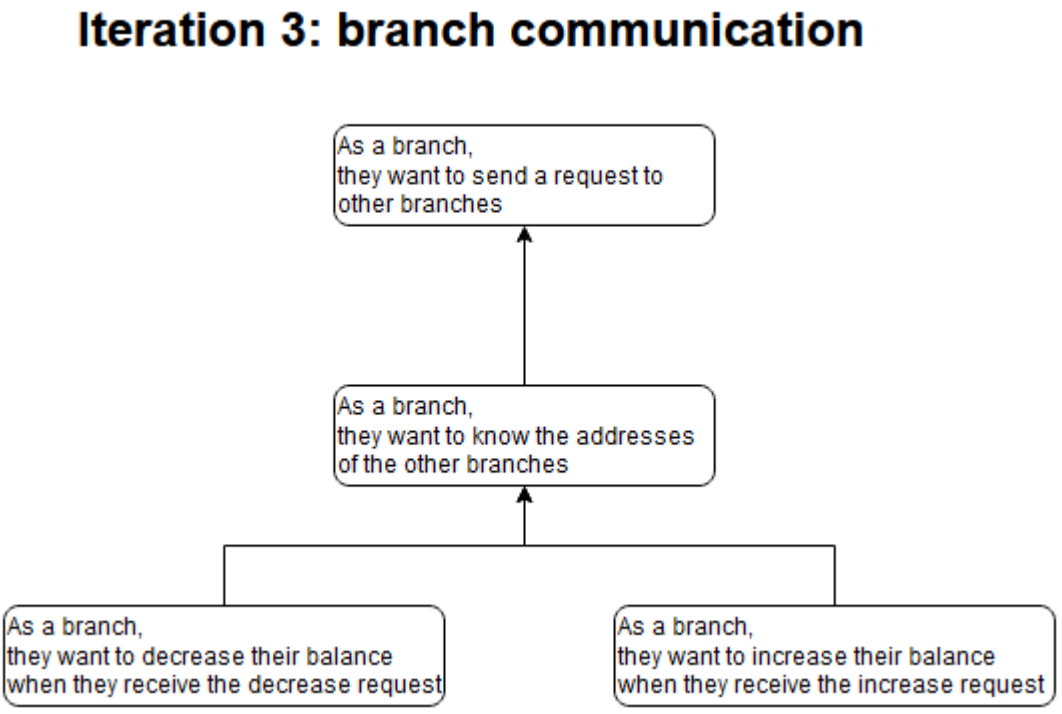
Appendix 1.A



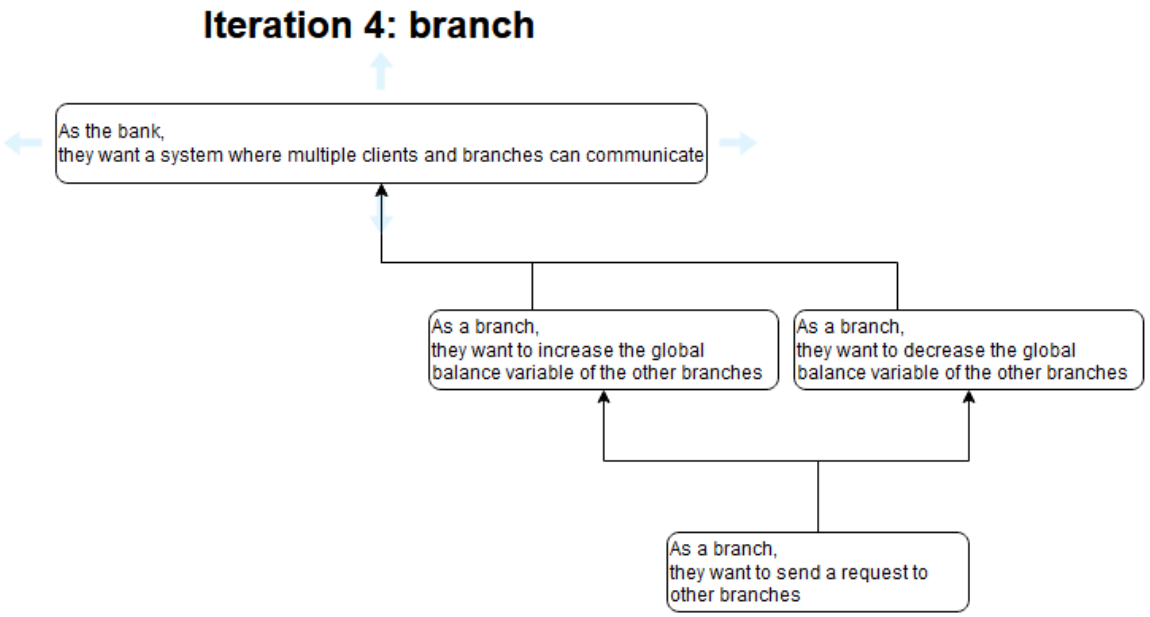
Appendix 1.B



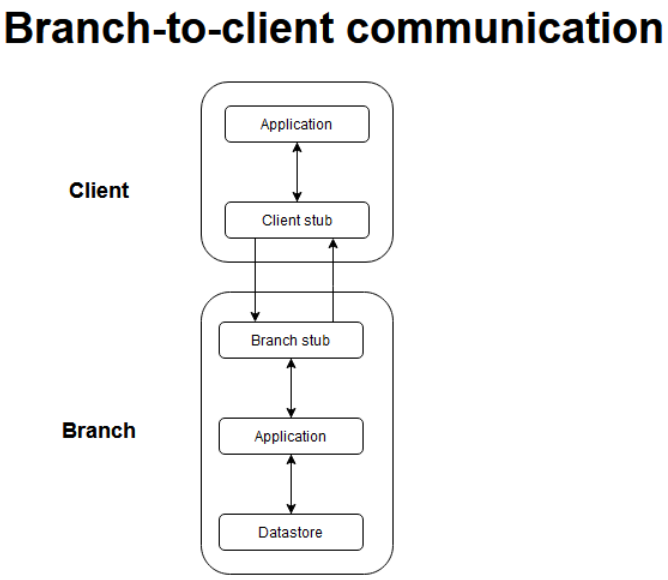
Appendix 1.C



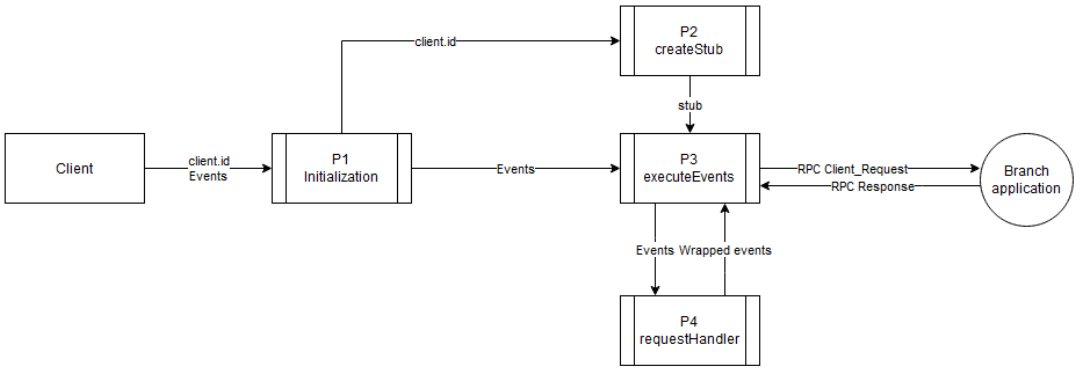
Appendix 1.D



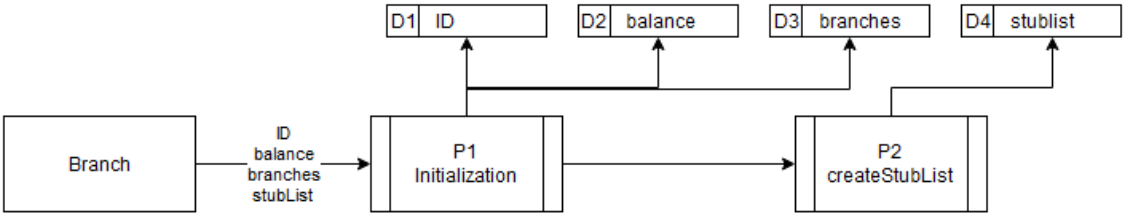
Appendix 2



Appendix 3: Client process



Appendix 4: branch initialization



Appendix 5: branch implementation