Global-Snapshot banking system

Simone Sandri
Dept. information engineering and computer science
University of Trento, Italy
simone.sandri@studenti.unitn.it

Damu Ding
Dept. information engineering and computer science
University of Trento, Italy
damu.ding@studenti.unitn.it

***Abstract*—This application is built with JAVA RMI, which is an object-oriented equivalent of remote procedure calls (RPC). For instance, the distributed snapshot uses Chandy-Lamport snapshot algorithm and applied into a simple banking application. Every node can be considered a server as well a client. The system is peer to peer: any bank can connect to each other and knows the exact location of all the banks with their IPs and ports. In addition, at most one snapshot can be taken at one time.**

1. Introduction

The aim of the project is to create a distributed bank system that is able to send and receive money from/to several branches. The system should also be able to obtain a global snapshot through the Chandy and Lamport algorithm any time it is required. Each branch of the system starts with a balance of €1.000.00, and at any time it sends from €1 to €100 to another random branch in a safe way. This means that each money transfer should be acknowledged with at most one semantic. To obtain this, the sender tries to send money to receiver until an acknowledgement comes back, and only then it changes its balance. On the other side, the receiver, should filter out all the incoming requests, removing the duplicates based on the sequence number. To be correct, in the system, at each time the sum of each bank balance and the money in transit must be constant. To check this invariants a periodically check must be done, capturing the global system state. The results of the global snapshot algorithm are collected by all banks in some log files.

2. How it works

Run *RunServers.java* first, when you see "The server is ready!" then run all *RmiClient.java* files. On every client screen you can find all the information about money transfer. In each *Log_server.log* file there are the snapshot logs. It's obvious that the amount in the snapshot is always kept as a constant €5.000.000.

3. Remote method invocation

The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM. The RMI provides remote communication between the applications using two objects stub and skeleton.

3.1. Features

- The application need to locate the remote method
- It need to provide the communication with the remote objects
- The application need to load the class definitions for the objects

3.2. Procedure

- Create the remote interface
- Provide the implementation of the remote interface
- Compile the implementation class and create the stub and skeleton objects using the rmic tool
- Start the registry service by rmiregistry tool
- Create and start the remote application
- Create and start the client application

3.3. stub

- It initiates a connection with remote Virtual Machine (JVM)
- It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM)
- It waits for the result
- It reads (unmarshals) the return value or exception
- It finally, returns the value to the caller

3.4. skeleton

- It reads the parameter for the remote method
- It invokes the method on the actual remote object
- It writes and transmits (marshals) the result to the caller

3.5. RMI in our case

- Remote interface: All the remote interfaces are defined in AddServer.java, every interface should throw RemoteException due to the communication error.

- Implementation of the remote interface: The key of the project Global-Snapshot is built in AddServer-Impl.java
- Registry service: We bind 5 banks to 127.0.0.1:1090-1094/Hello, the client can quickly find all the servers.
- Remote application: In RmiServer.java file, the server receives the random amount from the clients and use the methods in AddServerImpl.java to keep the balance between each banks.
- Client application: In all RmiClient.java files. , the clients will send random amount from 1 to 100 to a random bank. If the amount of the bank is not available, the random amount will return a value between 1 and the available amount.

4. Global Snapshot

4.1. Overview

The algorithm works using token messages. Each process that wants to initiate a snapshot records its local state and sends a token on each of its outgoing channels. All the other processes, upon receiving a token, record their local state, the state of the channel from which the token just came as empty, and send token messages on all of their outgoing channels. If a process receives a token after having recorded its local state, it records the state of the incoming channel from which the token came as carrying all the messages received since it first recorded its local state.

4.2. Implementation

To implement the Lamport global snapshot algorithm we introduced on server side the following variables:

- **boolean record**: It's a boolean variable that says if the snapshot algorithm is running or not;
- **boolean[] tok**: It's a boolean array of length equals to the number of servers, initialized to false. Every time a token arrives from node i, the server set tok[i]=true;
- **int[] val**: It's an integer array of length equals to the number of servers, initialized to 0. Every time a transfer request arrives from node i, the server satisfies the request and if record=true and tok[i]=false, then the amount of the transfer is added to the val[i] cell of the array. In other words it will store all incoming values of each node that arrives before the token from that node, when the algorithm is working (record=true).
- **int snapshot amount**: It's an integer value that contains the total money amount on the server when the first token arrives.

Additionally some methods has been implemented:

- **tokenLock & updateLock:** These two locks are used before updating the amounts for the concurrency control.
- **SetToken(int id, Map<Integer,AddServer>server)**: the tokens arrived from node id is processed by changing the tok[id] array value. If the arrived token is the first of the snapshot, the server will send its token to all other nodes. If id is the last missing token of the network, the server will broadcast its local snapshot values to all other nodes;
- **update(int identifier, int[] vals, int server amount)**: the local snapshot (vals[] and server amount) coming from node identifier, is processed, collected and stored into the snapshot amounts[] array and the snapshot matrix[][] matrix, to create a global snapshot view.

4.3. Algorithm

4.3.1. Initialization. As discussed before, each active server is provided by the previous methods and variables. The tok[] array is set to false, the val[] array to zero, the snapshot amount to zero and the record variable to false. Every time a client performs 1000 requests it calls the setToken() method of each active servers, sending to them a token, so that the algorithm can start.

4.3.2. Processing. Every time a token arrives, the server checks if it is the first one of the current snapshot, if so it sets the record variable to true. At the same time it registers that a token arrived from the node i, setting tok[i]=true, and if the token is the first to arrive, it sets also the snapshot amount variable value to the current node balance. Then, according to the original algorithm, the server sends a token to each other server. Every time a value arrives from another node i, the server checks if the algorithm is still working and if i has not already sent a token. If we are in these conditions, the value of val[i] is incremented by the incoming amount.

4.3.3. Termination. Every time a token arrives the server checks if a token is arrived from each node in the network by exploring the tok[] array. If so the local snapshot of that server is finished and the val[] array with the snapshot amount variable is sent to all components to the network by calling the update() method. All the variables used to compute the snapshot will now be reset to their original values according to the init section described previously.

4.3.4. Results' collection. At the end of the algorithm all nodes collect the local snapshot of each other node in the network, so that is able to compute the global snapshot. To do this, all snapshot amount values are collected in an array called snapshot amounts[] and all values coming from the various val[] arrays are stored into the snapshot matrix[][], where snapshot matrix[i][j] will contain the value transferred from node i to node j before node j has received a token from node i. In this way all nodes have the same global snapshot. All the logs will be stored in the corresponding server_log files.

5. Conclusion

In this report, we have modeled the Global-Snapshot banking system with RMI and Chandy-Lamport snapshot algorithm. The bank can be considered as a server also a client. RMI can help us to build the communication between each banks. Moreover, the clients have no need to store the methods in themselves, just pass the parameters to the servers, and the servers will use their methods to execute them then pass the results to the clients. Upon one node receives the token, the snapshot starts, the total amount in the snapshot is always kept constant. The node passes the token to all the other nodes until all the nodes receive the tokens from all the others, the snapshot terminates. This allows us to prevent bank withdrawing problem. Therefore, Chandy-Lamport snapshot algorithm is a good way to solve recording a consistent global state of an asynchronous system.