Università degli studi di Trento
Distributed Systems Project
A.Y. 2015-2016

Global Snapshot In a Distributed Banking Application

Submitted to:
Prof. Gian Pietro Picco
Timofei Instomin

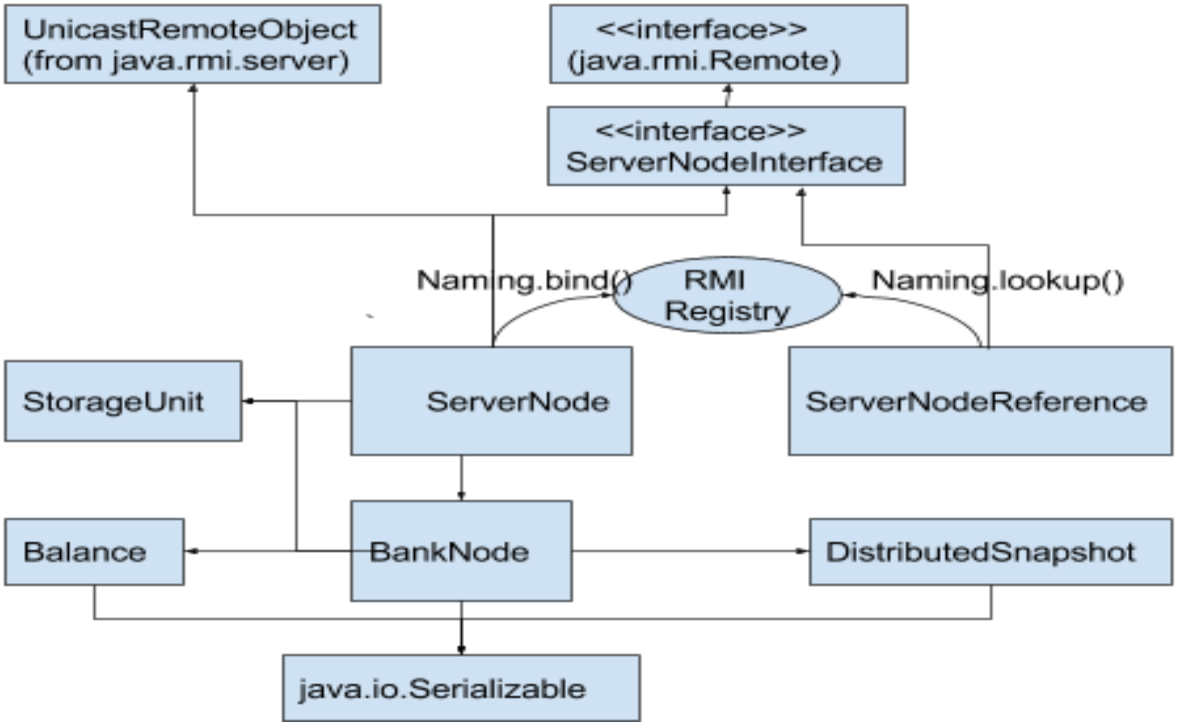
By:
Aymen Jlassi -[MAT. 168055]
Email: aymen.jlassi@studenti.unitn.it
Kinde Tefera - [MAT. 174219]
Email: kindegutema.tefera@studenti.unitn.it

1) IMPLEMENTATION

We have used java Remote Method Invocation (RMI) to implement the project, which is an equivalent of RPC for Objects in Java application. RMI allows Object functions calls between different Java Virtual Machines (JVMs). The JVMs can be located in different computers, yet they can invoke methods between each other. This communications is enabled between different Java Virtual Machines throw Objects called Stub/proxy and Skeleton.

2) ARCHITECTURE

The simple architectural diagram of our design looks like the following.



Let us briefly discuss each components

1. ServerNodeInterface

This is the core of RMI communication in our implementation, which is a subclass of ***java.rmi.Remote interface***. The important principle here is the definition of the Remote Object and their implementations are separate concepts and RMI allows them to run on separate virtual machine. The definition of the Remote Object is provided in the interface, ServerNodeInterface and the actual implementations is provided in the class ServerNode. Hence, local BankNodes/client are concerned about the definition of the services/Methods while the Remote BankNode/Servers are focused on providing the service.

In this Remote interface, we defined the following Remote methods to be implemented by the ServerNode (the Remote Object).

- public BankNode getBankNode() throws RemoteException
- public void addBankNode(int nodeID, String nodeHost) throws RemoteException
- public boolean receiveMoney (int senderNodeId, int amount) throws RemoteException

- public void sendMoney (int receiverNodeId, int amount) throws RemoteException
- public void receiveSnapshotToken(int nodeID) throws RemoteException

2. *ServerNode*

This is the Remote Object/class that implements the ServerNodeInterface and the implementation for each Remote Methods defined in the Remote interface is provided in this class. This class should also be a subclass of ***java.rmi.server.UnicastRemoteObject*** in order for the Remote Object to be exported to the caller.

For this Object to be invoked remotely: We have to create and install Security manager; We have to create and export one or more Remote Objects and finally we have to Register/bind at least one Remote Object with RMI registry using Naming.bind() service, which imports java.rmi.Naming and Directory interface for bootstrapping purpose.

In addition to the methods to be invoked Remotely, we have implemented the main method here to run the ServerNode and enable the RMI communication.

3. ServerNodeReference

This is a class that deals with RMI for Remote Object, it references the Remote ServerNode Object in RMI Registry. The BankNode that needs to invoke one or more Remote methods from the Remote BankNode have to reference the Remote Object using Naming.lookup() service. Basically this class is a Remote Reference which is ***a proxy/a Stub*** for the local BankNode. The Stub contains information that allows the local BankNode to connect to a Remote ServerNode Object, which contains the implementation of the Remote Methods defined in the ServerNodeInterface.

4. *BankNode*

This is a remote class that represents the BankNodes. Each BankNodes is identified by ID number and hostname. In addition, this class implements ***java.io.Serializable*** interface so that when the methods in this class are invoked, it will be Serialized and sent to the local Object which is call by value/parameter instead of call by reference.

5. *Balance*

This is remote class for for computing balance in each BankNodes. This class also implements java.io.Serializable interface, hence its value is serialized and sent to the local object.

6. *DistributedSnapshot*

Remote class for implementing snapshot algorithm (will be discussed in the next section)

The three classes above, BankNode, Balance and DistributedSnapshot implement the interface java.io.Serializable instead of java.rmi.Remote. Serializable Object is the one whose value can be marshaled. This means that the contents of the Object can be represented in a pointerless form a bunch of bytes and then reconstructed into their respective values as needed. This is useful, for example, in saving contents of an Object into file and then reading back the Object. RMI uses the java Object Serialization mechanism to transport Objects by Value instead of by Reference between Java Virtual Machines.

7. *NodeState*

An enumeration that indicates the NodeState. All ServerNode Object could be either in Connected State or Disconnected state.

8. StorageUnit

This class is a convenient class to deal with each BankNodes internal snapshot recording. It is used for Creating storage folder, Writing the Distributed snapshot results as CSV file and updating the result. It keeps CSV files of each BankNode under storageFolder directory in format: {snapshotID},{local balance},{sum of all incoming transfers upon receiving the marker}

3) DISTRIBUTED SNAPSHOT

The second part of the project is implementations of Global Snapshot Algorithm in the Distributed Banking Application. Distributed Snapshot uses Chandy-Lamport Algorithm to capture the global state of a distributed System. The algorithm works using Token/Marker messages. Each BankNode that wants to initiate the Global snapshot first records its local state and sends Token on each of the outgoing links. Upon receiving the Token, all the other BankNodes record their local state, the state of the channel from which the token came as empty and send the Token to all outgoing channels to help propaget the Marker. If a process receives the marker after recording the local state, it records the state of the incoming channel from which the marker cam as carrying all the messages received since it already recorded its own local state.

Assumptions for the algorithm to work:

1. Reliable communication Links and FIFO channels
2. The Algorithm is adopted so than multiple Snapshot can be taken at the same time by different BankNodes identified by the snapshotID.
3. Strongly connected Link and any BankNode may initiate the snapshot Algorithm
4. The snapshot process doesn't interfere with the normal execution of the System.
5. Each BankNode/process records its local state and the state of its incoming channel.

Distributed Snapshot effectively selects a consistent cut (no messages jump from future into the past and no message receipt is recorded without send) and is a non-blocking algorithm.

4) HOW TO RUN

1. Open terminal and change your working directory to the file location: JlassiTefera/GlobalSnapshot_IN_Distributed_Banking_System/src

2. Compile the Program classes using the command: ***javac ServerNode.java***

3. Launch the Program using the command: ***java ServerNode***
You will see the following output:

ENTER COMMAND: [MethodName NodeHost nodeID existingNodeHost existingNodeID] without the bracket
TO CREATE THE FIRST BANKNODE: createBankNode localhost 10
TO JOIN BANKNETWORK: joinNetwork localhost 20 localhost 10
TO VIEW THE NETWORK: viewTopology
TO VIEW GLOBAL SNAPSHOT: dss
Print IPAddress: IP_Address
BANKNODE IS READ FOR REQUEST…

4. To Create the first BankNode in the graph, write command: ***createBankNode localhost 10***

5. For the second BankNode to join the Network, open a new terminal and change your working directory to the file location. Launch another ServerNode like step 3 with the command ***java ServerNode*** and then write the command: ***joinNetwork localhost 20 localhost 10***
AS soon as the 2nd BankNode joins the Network, it will start to send and receive money randomly.

6. For the third BankNode to join the Network and to continue building a Distributed Banking Network with a bunch of BankNodes, which will transfer random amount of money in the range between 1 - 100 among themselves, repeate step 5 with different nodeID.

The new BankNode can join any of the existing nodes and announce itself to all the existing BankNodes. The existing BankNodes will automatically add the new node to the list of BankNodes they are aware of.

7. To view the topology of the graph, enter the command: ***viewTopology***

8. To take the Distributed snapshot, enter command: ***dss***