**DESIGN DOCUMENT**

### Members of Group 13

1. Nichit 2017MCS2089
2. Raj 2017MCS2098
3. Fai 2017MCS2525

### Technologies Used

### Language: Java

### Simulator: Mininet

### Overview

We have implemented the project in a **Non-Blocking Fashion**. So, each client node will be running independentlyand it will also be simultaneously listening for the broadcasts and the receive operations that are destined to it.

1. We have used the Pastry implementation of Distributed Hash Table for storing the public keys of the nodes.
2. Two phase commit will be required before initiating every transaction.
3. The transaction will be broadcasted once it has been verified by the witness and the receiver.
4. After receiving the broadcast every node verifies the digital signature of the sender and also verifies the input transactions provided before adding that transaction to its ledger.
5. We have implemented the verification of the ledger for each node by generating the hash code.
6. And lastly each transaction can only be used once as the input transaction. After that it will be invalidated in every node’s ledger.

### Protocol

Each node has a generic listener (server socket). The sender node will have a client socket for sending out messages. Based on the message header, the receiver will determine which internal function to call.

### Description of Major Classes

1. **Client.java (Node) (Only major methods)**

|  |  |
| --- | --- |
| Method | Description |
| initiateTransaction() | Any node which wants to initiate a transaction will call this method |
| listenTransaction() | The receiver and the witness will decide whether to commit the transaction or not |
| Broadcast() | Transaction initiator will broadcast the transaction information to all online nodes |
| broadcastReceive() | All nodes will listen to the broadcast, verify the digital signature and input transactions. Once verified it will write the transaction to its ledger |
| handleTransactionResponse() | It will verify if the transaction has been committed by both the witness and the receiver. Once verified it will call broadcast() method. |
| selectInputTransactions() | It will select the set of valid input transactions from the ledger that are going to be sent in the transaction |

1. **Ledger.java**

|  |  |
| --- | --- |
| Method | Description |
| addTransaction() | It will add the successfully verified transaction to that particular client’s ledger |
| verifyTransaction() | It verifies if the input transaction provided in the transaction are valid or not |
| invalidateInputTransaction() | It will invalidate the input transactions that are used in the transaction in the current client’s ledger |

1. **Pastry Class**

|  |  |
| --- | --- |
| Method | Description |
| get() | Get the public key if present in the leafset otherwise route to the next node |
| nodeInitialization() | It initializes the leafset, neighborhood set and the routing table of a new node |
| routeToNode() | It will locate the correct location of the new node in the network by routing and then it will place it there |
| broadcast() | The new node will broadcast that it is there in the network. So that other nodes can make the appropriate changes |
| addBroadcastNodesMap() | It will receive details of the newly added node and update it s respective variables |

### Some important class schemas

**Transaction class**

**private** String transactionId;

**private** **double** amount;

**private** String senderId;

**private** String receiverId;

**private** String witnessId;

**private** List<String> inputTransactions;

**private** **boolean** witnessCommitted;

**private** **boolean** receiverCommitted;

**private** **boolean** valid;

**Ledger class**

**private** List<Transaction> ledger;

**Client class**

**private** String accountId;

**private** Address address;

**private** ServerSocket serverSocket;

**private** List<Transaction> inProgressTransactions;

**private** List<SignedObject> inProgressMessage;

**private** Integer transactionCounter;

**private** ClientWriter clientWriter;

**private** Ledger ledger;

**private** PrivateKey privateKey;

**private** PublicKey publicKey;

**private** Pastry pastry;

**private** Map<String, Address> nodesMap;

**Pastry (Distributed Hash Table) class**

**public** **static** **final** **int** ***B*** = 2;

**public** **static** **final** **int** ***L*** = 4;

**public** **static** **final** **int** ***M*** = 8;

**private** Map<String, PublicKey> pkMap;

**private** String accountId;

**private** Map<String,Address> nodesMap;

**private** String[][] routingTable;

**private** List<String> neighborhoodSet;

**private** List<String> leftLeafSet;

**private** List<String> rightLeafSet;

**private** PastryWriter pastryWriter;

**private** PastryListener pastryListener;

### Scenarios that have been tested

1. **Transactions**
   1. 2-phase committed transaction, accepted by all parties.
   2. Transaction ID is unique by the Lamport clock.
   3. Broadcast of transactions happen successfully.
   4. Insufficient balance.
   5. Invalid receiver.
   6. Invalid witness.
   7. Transactions are rejected by witness / receiver.
   8. Invalid input transactions.
2. **Ledger**
   1. Valid transactions are added to the ledger.
   2. Changes are automatically processed by ledger (i.e. A to B 700, with input transaction of 1000. Then The ledger should have 2 entries: A to B 700 and A to A 300).
   3. Input transactions are invalidated after they are used – to avoid double spend.
   4. When multiple transactions are broadcasted, each node’s ledger process those in the same sequence.
   5. Transaction with invalid input transactions is rejected.
   6. Transaction with insufficient balance is rejected.
   7. When a new node is added.
      1. Ledgers are copied to the new node from the neighbor.
   8. In the end, all ledgers’ hash codes are the same.
3. **Digital signature**
   1. Broadcast message and Transaction message are all checked with digital signature.
   2. All digital signatures are generated randomly.
   3. Private key is only kept in the original node.
   4. Public key is available only in DHT.
4. **Distributed Hash Table**
   1. During start up the routing table, leaf set and neighborhood set are initialized correctly.
   2. Each node only stores its own and leaf set nodes public key.
   3. Able to route from any node to any other node to fetch the public key.
   4. When a new node is added.
      1. New node will have its routing table, leaf set and neighborhood set initialized.
      2. Other nodes will have their routing table, leaf set and neighborhood set updated.
   5. When a node is deleted.
      1. Routing is not affected.
5. **Others**
   1. Each node knows the identity (IP Address) of the other nodes