This is a report for DISSY Handin2 Exercise 4.6 of group 8. This report briefly explains how we build this p2p system and extend some detail on key steps. To run our code, simply run this command in the terminal. Wait for several seconds until the program prints out the ledger of all nodes.

---

go run P2P_ledger_final.go

- The report should briefly describe how you implemented the system and how to run it.

We divide this system into three different parts/modules, data structure, net utils, and test modules.

Data structure

In the data structure module, we defined several “struct” in golang to help us solve the problem. Besides, we have some util functions to help us initialise or operate these structures easier.

First we define a structure called Transaction, which contains transactionID, From, To and Amount which are the basic description of a transaction. Function Transact is used to make / process a Transaction locally. It takes a pointer to object Transaction as input, and transfers the “Amount” money from sender “From” to receiver “To”. Note that we use mutex lock to ensure this operation is atomic.

Then we define Ledger structure, which uses a map to record the balance of each account, and it also has a Mutex lock which can be used to prevent context switching when we are processing the transaction. Function MakeLedger simply creates a new ledger object, initialises five accounts and sets their corresponding balance to 0.

To do communication between different nodes, we create a Message struct, which contains two fields, Message Type and Message Content. Their meaning is revealed as their name. All the messages sent between nodes use this structure. We use json to marshal this structure before sending, and unmarshal the byte stream to the Message object every time when we receive a new message.

At last, we have two similar structs, Peer and PeerInfo. Peer contains this node’s IP address, port, Peers(which is a slice of PeerInfo), and a Ledger. PeerInfo only contains a node’s IP address and port, which is used to store the basic information of a node.

Net utils

In the net utils module, we defined several member functions for Peer.

First function is Connect. It takes the target IP address and port as input, trying to establish a TCP connection with the target. If the target address is invalid or cannot be connected, it

calls the Listen function to establish its own p2p network, listen on its own port and wait for incoming connections. If it successfully connects to the target, it will call the askPeerInfo function to ask the target to get a copy of peer information in this network. Then it calls floodJoin function to send a Join Message to all these peers to announce itself. At last it goes into the Listen function and starts to wait for new connections.

Then we have the askPeerInfo function. As mentioned above, this function asks the target for its peers' information. It sends a Message with Message Type “askPeersInfo”. Ideally, It will receive a Message carried with a slice of PeerInfo. It calls recordPeers function to add all these peers’ information into its own Peers record, and avoid repetition.

The Listen function simply binds to its own port and uses a loop to wait for new connections. When there’s an incoming connection, it starts a coroutine and calls HandleConnection function to process it.

The HandleConnection function is called on the receiver side. It will read and unmarshal the Message object. Make actions or prepare replies based on received Message Type and Message Content. If the Message Type is “askPeersInfo”, it will marshal its own Peer slice in Message Content in response. If the Message Type is “joinMessage”, it will update its own Peers and add this new node into its own peers info slice. If the Message type is “Transaction”, it will call the Transact function to make this transaction locally.

Then we have FloodMessage function, which takes Message as input, and sends this message to all of its peers stored in Peers.

floodJoin and FloodTransaction prepares corresponding Message and calls FloodMessage to flood them.

Test module

At last, we have a test module.

log function is used to print out peer information and some intermediate status , which is helpful to debug this program.

The MakeRandomTransaction function takes an integer “num” as input, and make “num” random transactions on p, which means these transactions have random sender, receiver and random amount. Specific test cases are designed in the main function and will be explained later.

- Test your system and describe in the report how you tested it. You are strongly encouraged to make an automatic test where you use a Go-program or script to do the testing. In some of the upcoming exercises this will be a requirement. Later the system will become so complicated, and therefore error prone, that it will be extremely time consuming to develop without automatic testing. The overwhelming feedback from students having done the series of practical exercises is that they wish they had started doing automatic testing from the first exercise.

We start n = 10 peers in a network and each peer sends τ = 10 transactions which relate to same 5 accounts from “Account_1” to “Account_5”

This is how we tested it:
1. We form a network by starting peers one by one. Eventually all peers hold the same set of peers.
2. Each peer starts sending transactions at the same time
3. When the transactions are all over, the result will be printed on the terminal and all peers hold identical ledgers and each account has the same balance at all peers.

- For the report: If you did the simple flooding solution, give a scenario of use which leads to eventual consistency and give a scenario of use which does not lead to eventual consistency. For instance, what happens if all peers join the network one by one before any messages are sent?

We did the simple flooding solution so if all peers join the network one by one before any messages are sent, that is, a transaction happens after the network has been already built, this scenario will lead to eventual consistency. Once the network has been built and transactions happen, if more peers join the network, then the new peers would be inconsistent with the old peers because they lack the information of previous transactions.

- For the report: Assume we made the following change to the system: When a transaction arrives, it is rejected if executing the transaction would make the From account go below 0. Does your system still have eventual consistency? Why or why not?

In this scenario, the system won’t have eventual consistency because the transaction happens concurrently at all the peers which means the sequence of transactions at each peer may not be the same that leads to inconsistency.
For example, given

“Account_1” of P1 has amount of 20
“Account_1” of P2 has amount of 20

There are 2 transactions,

T1: transfer from “Account_1” to “Account_2” with the amount of 15
T2: transfer from “Account_1” to “Account_3” with the amount of 10

If P1 executes T1 and then T2, then “Account_1” of P1 would be 5 (T2 is rejected)
If P2 executes T2 and then T1, then “Account_1” of P2 would be 10 (T1 is rejected)