Handin 2 - Disys

By Helena Cooper (201906086), Timmi Andersen (202105859) and Thea Hende (202105228).

Exercise 4.6 - Implementing a Peer-to-Peer ledger

Description of how we implemented the system

In our system we have defined three different kinds of structs, the `Peer`, `ConnectedPeer` and `MessageStruct` each serving different purposes.

First the `Peer` struct is our way of representing a Peer in a Peer-to-Peer network, and it contains all relevant information. First off this is a `Name`, an `Ip` and a `Port` used to identify the peer. Furthermore it has a `Ledger`, and a set containing all Peers the respective Peer is connected to, `Peers`. Therefore if we were to draw the network, each node would be represented by an instance of the `Peer` struct.
Next up the `ConnectedPeer` struct represents a connection made to a peer. This struct is therefore the type that is held in an array in each Peer's `Peers` field. Note that we could have chosen not to have this struct since it is very similar to the `Peer` struct. However we have chosen to include this to distinguish between `Peers` in the network and the peers a peer has made a connection to. In this way we also do not have to worry about the set of connected peers in the `ConnectedPeer` struct.
Finally we have the `MessageStruct` struct which represents a message sent between peers over the network. Therefore this contains a `Message` describing either a request or response from a peer, and some `Data` that the peer may use depending on the message.

Next we can take a look on the `Connect`, `serverHandling` and `clientHandling` methods. As a peer has the functionality of both a client and a server, we have separated this functionality into two methods, each handling one of the cases. When a peer starts up, in our case it is called `Connect`, we should therefore both do client and server stuff. Here we spawn one thread for server functionality and one for client functionality such that they can run concurrently.
In `clientHandling` we do client stuff, which is first to attempt to make a connection to a given `port`. Depending on whether or not a connection was made succesfully we print a message of what happened. If we have a connection to another peer we request it to send us its set of connected peers using a `MessageStruct` containing the message `GetConnectedPeersRequest`. The sending of requests over the network is handled by the `sendRequest` method. At last we can receive responses on our connection using the `handleResponse` function.

In `serverHandling` we do server stuff, which first is to start listening on a random port. Next we make sure that the peer we are working with is in its own set of `Peers`. This is to make sure that whenever we are asked to pass our set of peers to another peer that peer can add us to its own set. Then we run an infinite loop, where we can accept connections, and if we get

connections we can handle requests coming from that connection through the `handleRequest` helper function.

To handle requests and responses we have implemented a total of four helper functions, namely `sendRequest`, `sendResponse`, `handleRequest` and `handleResponse`. In `sendRequest` and `sendResponse` we send either a request or response over the network by using `gob` and an `encoder`.
Furthermore `handleRequest` and `handleResponse` decodes a message sent with `gob` and reacts according to the `Message` string held in the `MessageStruct`.
In case of `handleRequest` we have three different possible requests. First `GetConnectedPeersRequest` handles the case where a newly connected peer asks for the set of peers from the peer already in the network. Here we create a new `MessageStruct`, convert all `ConnectedPeers` held in the current peer to an array of strings (using the `prepareConnectedPeerToSend` helper function), and sends this as a reponse.
Next "joinRequest" handles the case where a newly connected peer tells another peer that it has connected to the network. Here we create a `ConnectedPeer` from the `Data` array using the `makingConnectedPeer` helper function, and then adds this to peer's map of peers, `Peers`.
At last `MakeTransactionRequest` handles the case where a `Transaction` was sent to a peer. Here we first create a `Transaction` from the information held in `Data` and calls the `Transact` method on the peer´s ledger.
In case of `handleResponse` we currently have one type of response we can handle, namely `GetConnectedPeersResponse`. This handles the case, where a peer sends it set of peers to a newly connected peer in the network. Here we run through the `Data` array and unfold each set of three strings in the array to one `ConnectedPeer`, and we add this to newly connected peer´s set of peers. Because we assume the peer we connected to has a connection to all other peers, then our current peer should now be connected to all other peers as well. At last we prepare a message to tell all other peers that we joined the network, through the `"JoinRequest"`. This message is then flooded across the network.

At last the `Peer` can access two other functions, namely `FloodMessage` and `FloodTransaction`. In `FloodMessage` a peer can flood a message to all other peers in the network, which is done by iterating through the set of connected peers and making a temporary connection in where we send the message. Furthermore in `FloodTransaction` we can flood a transaction to all peers in the network by making a `MessageStruct` where we input the information from the `Transaction` into the `Data` array.

Description of how we tested the system

To test the system we have made the `handin.go` file as asked, where we first create 10 peers, `p1` through `p10`. When the peers are created they are equipped with a ledger that contains accounts `account1` to `account5` all initialized with balance 0. Afterwards we connect all peers to each other by calling `pi.Connect` with the `addr` and `port` of the previous peer, `p(i-1)`.

We sleep in between the calls to `Connect` to allow the execution of previous function calls. After all peers are connected to each other we perform `sendTransactionTest` where we make each peer send ten transactions each associated with all accounts. In the first four transactions we send 1 kr from `account(i)` to `account(i+1)`. Because all peers floods this transaction, the balance after this should be `-10` for `account1` and `10` for `account5` while all others are 0. In the four following transactions we send 2 kr from `account(i+1)` to `account(i)`. Therefore `account5` loses 20 kr while `account1` gains 20 kr after this, resulting in a balance of 10 for `account1` and -10 for `account5` while all others are 0. In the last two transactions we send 3 kr from `account2` to `account3` and then 3 kr from `account3` to `account4`. In this way `account2` should end up with -30 kr, `account4` should have 30 kr and `account3` should have 0 kr. In the end we should therefore have:

- `account1` : 10 kr
- `account2` : -30 kr
- `account1` : 0 kr
- `account1` : 30 kr
- `account1` : -10 kr

At last we print whether or not balance of the accounts of each peer's ledger has the expected value.

Eventual consistency

Right now our network has eventual consistency because we allow all transaction to go through. This means that all peers will eventually hear all messages resulting in every peers ledger being identical. If a transaction were to be rejected due to a account going below zero it would cause problems for the system. We will prove this with an example:

Peer 1 receives a transaction to move all money from account 2 to account 3 and performs the re
Peer 2 receives a transaction to move all money from account 2 to account 5 and performs the re

Now peer 1 also receives a transaction to move all money from account 2 to account 5 which is re
Now peer 2 also receives a transaction to move all money from account 2 to account 3 which is re

Peer 1 receives a transaction to move all money from account 2 to account 3 and performs th
e requestPeer 2 receives a transaction to move all money from account 2 to account 5 and pe
rforms the requestNow peer 1 also receives a transaction to move all money from account 2
to account 5 which is rejectedNow peer 2 also receives a transaction to move all money fro
m account 2 to account 3 which is rejected
Here it is obvious that peer 1 and peer 2 will have different ledgers because they receive

conflicting messages at the same time. If this doesn't happen then the network should have eventual consistency.

Exercise 5.1 - One-time pad theory

Question 1

In this case the employees salary will be represented with maximum 20 bits assuming he earns less than 1 million kroner. He then knows he will never get a 1 on the most significant bit because this would mean he would get an extra million in salary. If he knows that the left-most bit will be 0 on the original message he can predict the secret-key depending on the right-most bit. Say the number is:

01111010000100011111 1

01111010000100011111 1
He should change the right-most number to 1 so that it will be XOR'ed to a 1. Say the number is:

11111010000100011111 1

11111010000100011111 1
He should change the right-most number to 0 so that it will be XOR'ed to a 1.

Question 2

The above problem is not a confidentiality problem but an authenticity problem. The issue is that the employee is able to tamper with the ciphertext which will result in a better salary for him. He should not have access to this encryption in the first place. It's not a confidentiality problem because he is not

Question 3

As stated we have an authenticity problem and not a confidentiality problem. If done correctly one-time pad is considered unbreakable. The message cannot be deciphered unless you have the secret-key. In this case the adversary already has information about his salary which allows him to easily decipher the key and hence control the decrypted message. This is in reality not a problem because he wouldn't be able to do the same thing for any others salary. He is only succesful because he knows his own salary. Even though he can manipulate the most significant bit succesfully he cannot use the information ever again because one-time pad keys are random and discarded after one use. He would also always need to know information about the original message. One-time pad insures confidentiality and not authenticity.

Question 4

Suppose we know that $m_i = 0$mi = 0 with probability $p = 0$p = 0 which would mean that $m_i = 1$mi = 1. When encoding the message we get:

$$c = k \oplus 1$$

c = k ⊕ 1

If we want to make sure that we get a 0 instead when decoding we should flip the bit to retrieve:

$$c' = k \oplus 0$$

c′ = k ⊕ 0

which will be decrypted to $m_i' = 0$mi′ = 0. This is because our key will always retreive our original message from the ciphertext. If we want to make sure that $m_i \neq m_i'$mi = mi′ we must flip the bit to get the opposite bit.

Suppose we know that $m_i = 0$mi = 0 with probability $p = 1$p = 1 which would mean that $m_i = 0$mi = 0. When encoding the message we get:

$$c = k \oplus 0$$

c = k ⊕ 0

If we want to make sure that we get a 0 again when decoding we should leave the ciphertext as it is, because our key will always retreive our original message from the ciphertext.

With this information we should flip $m_i$mi depending on the probability pp. If $0.5 \leq p$ 0.5 ≤ p we should keep the bit as it is. If $0.5 > p$0.5 > p we should flip the bit. This way the adversary will be able to make the receiver obtain a 0-bit in position i with probability max(p, 1 − p) by making a decision depending on which probability is higher.