Team Rap: Rasmus Jørgensen (201909451), Mathias Weller (201907948) & Anna Blume (201907691)

Hand-in 2 (DISSY)

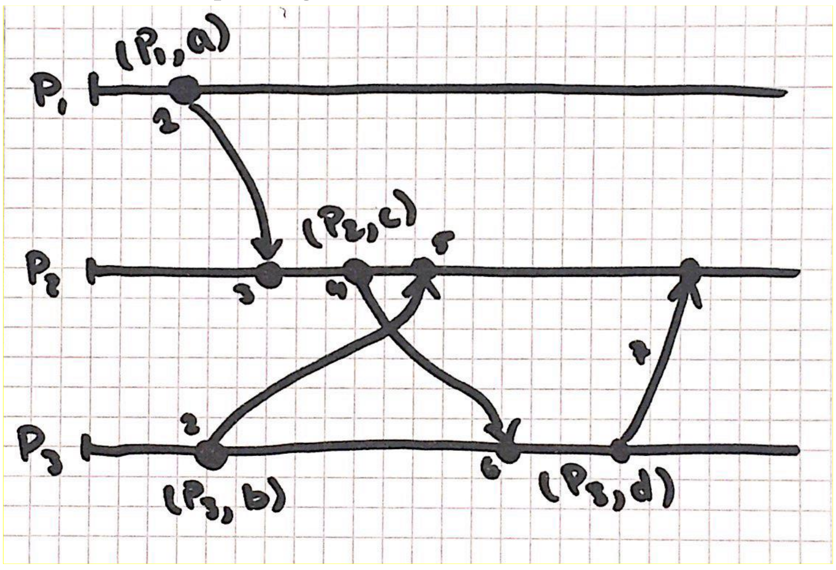
<u>4.3:</u>

We are looking at the following sequence of events (from example 4.3):

1. P1 sends message a (send-event (P1; a) happens at P1).
2. P3 sends message b (send-event (P3; b) happens at P3).
3. Message (P1; a) arrives at P2 (receive-event (P1; a) happens at P2).
4. P2 sends message c to P3.
5. Message (P3; b) arrives at P2.
6. Message (P2; c) arrives at P3.
7. P3 sends message d, which arrives at P2.

Specify the vector clocks that accompany every message and for every pair of messages determine if the messages in the pair are concurrent or in a causal relation.

First, let us set up a diagram to have a better view of the situation:



Now we can look at each message's vector clock.

a: (1,0,0). No other message has been sent by the system yet.

b: (0,0,1). No message has been sent by P3 before or arrived at P3 yet.

c: (1,1,0). Only (a) has arrived at P2 so far, so we include (a)'s causal past.

d: (1,1,2). (c) has arrived at P3, carrying itself and (a) in its causal past, so they are included. Also (b) was previously sent from here.

Finally look at the pairs of messages:

(a,b): Concurrent since neither $VC(a) \leq VC(b)$ or $VC(b) \leq VC(a)$

(a,c): (a) is in $CausalPast(c)$ since $VC(a) \leq VC(c)$

(a,d): (a) is in $CausalPast(d)$ since $VC(a) \leq VC(d)$

(b,c): Concurrent since neither $VC(b) \leq VC(c)$ or $VC(c) \leq VC(b)$

(b,d): (b) is in $CausalPast(d)$ since $VC(b) \leq VC(d)$

(c,d): (c) is in $CausalPast(d)$ since $VC(c) \leq VC(d)$

Team Rap: Rasmus Jørgensen (201909451), Mathias Weller (201907948) & Anna Blume (201907691)

4.6:

1. Test your system and describe how you tested it.

The small methods have been tested by automatic testing (peer_test.go and account_test.go). We have performed manual testing on the whole system by connecting a few peers, checking that their connectionsURI lists are the same, and sending some transactions via the command line and checking that the ledgers are the same by printing them (primitive, but it works). We made automated testing to test for networks larger than 10, which we also used to confirm that a peer indeed only connects to the last 10 entries in the connectionsURI list.

2. Discuss whether connection to the previous ten peers is a good strategy with respect to connectivity. In particular, if the network has 1000 peers, how many connections need to break to partition the network?

It is quite hard to partition the network if there are only a few peers in the network, because most peers will be connected to most other peers. When connecting 1000 peers together, each peer will be connected to at most 20 other peers, and we would be able to partition the network by breaking the 10 connections which the last peer to enter the network has formed. Also, it would be quite easy to do, since the adversary would be able to enter the network with 10 computers one after another, and then the next peer to enter the network would only be connected to those 10 peers. Thus, it might be a better strategy to connect to 10 random peers, which would make it harder for the adversary determine how to partition the network.

3. Argue that your system has eventual consistency if all parties are correct, and the system is run in two-phase mode.

The system is run in two-phase mode, so we first must argue that all peers end up having the same list of peers that are in the system (the list connectionsURI in our system) and then that the ledgers will end up agreeing.

All peers have the same connectionsURI list: Whenever a peer joins it receives a list of URIs (or starts a new one if it is the first in the network). Then it adds itself to the list and broadcasts to everyone that it is now a part of the network. This broadcast will be propagated through the network (like the string messages in the last hand-in) and reach everyone. They will also add the new peer to their list of URIs. We know that all peers join with enough time between them that there are no conflicts in the indices of the list. So, everyone ends up with the same list because everyone gets the same peers added to their list one at a time, and no peer can join the network without everyone else hearing about it.

All ledgers end up in the same state: This is the same argument as for all messages eventually arriving at all peers in the last hand-in. Since we are just adding and subtracting numbers from the ledgers, it does not matter in what order we do it. We have made a choice in the implementation, that any account that a ledger has not seen before will be created and its balance set to 0. This means that if all transactions arrive at all peers their ledgers will end up agreeing, since they all end up having the same accounts and the same transactions happening on those accounts. All peers will get any new transaction, because any new message in the system is broadcast (the network is a flooding network), exactly as the string messages in the last hand-in.

Team Rap: Rasmus Jørgensen (201909451), Mathias Weller (201907948) & Anna Blume (201907691)

4. Assume we made the following change to the system: When a transaction arrives, it is rejected if the sending account goes below 0. Does your system still have eventual consistency? Why or why not?

This system will not have eventual consistency, which we can show by an example. Say acc1 has a balance of 100. Two transactions are in the system. T1 wants to take 200 from acc1. T2 wants to put 200 in Acc1. At P1 T1 arrives first and is rejected, so Acc1 still has 100 in it. Then T2 arrives Acc1 has a balance of 300. At P2 T2 arrives first and Acc1 now has 300 in it. Then T1 arrives, is not rejected, and Acc1 ends by having 100 in it. Now P1 and P2 are not in agreement on Acc1, and the system is not consistent.