Assignment 2

Exercise 4.3:

Recall the series of events we considered in Example 4.3

Assume: Players use the protocol from theorem 4.7 based on vector clocks to communicate. For each send-event, specify the vector clock that accompanies the message that is sent. There are 4 messages sent, and hence 6 pairs of messages.

Task: For each such pair, use the vector clocks associated to the messages to determine if the messages in the pair are concurrent, or the pair is in the causal past relation.

First, one must recall the theorem given on page 111 of the DNO book, presented below:

Theorem 4.7 FIFO2Causal + FIFO $\sqsupseteq$ Causal.

Notice that once all messages have been tagged with vector clocks, we can efficiently check if there are in each others causal pasts or whether they are concurrent. It is easy to see that

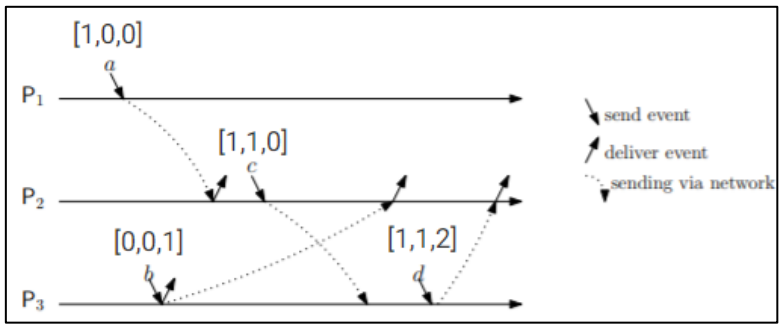
$$(\mathsf{P}_i, m_i) \hookrightarrow (\mathsf{P}_j, m_j) \iff \mathsf{VectorClock}(\mathsf{P}_i, m_i) \leq \mathsf{VectorClock}(\mathsf{P}_j, m_j) \ .$$

This gives an efficient implementation of causality testing. We simply keep track of the vector clocks, and when we want to know whether $(\mathsf{P}_i, m_i) \hookrightarrow (\mathsf{P}_j, m_j)$ we simply check whether $\mathsf{VectorClock}(\mathsf{P}_i, m_i) \leq \mathsf{VectorClock}(\mathsf{P}_j, m_j)$.

When we have two message (P_i, m_i) and (P_j, m_j) for which $\mathsf{VectorClock}(\mathsf{P}_i, m_i) \not\leq \mathsf{VectorClock}(\mathsf{P}_j, m_j)$ and $\mathsf{VectorClock}(\mathsf{P}_j, m_j) \not\leq \mathsf{VectorClock}(\mathsf{P}_i, m_i)$, we say that the messages are concurrent. Neither of these messages have affected each other. When two messages are concurrent we write

$$(\mathsf{P}_j, m_j) \| (\mathsf{P}_i, m_i) \ .$$

Second, one can write the vector clocks on figure 4.11, given below:



Third, based on this figure, one can deduce the following relations, based on theorem 4.17:

$$\{m_a, m_c, m_d\} \text{ and } \{m_b, m_d\}$$

$$m_a \rightarrow m_c \rightarrow m_d \quad [1,0,0] < [1,1,0] < [1,1,2]$$

$$m_b \rightarrow m_d \quad [0,0,1] < [1,1,2]$$

Thus, the causal order can be:

$$\{m_a, m_c, m_b, m_d\} \text{ or } \{m_a, m_b, m_c, m_d\} \text{ or } \{m_b, m_a, m_c, m_d\}$$

Fourth, determine the lexicographical order, which would be:

By: Deyana Atanasova, Henrik Tambo Buhl & Alexander Stæhr Johansen

$$\{m_b, m_a, m_c, m_d\}$$

Fifth, determine the concurrent message pairs.

By inspecting the flow of *process 1 message a* and *process 3 message b*, one can write the following relation in process notation:

$$(P1, a) \nRightarrow (P3, b) \wedge (P3, b) \nRightarrow (P1, a)$$

Giving, in vector clock notation:

$$[1,0,0] </ [0,0,1] \text{ and } [0,0,1] </ [1,0,0]$$

Therefore, *process 1 message a* and *process 3 message* b ARE concurrent.

By inspecting *process 1 message a* and *process 2 message c*, one can observe the following relation:

$$(\mathrm{P1}, \mathrm{a}) \rightarrow (\mathrm{P2}, \mathrm{c})$$

Giving:

$$[1,0,0] < [1,1,0]$$

Therefore, *process 1 message a* and *process 2 message* c are <u>NOT</u> concurrent.

And, by inspecting *process 1 message a* and *process 3 message d*, one can observe:

$$(\mathrm{P1}, \mathrm{a}) \rightarrow (\mathrm{P3}, \mathrm{d})$$

Giving:

$$[1,0,0] < [1,1,2]$$

Therefore, *process 1 message a* and *process 2 message* c are <u>NOT</u> concurrent.

Then, by looking at *process 2 message c* and *process 3 message b*, one can see that:

$$(\mathrm{P2}, \mathrm{c}) \nRightarrow (\mathrm{P3}, \mathrm{b}) \wedge (\mathrm{P3}, \mathrm{b}) \nRightarrow (\mathrm{P2}, \mathrm{c})$$

$$[1,1,0] </ [0,0,1] \text{ and } [0,0,1] </ [1,1,0]$$

Therefore, *process 2 message c* and *process 3 message b* ARE concurrent.

Then, by looking at the pairs of *process 2 and message c* and *process 3 and message d*, one can see:

$$(P2, c) \rightarrow (P3, d)$$

$$[1,1,0] < [1,1,2]$$

Therefore, *process 2 message c* and *process 3 message b* are <u>NOT</u> concurrent.

At last, by looking at process 3 message b and process 3 message d one can see:

$$(P3, b) \rightarrow (P3, d)$$

$$[0,0,1] < [1,1,2]$$

Therefore, *process 3 message b* and *process 3 message d* are <u>NOT</u> concurrent.

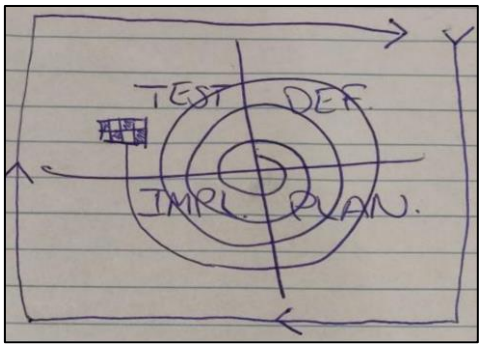
By: Deyana Atanasova, Henrik Tambo Buhl & Alexander Stæhr Johansen

Exercise 4.6:

Exercise 4.6.1:

Test you system and describe how you tested it.

To implement and test the solution, the incremental software development process was applied using the spiral approach, sketched below:



First, the requirements specification was analyzed thoroughly and the overall system was defined. Second, an overall plan was outlined of how to structure and implement the requirements specified in testable modules. Having made the plan, the process began as the group, first, defined the first module to implement. Second, planned how to implement it. Third, implemented it, and, fourth, tested it. To test a module its functionality was simply evaluated against the requirement it was sought to implement. Thus, if the module implemented the requirements defined, the module was considered successful. Once a module was implemented, the next iteration began of 1) define module, 2) plan implementation, 3) implement and 4) test. This was iterated until product of this project was realized, which can be found in the folder associated with this .pdf.

Exercise 4.6.2:

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?

This depends on the number of peers on the network.

In the case that only 10 peers are on the network, the strategy of connecting to $P - 10$ peers is excellent as it is perfectly fault tolerant.

However, in a setting where 1000 peers are on the network, this provides a fault tolerance of $f = \frac{m}{n} = \frac{10}{1000} \cdot 100 = 1\%$. Whether this is acceptable is a design decision which needs to be made with respect to the application of the network.

Exercise 4.6.3:

Assuming the two-phase startup the system has eventual consistency due to how the broadcast mechanism works.

Whenever a transaction is made, it is broadcast by the peer that makes it to itself and the other 10 peers it is connected to. This is then broadcast by these 10 peers to the next 10 peers they are connected to, and so on, ensuring all nodes on the network have received the transaction, thus ensuring eventual consistency. As implicitly declared, this means every peer receives a transaction multiple times, but as every peer stores the transaction it has processed locally, it is able to recognize

By: Deyana Atanasova, Henrik Tambo Buhl & Alexander Stæhr Johansen

when a duplicate transaction is received, thus it stops from broadcasting it and prevents flooding cycles.

Exercise 4.6.4:

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?

Assuming that a transaction is rejected if the account balance goes under 0, the system is no longer eventually consistent, if the network is large enough. The reason is that two transactions can be made, before the previous is processed.

A perfect example of this flaw is the US check system that was exploited by Mr. Frank Abagnale Junior.

By: Deyana Atanasova, Henrik Tambo Buhl & Alexander Stæhr Johansen