Distributed Algorithms - Exercise
Winter term 2017/18

Danh Le Phuoc
Qian Liu
ODS

Exercise sheet 3

Group 8:
Sebastian Schasse
Kosta Ivancevic

Exercise 3.1: Vector Clocks

Questions

Causal Order

The concept of the "causal broadcast" has been introduced in the lecture as an application of vector clocks. In order to guarantee the proper execution of the algorithm it was necessary to send each message to all participating nodes.
If we want to avoid sending each message to all nodes, why wouldn't it be sufficient to only use the vector as applied by the causal broadcast to achieve causal order?

In the "causal broadcast", each message shall be delivered to all processes, with the order of message deliveries satisfying causality.
Each process P_i increments the V[i] component of a vector clock if it sends a message.
For the message originating from P_i to P_j to be delivered, the following constraints are to be followed:

- $T[i] = T_j[i]+1$
- $\forall k \neq i: T[k] \leq V_j[k]$

First constraint prevents that messages originating from the same source get delivered in the order different from which they are sent. The second constraint prevents delayed messages from other sources to be delivered out-of-the order. Example is shown below.
On Figure 1, following scenario can be observed: the requirement that, each message is not delivered to all nodes, is not satisfied. A process P4 can be ignored, and, after message exchange, its internal vector isn't updated along with the other processes. This leads to following problems:

- Receive problem: At message from P1 delivery point: doesn't satisfy $\forall k \neq i: T[k] \leq V_j[k]$ because internal vector V_j wasn't updated accordingly.
- Send problem: When trying to send a message to P_2: can't generate sufficiently large vector which would satisfy $\forall k \neq i: T[k] \leq V_j[k]$ on the receiving process(P_2), as a direct result of being outdated.

To summarize, if a process is skipped once in the message reception, in future, it can never satisfy the requirement:

- $\forall k \neq i: T[k] \leq V_j[k]$

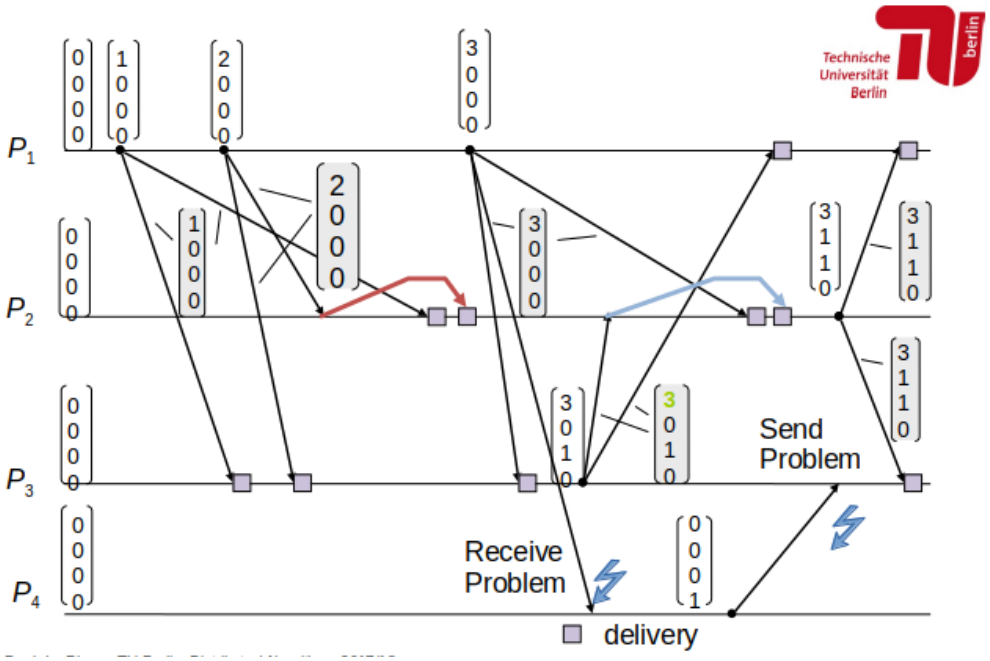

Figure 1: Illustration of relaxed requirement (Each process doesn't have to send a message to all other processes)

Exercise 3.2: Snapshot

Questions

Give an example to show that the Chandy-Lamport algorithm is flawed if channels are not FIFO.

We can divide a set of messages into:

- Application messages – relevant to current application
- Snapshot markers – indicators of a snapshot request

As Figure 2 shows, when FIFO order is not satisfied, there can be message overtaking in the channel. If such event happens, initiator process (P_2) assumes that an effect of the first message is included in the snapshot. On the other hand, receiving process (P_1) first receives a snapshot message, and, not knowingly about the departure of the application message, makes a snapshot without including the effect which is caused by an application message. Such scenario can be seen as having inconsistent cut, illustrated on Figure 3.

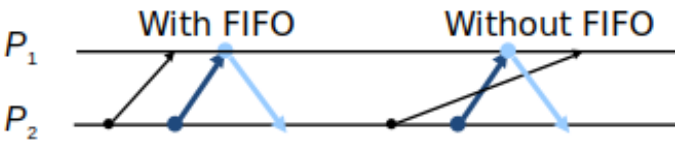

Figure 2: Comparison of situations with and without FIFO relation

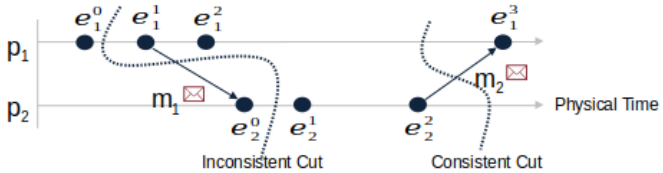

Figure 3: Comparison between consistent and inconsistent cut

Propose an adaptation of the Chandy-Lamport algorithm, in which basic messages may be buffered at the receiving processes, and the channel states of the snapshot are always empty.

Lamport clocks can be used in this purpose. Each message can have appended time stamp, based on which the process knows if the message is from the corresponding side of the Cut. In this manner, each message can be classified as “pre-snapshot” or “post-snapshot”.

Give(demonstrate your solution with text, pseudo-code, and diagrams) a snapshot algorithm for undirected networks with non-FIFO channels that uses:

1) **marker messages, tagged with the number of basic messages sent into a channel before the marker message**
2) **acknowledgments, and**
3) **temporary (local) freezing of the basic execution**

Each process counts the number of sent and received messages between two snapshot markers in each channel. Each message is appended with a number, which indicates the order in which it was sent. On the receiving side, events are executed in the ascending order of their numbers.
Freezing is needed to achieve causality between reordered messages, corresponding to each channel.

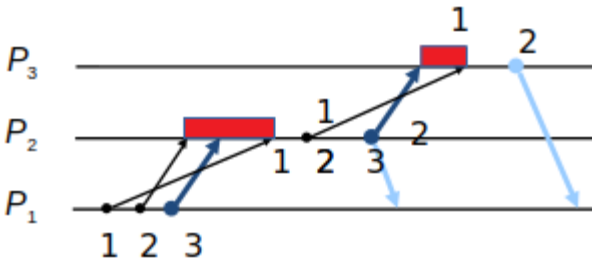

Figure 4: Algorithm for non-FIFO channels

When the process receives a marker message, it will take a snapshot after all “presnapshot” messages are executed. Afterwards, the process sends an acknowledgement to its initiator, and propagates the marker further to all processes.

Give an example in which the Lai-Yang algorithm computes a snapshot that is not a configuration of the ongoing execution.

Since the Lai-Yang algorithm doesn’t require control messages (snapshot messages). If a process receives snapshot message, it will append the information(lecture: becomes RED) to all outgoing messages, and that is how the snapshot message will spread from the initiator to all other nodes. It does not ensure that all processes will make a snapshot. As Figure 5 shows, in a situation where a process doesn’t receive messages from processes who received their snapshot messages, it doesn’t get the request to take a snapshot.

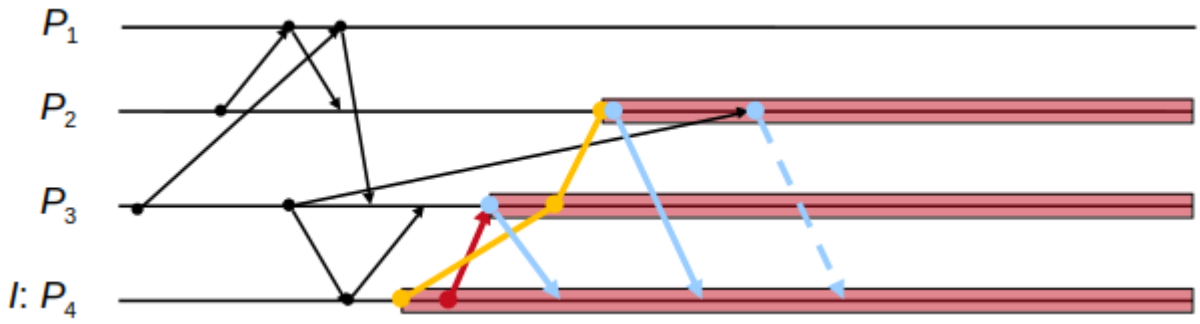

Figure 5: Lai and Yang algorithm

The Lai-Yang algorithm doesn't require FIFO channels. By having a situation in which FIFO relation is not sufficed, results in message overtaking. That can lead to the outdated data. Recall Figure 2. If an snapshot marker overtake application message on the channel, inconsistent cut will be made. Thus results in the snapshot initiator expecting the effect of the sent message to result in snapshot receivers configuration. On the other hand, both application message and snapshot message are delivered in reversed order, so receiving process makes a snapshot without including the effect of the message, which it will receive in the future.

Exercise 3.3: Fault Tolerance

Questions

Buffer overflows(also known as pointer or heap or stack smashing) are a well-known source of programme failure. What kind of fault models would capture their effects on the rest of the system.

The fault model would be dependent on how good buffer overflow is handled and how significant a certain buffer is to control path is. Good hardware and software co-implementation will indicate the overflow by setting indicator flags. We can distinguish two different scenarios, purely based on the importance of the buffer.

- Best case: Omission fault – some actions are not performed. Certain implementations are based on "data_ready-data_valid" flags. Data_valid is indicator that receiver (buffer) is ready to receive data, while data-valid is the indicator that data on the bus is validly set by the data provider. In some implementations, keeping data_ready on "0" will result in throughput degradation, and will stop failure propagation.
- Worst case : Crash fault – a node is not executing any valid actions. This is a result of buffer overflow on some very important buffers(PC,SP,IP,...) where state transition is not handled. The errorneous behaviour is quickly spread through the system, which results in total malfunction.

Consider the following real-life failure scenarios and examine if these can be mapped to any of the known fault classes introduced in the lecture:

1) **On January 15 1990, 114 switching nodes of the long-distance system of AT&T went down. A bug in the failure recovery code of the switches was responsible for this. Ordinarily, when a node crashed, it sent out-of-service message to the neighboring nodes, prompting the neighbors to reroute traffic around it. However, the bug (a misplaced break statement in C code) caused the neighboring nodes to crash themselves upon receiving the out-of-service**

message. This further propagated the fault by sending an out-of-service message to nodes further out in the network. The crash affected the service of an estimated 60,000 people for 9h, causing AT&T to lose $60 million revenue.

http://users.csc.calpoly.edu/~jdalbey/SWE/Papers/att_collapse.html
This failure can be classified as Byzantine / Malicious because the error, which caused one server to go to fault state, was propagated to other neighbors. Within short period of time, malfunction was spread to neighboring computer-operating switching centers, ultimately reaching 114 switches in the network.
If we consider only a single switch, we can classify the failure as crash, but direct result of the error propagation classifies this into Byzantine / Malicious failure.

2) **A programme module of the Arianne space shuttle received a numerical value that it was not equipped to handle. The resulting variable overflow caused the shuttle's on board computer to fail. The rocket went out of control and subsequently crashed.**

This fault can be classified as a crash failure if we consider the rocket as a node. As defined, a crash fault results in a whole node failure, after which it is not capable of performing any actions (the rocket ultimately crashed)
On the other hand, this can be classified as an omission fault because inconsistent numerical values resulted in board computer failure. If the rocket is considered a system, a certain action execution were affected, and the performance, which is not coordinated with board computer were not affected.
Also, due to propagation of erroneous behaviour, this can be classified as Malicious.