TUM I 13
Chair of Application and Middleware Systems
Prof. Dr. H.- A. Jacobsen

Tutorial
Distributed Systems (IN2259)
WS 2020/21

SAMPLE SOLUTION: EXERCISES ON TIME IN DISTRIBUTED SYSTEMS

Note: this sample solution is only a suggestion to solve the assignment; there might be various other possible solutions. Also note that this sample solution might still contain errors or other fallacies.

EXERCISE 1 Cristian's Algorithm

A client attempts to synchronize its clock with a time server using Cristian's algorithm. Therefore, the client records the round-trip times of requests and the timestamps returned by the server as depicted in Table 1.1.

ROUND-TRIP TIME (ms)	SERVER TIME (hh:mm:ss:fff)
22	10:54:23:000
25	10:54:25:000
20	10:54:28:000

Table 1.1: Round-trip times and timestamps.

(a) Which of these times should the client use to adjust its clock?

Solution: The one with the minimal RTT, which is $\underline{10:54:28}$.

(b) To what time should the client set its clock?

Solution: The client should set it's clock to: $t_C = t_S + \frac{RTT}{2} \Rightarrow t_C = 10:54:28:000 + \frac{20}{2}ms = \underline{10:54:28:010}$.

(c) Estimate the accuracy of the setting with respect to the server's clock.

Solution: The accuracy is calculated by: $acc = \pm(\frac{RTT}{2} - min)$.
As we have no information about min, we consider $min = 0ms$.
$\Rightarrow acc = \pm(\frac{20}{2}ms - 0) = \pm\underline{10ms}$.

(d) If it is known that the time between sending and receiving a message in the system is at least 8 ms, do your answers to the above questions change?

Solution: $acc = \pm(\frac{20}{2}ms - 8ms) = \pm\underline{2ms}$.

(e) Discuss if we can synchronize the client's clock with the time server to within 0 milliseconds, (i.e., fully accurately).

Solution: It is impossible to do that. Fully accurate synchronization of physical clocks among nodes in a distributed system is impossible because the nodes can never obtain current information about the other nodes' clock values. The main reason is that all messages arrive after an unknown and variable delay which cannot be predicted precisely. Even if the message delays were always the same and the nodes knew this value exactly, they still could not synchronize the clocks perfectly because of the variable hardware clock drifts, i.e., a node cannot determine exactly how much another clock progressed since the last message arrived.

EXERCISE 2 Berkeley Algorithm

A collection of five processes (P1 - P5) want to synchronize their clocks according to the Berkeley algorithm. Process P3 has been elected as the master. The threshold value for acceptable deviation $\delta = \pm 3000ms$. For the given round of the algorithm, the times of all individual processes have been collected by the Master (cf. Table 2.2). (*Hint: Different to the original Berkeley algorithm, the master did not record any RTTs in this setting. So for this task, you can omit the calculation of a better estimate for the individual process times and instead work with the times given in the table.*)

PROCESS	PROCESS TIME (hh:mm:ss:fff)
P1	08:44:56:144
P2	08:44:52:874
P3	08:44:53:123
P4	08:44:53:100
P5	08:44:50:996

Table 2.2: Individual process times gathered at the master.

(a) What is the main difference between the Berkeley algorithm compared to Cristian's algorithm?

Solution: The Berkeley algorithm performs an internal synchronization, while Cristian's algorithm performs external synchronization. Hence, the Berkeley algorithm does not require a dedicated time server.

(b) What time will the master process P3 calculate as the reference time for synchronization.

Solution: Process P3 calculates the mean value of the process times that have been reported by P2, P3, P4, and P5. It will not consider the time reported by P1. The new reference time t_{ref} is consequently calculated by:

$$t_{ref} = \frac{08:44:52:874 + 08:44:53:123 + 08:44:53:100 + 08:44:50:996}{4} = \underline{08:44:52:523}$$

(c) What information, i.e., which values will P3 send to the other processes for synchronization? What is the content of the message?

Solution: The master process P3 estimates the clocks of all processes and only send the amount of time (positive or negative) that the individual processes must adjust their clocks for. Hence, the message content is comprised of the skew between the newly calculated reference clock and the clocks of the individual processes.

PROCESS	UPDATE VALUE
P1	08:44:52:523 - 08:44:56:144 = -00:00:03:621
P2	08:44:52:523 - 08:44:52:874 = -00:00:00:351
P3	08:44:52:523 - 08:44:53:123 = -00:00:00:600
P4	08:44:52:523 - 08:44:53:100 = -00:00:00:577
P5	08:44:52:523 - 08:44:50:996 = +00:00:01:527

(d) Give an advantage and a disadvantage for sending, in particular, those kind of values that have been calculated in subtask (c) to synchronize the clocks.

Solution:
Advantage: There is less additional error introduced by another transmission delay between the master the slaves.
Disadvantage: The skew value (update value) has to be calculated for every process in the set individually, which might influence scalability.

EXERCISE 3 Timestamping with logical clocks

Consider the time-event diagram depicted in Figure 3.1 and accomplish the following tasks. Each arrow in the diagram represents a message transmission. For example, a is the sending event and b is the receiving event for a message that is sent by process A to process B.

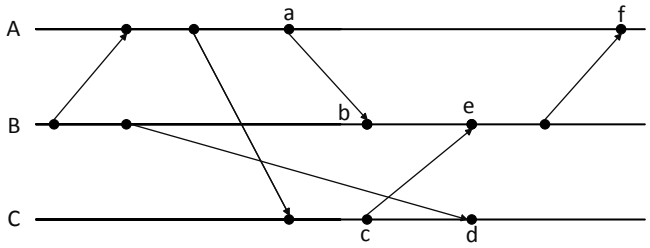
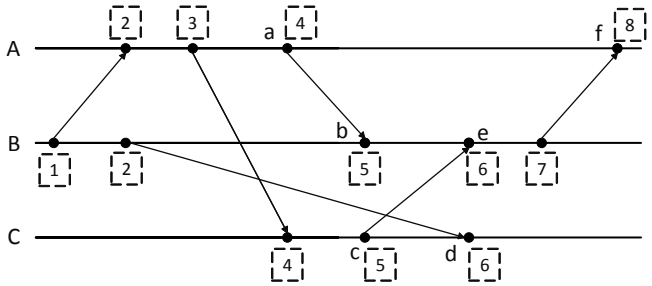


Figure 3.1: Time-event diagram.

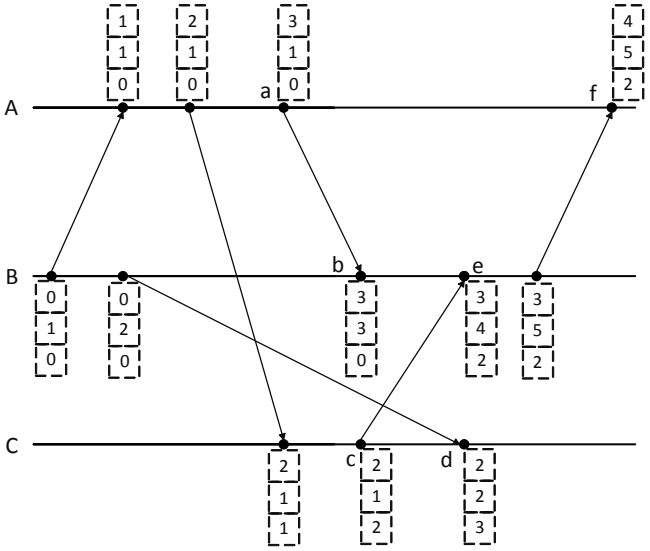
(a) Timestamp each event based on the *Logical Clock* algorithm introduced by Lamport.

Solution:



(b) Timestamp each event based on the *Vector Clock* algorithm.

Solution:



(c) Using only the Vector Clock timestamps calculated in subtask (b) show whether, for each of the below event pairs, the *happened-before* relation ($\rightarrow$) holds or not. Two events that are not in *happened-before* relation are also called *concurrent* ($\|$) events.

Solution: Comparing Vector clocks (if $t_a < t_b$ then $a \rightarrow b$)

$$
\begin{aligned}
t_a = t_b &\iff \forall i:\ t_a[i] = t_b[i] \\
t_a \neq t_b &\iff \exists i:\ t_a[i] \neq t_b[i] \\
t_a \leq t_b &\iff \forall i:\ t_a[i] \leq t_b[i] \\
t_a < t_b &\iff t_a \leq t_b \wedge t_a \neq t_b
\end{aligned}
$$

(i) a, c **Solution**: $V_a = (3,1,0), V_c = (2,1,2)$ Neither $V_a < V_c$, nor $V_a > V_c$ $\Rightarrow$ $\underline{\underline{a \parallel c}}$

(ii) b, d **Solution**: $V_b = (3,3,0), V_d = (2,2,3)$ Neither $V_b < V_d$, nor $V_b > V_d$ $\Rightarrow$ $\underline{\underline{b \parallel d}}$

(iii) c, e **Solution**: $V_c = (2,1,2), V_e = (3,4,2)$ $V_c < V_e$ $\Rightarrow$ $\underline{\underline{c \rightarrow e}}$

(iv) c, f **Solution**: $V_c = (2,1,2), V_f = (4,5,2)$ $V_c < V_f$ $\Rightarrow$ $\underline{\underline{c \rightarrow f}}$

(d) Can we certainly determine the existence of either *happened-before* or *concurrency* relation between a pair of events only by knowing logical clock timestamps?

Solution:

$$
\begin{aligned}
a \rightarrow b &\ \Rightarrow\ L(a) < L(b) \\
L(a) < L(b) &\ \not\Rightarrow\ a \rightarrow b
\end{aligned}
$$

Only by knowing logical clock timestamps, we cannot determine the existence of either *happened-before* or *concurrency* relation between a pair of events. In the depicted diagram L(a) < L(c), which would lead us to the following *happened-before* relation a $\rightarrow$ c. However, as previously showed, a *concurrency* relation (a $\|$ c) was demonstrated using vector clocks.

EXERCISE 4 Atomic total-order broadcast

Assume an asynchronous distributed system without failures that is comprised of n processes: $P = \{p_1, p_2, \ldots, p_n\}$. The only available communication mechanism in this system is broadcast, i.e., if a process sends a messages, this message is delivered to all processes in the system including the sender itself. Now, this system should be provided with a new feature referred to as *atomic total-order* broadcast (*ATO-broadcast*). In *ATO-broadcast* each process p_i ($1 \leq i \leq n$) receives messages in the same order as every other process.

For example, if process p_i receives messages in the order $[m_2, m_1, m_6 \ldots, m_k]$, then all other processes must receive messages in this order. We assume that processes are aware of their *processIds* (i.e., $\{p_1, p_2, \ldots, p_n\}$) and that no message gets lost in the communication between processes. Also, messages from the same sender are received in the order that they were sent (i.e., FIFO channel).

Design an algorithm to provide the system with this feature using *Lamport clocks* and process IDs. **Hint:** Write two pseudo code snippets. One describes the algorithm on the sender side (i.e., function `broadcastATO(msg)`), the other describes the algorithm on the receiver side (i.e., function `onReceiveATO(msg)`). In the pseudo code, you are allowed to use basic data structures like arrays, lists, queues, etc. with a simplified syntax.

Solution:

Assume that each process in the system has a logical clock, which is updated every time a message is sent or received, respectively. Further assume that each process maintains an ordered message queue that caches messages ordered by the tuple (timestamp, processId), (i.e., totally ordered), and also a list of acknowledgement messages from other processes.

Algorithm 1 ATO_Broadcast_Sender

```
1:  procedure BROADCASTATO(msgContent)   ▷ ATO broadcasts the msgContent to all processes in the process group
2:      timestamp ← timestamp + 1                                  ▷ increment Lamport Clock
3:      msg[content] ← msgContent                                  ▷ prepare a new message and set the content
4:      msg[ts] ← timestamp                                        ▷ add Lamport timestamp to message
5:      msg[pid] ← processId                                       ▷ add process identifier to message
6:      broadcast(msg)                                             ▷ send message to all processes (including itself)
7:  end procedure
```

Algorithm 2 ATO_Broadcast_Receiver

```
1:  procedure ONRECEIVEATO(msg)                  ▷ on receive of message over regular broadcast mechanism
2:      if msg is atoMessage then
3:          timestamp ← MAX(timestamp, msg[ts]) + 1                ▷ increment Lamport Clock
4:          queue.add(msg)          ▷ insert message into an ordered queue, i.e. ordered by tuple ⟨ts, pid⟩
5:          ack[content] ← msg                                     ▷ prepare acknowledgement message for the received msg
6:          ack[ts] ← timestamp                                    ▷ add Lamport timestamp to acknowledgement
7:          ack[pid] ← processId                                   ▷ add process identifier to acknowledgement
8:          broadcast(ack)                  ▷ send acknowledgement for msg to all processes (including itself)
9:      else if msg is acknowledgement then                        ▷ differentiate between message types
10:         acks.add(msg)                                          ▷ acknowledgement messages are stored separately
11:     end if
12:     msgToDeliver ← queue.top()                                 ▷ get oldest message from the queue
13:     acknowledged ← true
14:     for p_i ∈ P do                  ▷ For all processes check if the oldest message has been acknowledged
15:         if !acks.contains(acknowledgement for msgToDeliver from p_i) then
16:             acknowledged ← false                               ▷ some process p_i did not yet acknowledge this message
17:         end if
18:     end for
19:     if acknowledged then                                       ▷ all processes acknowledged the receipt of the message
20:         deliver(queue.pop())        ▷ hence msgToDeliver can be popped and delivered to the application
21:     end if
22: end procedure
```

Note that in the above algorithm, since the links are FIFO, any message generated by a process carry higher timestamps than the timestamps of previous messages generated by the process.

EXERCISE 5 Causal Broadcast

Given the following definition and the logical clock algorithms discussed in the lecture, provide an algorithm to implement *Causally-ordered broadcast* (*CO-broadcast*). *CO-broadcast* extends the usual broadcast semantics with the following properties: (Note: $\rightarrow$ is the *happens-before relation*).

(i) Let m and m' be two broadcast messages such that broadcast(m) $\rightarrow$ broadcast(m'), then each process must receive m before m';

(ii) Let m and m' be two broadcast messages such that broadcast(m) $\parallel$ broadcast(m'), then m and m' can be received in different orders by different processes;

Solution:
At each process p_i with $i \in \{1 \ldots n\}$ in the system, initialize a vector clock vc_i with size $|vc_i| = n$ and $vc[j] = 0$ for $j \in \{1, \ldots, n\}$.

Algorithm 3 CO_Broadcast_Sender

```
1: procedure BROADCASTCO(msgContent)        ▷ CO broadcasts the msgContent to all processes in the process group
2:     vc_i[i] ← vc_i[i] + 1                ▷ increment Vector Clock at own index, here i
3:     msg[content] ← msgContent            ▷ prepare a new message and set the content
4:     msg[ts] ← vc_i                       ▷ add Vector Clock timestamp to message
5:     broadcast(msg)                       ▷ send message to all processes (including itself)
6: end procedure
```

Algorithm 4 CO_Broadcast_Receiver

```
1: procedure ONRECEIVECO(msg)               ▷ on receive of message over regular broadcast mechanism
2:     queue.add(msg)                       ▷ insert message into a local hold-back queue
3:     for tmpMsg ∈ queue do
4:         v_j ← tmpMsg[ts]
5:         if (v_j[j] = v_i[j] + 1)  ∧  (v_j[k] ≤ v_i[k] for k ≠ j) then
                                            ▷ p_i has delivered every other message that p_j had broadcasted
                                              before it broadcasted (tmpMsg, v_j)), and p_i has delivered
                                              any other message that p_j had already delivered at the
                                              time it broadcasted the message (tmpMsg, v_j))
6:             deliver(msg)                 ▷ delivered the message to the application
7:             v_i[j] = v_i[j] + 1          ▷ update Vector Clock
8:         end if
9:     end for
10: end procedure
```