CS6550 Advanced Operating Systems

Homework Assignment 4

Question 1 (Synchronizer, Problem 11.10 in textbook):
Consider an array of processes 0, 1, 2, ……, 2*n* − 1 that has a different type of synchrony requirement: We call it *interleaved synchrony*. It is specified as follows:

(1) Neighboring processes should not execute actions simultaneously;
(2) Between two consecutive actions by any process, all its neighbors must execute an action.

When *n* = 4, a sample schedule is as follows:

0, 2, 4, 6, : tick 0
1, 3, 5, 7, : tick 0
0, 2, 4, 6, : tick 1
1, 3, 5, 7, : tick 1

The computation is an infinite execution of such a schedule. Design a synchronizer to implement interleaved synchrony. Consider two different cases: (1) an ABD (Asynchronous Bounded Delay) system and (2) a system without physical clocks.

Solution:

1:
An Asynchronous Bounded Delay
Let's take the ring topology for example.

- The nodes (0,2,4,6) initialize the synchronizer operation.
- They send a initialize message to themselves and send a start signal to the node (1,3,5,7).
- The nodes (1,3,5,7) wake up and complete the initialization.

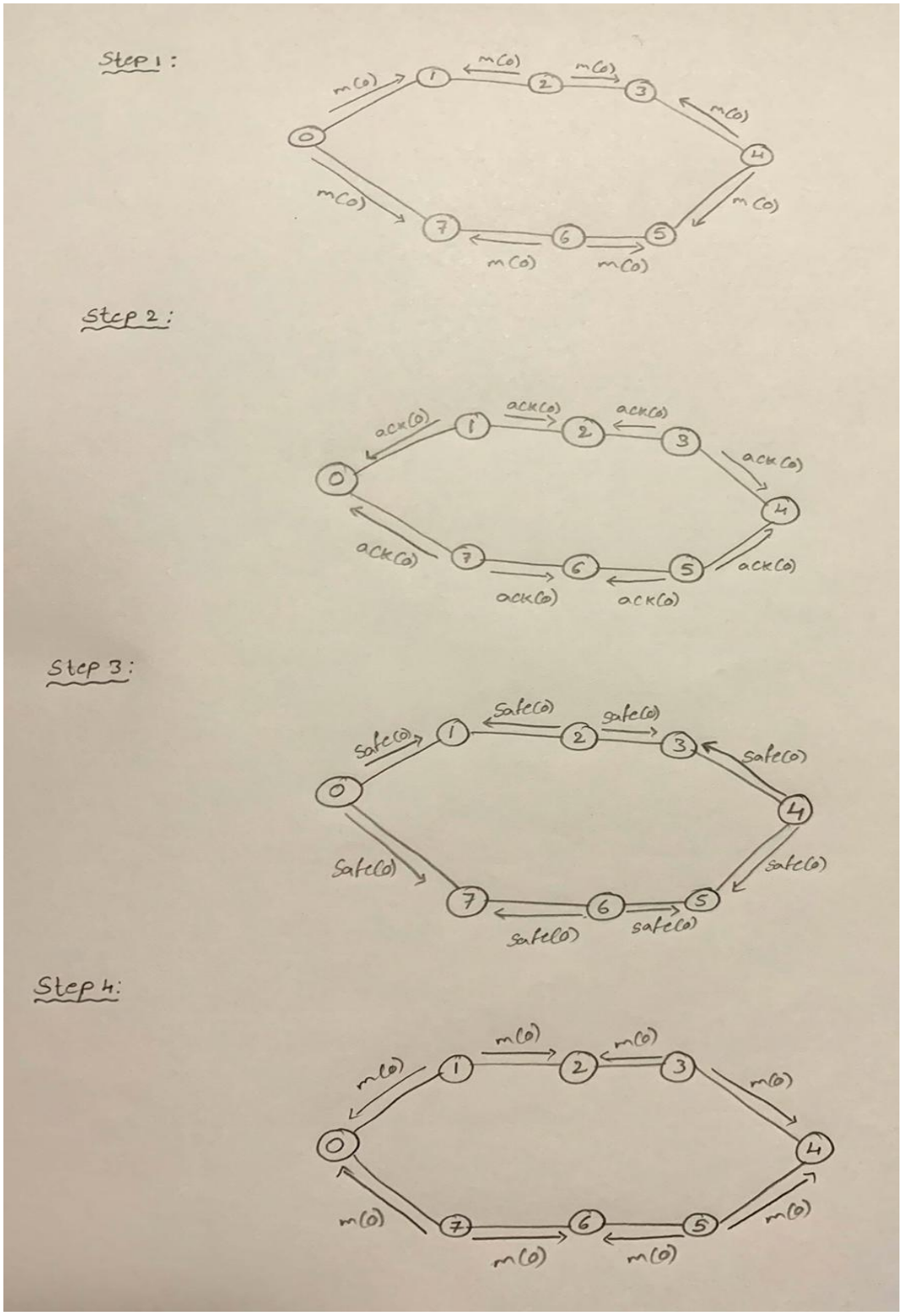
The above steps show the way how the tick 0 is completed. It is how they complete the tick 0. Similarly, then it takes place once again for the tick 1 and the above three steps method is being repeated.

2:
A system without physical clocks

We may use the below algorithm for the clocks to get synchronized.

- Send and receive messages < *m* , *i* > for the current tick *i* ;
- Send < *ack,i* > for each incoming message, and receive < *ack,i* > for each outgoing message for the current tick *i* ;
- Send < *safe,i* > to each neighbor;

The Synchronizer works as follows without any physical clock



The above depicts flow of how the clocks are being synchronized.
The steps can be carried out until all the clocks are being synchronized.

Question 2 (Fault tolerance in communication, Problem 12.9 in textbook):
A sender P sends a sequence of messages to a receiver Q. Each message *m* is tagged with a unique sequence number *seq* that increases monotonically, so the program for P can be specified as follows:

```
program sender
define   seq : integer {message sequence number}
initially seq = 0
do true → send(m[seq],seq); seq := seq + 1 od
```

Messages may reach Q out-of-order, but they are never lost. To accept messages in strictly increasing order of sequence numbers, Q uses a *sequencer* that re-sequences the message delivery.

1. Describe the program for the sequencer. Calculate its buffer requirement.
2. Now assume that the sequencer has the ability to hold *at most two messages* in its buffer. Rewrite the programs of P and the sequencer, so that Q receives the messages in strictly increasing order of sequence numbers.

Solution:
1:
The Program for the sequencer works in the following method.

- Sender Sends its message to receiver
- Receiver initialize the local variable recvCnt=0;
- It also checks the sequence condition if seq=recvCnt;
- Then it accepts the message or else it stores the message in buffer.

SENDER	RECEIVER

Send (msg [seq], seq);
 	
seq = seq+1;
	Initially recvCnt = 0;
	
	If (seq== recvCnt) → Accept the message sent
	recvCnt = recvCnt +1;
	
	Else
	Add sender message to the buffer

The buffer size would be the number of messages in the sender side.
All the messages sent from the sender side are being stored in the buffer and hence the message either received or stored in the buffer, there is no message being lost during the transactions.

Hence by using this above method we can ensures that the receiver Q strictly accepts the message in the given sequence.

2:

The message is being accepted strictly in the program below by using the sequence number. We are allocating window size to 2 as max, by using this we specify that receiver receive only 2 messages at its end. We can also increase the size of the window as per our need.

SENDER	RECEIVER
define seq, last, i, w: int; initially seq = 0, last = -1, w=2 (The window size is 2 as the maximum) do last+1 ≤ seq ≤ last+w → send (msg [seq], seq); seq =seq+1 (ack, i) is received → if i>last → last:=i i ≤ last → skip timeout (R, S) → next := last+1; {retransmission begins}	define i : int; initially i = 0; do(msg [seq], seq) when received → if i = next → accept message; send (ack, i); i := i+1 i ≠ seq → send (ack, i-1);

In the above program we make sure that it accepted only when it receives the message.
If the message is not received it should be stored in buffer.
If the program timeout we reinitialize the program once again until all the messages are received in the increasing ordered sequence.

References:

- Sukumar Ghosh. *Distributed Systems: An Algorithmic Approach.* 2nd Ed. Chapman/CRC 2014.
- http://homepage.divms.uiowa.edu/~ghosh/16612.week10.pdf
- Discussed with fellow mates.
- Professor's slides from CS-6550.