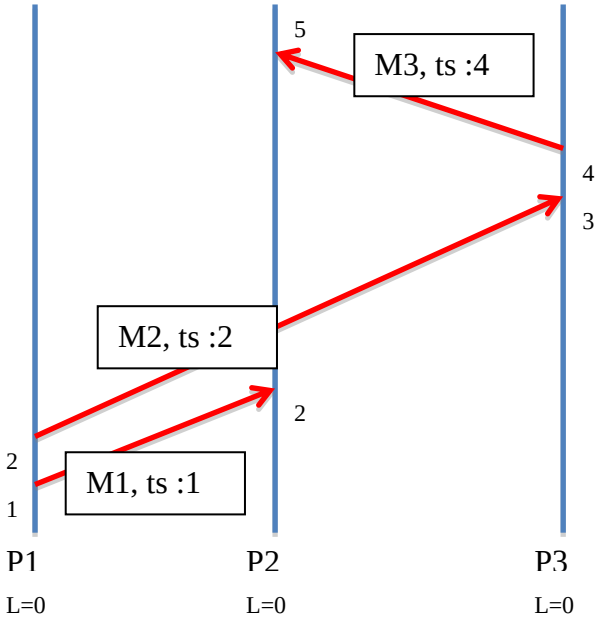
Distributed Algorithmics – TD2 -
M2 IFI, Ubinet-CSSR

Exercice 4 – CORRECTION and SOME EXPLANATIONS

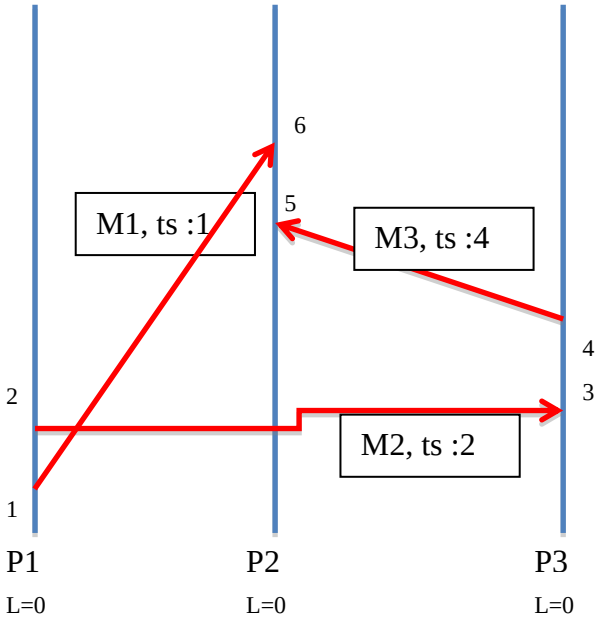
Exercice adapted from one from Chapter 6 of Ghosh book.

In a network of N processes (N>2), all channels are FIFO (and bi-directional), and of infinite capacity. Every process is required to accept data from the other processes in strictly increasing order of timestamps. You can assume (i) processes send data infinitely often, and (ii) no messages is lost in transit.
First, build a 3 processes example, that illustrates this requirement on the following scenario: Process P1 sends 2 messages, one to P2, one to P3. On reception, P3 sends a new message to P2. All messages are timestamped using Lamport's clocks. Then, propose an implementation to make it possible that processes respect the requirement. (Hint: Consider using null messages through a channel to signal the absence of a message from a sender).

Proposed simple scenario should have helped you to identify the requirement, i.e. the core problem to solve.
A first possible execution raises no problem because P2 receives two messages from the others, and by chance, treat them in increasing order of their timestamps (1 for M1, 4 for M3).



A second possible execution raises a problem according to our requirement: because P2 receives the two messages from the others, but not following an increasing order of their timestamps (1 for M1, 4 for M3). Remark: channels are FIFO, but remember channels are point-to-point connections. So, M1 has not bypassed M2 while they were transmitted on the two different channels.



So the problem we want now to solve is: **how to make sure that when P2 receives a message from another process, it is sure that it can safely treat this message because it will not receive later any message from the other processes that may be timestamped with a lower value.**
The simple idea behind will be, on each process, to systematically await one message from each possible source before deciding to effectively treat a message at the application level. This implies that each time a process sends an applicative-level message to a given destination, it will also broadcast a "null" message (with a timestamp) to the others. So the receiver is sure that eventually it gets a message from each of the other process, so that it can compare the timestamps and decide when it is safe to treat a message because now it is the oldest one.
There it is useful to rely upon the given assumptions:

1. *infinite capacity of channels and no messages is lost in transit. :* there is no problem whenever a process sends many messages to a given destination, all these messages are buffered and no one is lost
2. *processes send data infinitely often:* as it is difficult to spontaneously trigger sending of null messages, each time a process sends a message with application level data to another process, we will take this opportunity to transmit a null message to the others. As data messages are sent infinitely often, this also implies that null messages will also be sent to the others infinitely often

Algorithm on Process Pi could be such that for each application level message that is received, Pi has to also receive one message from each other, so that it can order these messages according to their timestamps. Of course, the usual algorithm of logical clocks management due to Lamport still applies, as in the examples above.

<u>First version for an Algorithm on Pi</u>
{Initialization}: // create a Log for incoming messages from the other processes
Message M = null, with M.ts= - infinity
For each k<>i Log[k]=M; // initialize each provided message from Pk except Pi, as empty

{a message M has arrived}: // and in the sequel, consider sender of M is process number j

Synchronize Pi own clock using Lamport algorithm

Log[j]=M;

/* check if in the log, the message from Pj is the older one, thanks to timestamps ts (timestamps are ordered by the '<' relation, as they correspond to integers and in case of equality, we also use the process number corresponding to the sender, here j, to have '<' be a total order relation) */

OK=true;

For each k<>i and k<>j { if Log[k].ts <M.ts then OK=false }

// if OK still true here, means M holds the oldest timestamp

If OK and M<> null Deliver M to the application level

{xxx any guard and code in which a sending of an applicative level message could happen}:

// we apply both classical Lamport algorithm by timestamping sent messages, but

// more importantly for the exercise: we also broadcast a null message to the others

Pi own clock +=1

Send the message M timestamped with Pi own clock to the destination, and Send a message "null" timestamped the same to all other processes

There are 2 remaining problems in this first solution,
First one comes from the fact that if Log[j] already contains a non null message not yet delivered to the application, but Pj has send again an application level message, we will lose the not yet delivered messages by overwriting Log[j] with M. So, we must buffer them safely. Such a situation can happen whenever Pi is still awaiting a message (null or not null) from Pk, k<>j, and in the meantime, Pj sends many application level messages to Pi.

So we can use the assumption about channels being of infinite capacity: the guard {a message M has arrived} changes to {a message M originated from j has arrived and (Log[j] =null or Log[j] <> null but has already be delivered)} and make sure that when M is delivered to the application level, the corresponding Log[j] entry is marked as delivered. This implies that the message M will still be kept in the communication channel whenever there is a risk of overwriting a not yet delivered application level message. This does not prevent of course to receive messages coming from other channels.
Remark: when we enter the code that is guarded by a guard such as {a message M has arrived} this means that we consume the message M, i.e. the code effectively extracts M from the communication channel. Say in another way, crossing a guard {a message M has arrived} and starting to execute the code behind the guard means that the first implicit line (i.e. even if not explicitly written) of this code is "receive M" which effectively takes the message M out from the channel.

Second one is that when we receive a null message M, of course, no need to deliver it at all to the application. But, receiving M could in fact mean that a non null message can now be the oldest one, so must now be delivered. So, the code should be rewritten as:

{a message M originated from j has arrived and (Log[j] =null or Log[j] <> null but has already be delivered)}

Synchronize Pi own clock using Lamport algorithm

Log[j]=M;

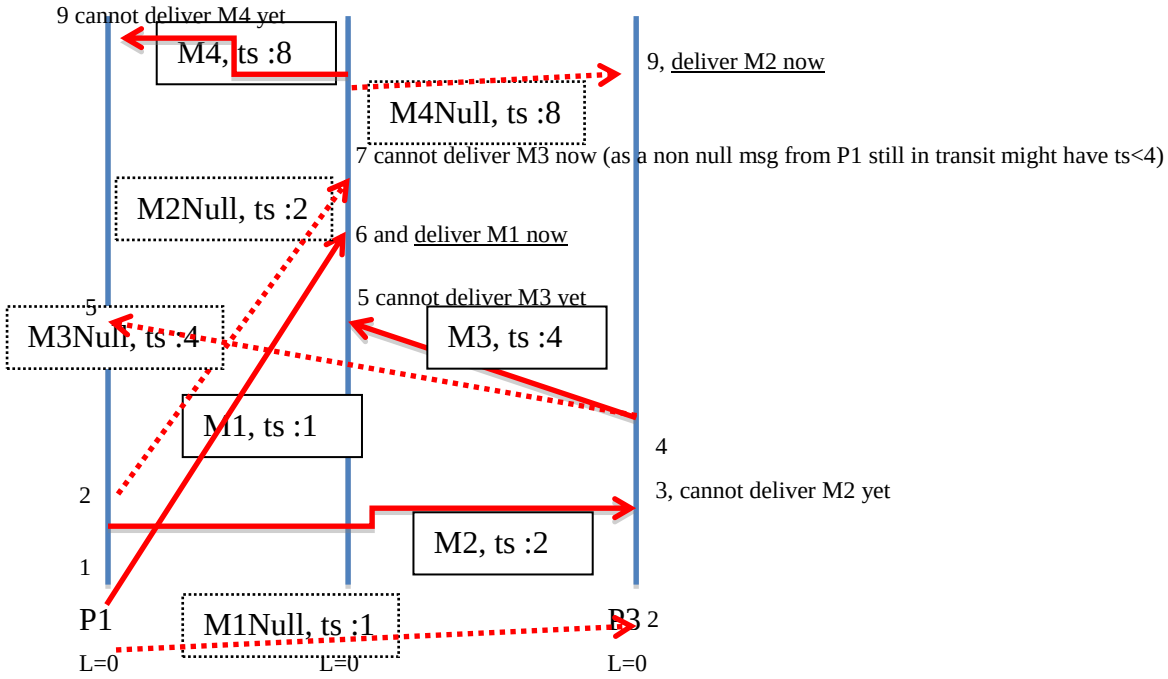
/* check if in the log, if there is a non null message now ready to be safely delivered to the application layer */

OldestM=M;

// Find the entry containing the message with the smaller timestamp (oldest message)
For each k<>i and k<>j { if Log[k].ts <OldestM.ts then OldestM=Log[k] }
// OldestM contains the message which holds the oldest timestamp
// It can happen that OldestM has already been delivered, if it is a non null message
// In this case, we do not deliver it again
If (OldestM<> null) and (OldestM not yet delivered)
        Deliver OldestM to the application level
        mark corresponding log entry as delivered

Remark that we do have to consider the non null messages already delivered to the application layer in the finding of the oldest message in the log. And so, the oldest message may be one application level message already delivered. Indeed, we must be sure that the process say k, that sent this already delivered message does not have send a second application level message, still in transit, but that could be the oldest one among all the ones we have received and in the Log. So, we need to wait for either a null or non null message from this process k. Implicitly, this waiting translates into considering that the yet already delivered message is the oldest, thus refraining from delivering other non null messages from other processes. The assumption number 2 is here crucial, as it ensures that we will eventually receive a message from k, thus letting another message in the Log becomes the oldest.

Below you find an illustration of the proposed algorithm on the (extended) scenario.



Remark that we need that P2 sends a message (M4) to the others otherwise we are blocked in delivering messages to the application level on P3. Also, remark that P1 will not deliver M4 when receiving it, which is safe. Indeed, it has a message from P3 (M3Null, with timestamp 4), but, perhaps another message from P3 with a timestamp less than 8 (the one of M4) might be in transit. Eventually, P3 will send something to P1 according to assumption 2. Also remark that upon reception

of M2Null, P2 is not able to deliver M3 now for the reason that M2Null has timestamp 2. So again, P2 is safe to wait because P1 could have sent to P2 an applicative level message with timestamp e.g. 3, less than the one of M3.

Final remark: you will be happy to notice that the principle of this solution is in fact very close to the Mutual Exclusion algorithm proposed by Lamport as an application for his logical clocks (see very last course). Indeed, before entering into the critical section, a process must be sure that no other process made a request for mutual exclusion that may be older than the one made by itself.