Master 2 IFI Ubinet – Exam, 3 hours
An algorithmic approach to distributed systems

F. Baude, L. Henrio

13 November 2017

1 Election (approx 4 pts)

Here is an election algorithm for arbitrary topologies. It assumes each process knows the diameter of the graph.

```
Algorithm on Pi:
  var r=0 // round number
  constant D // diameter of the topology
  var L=i // leader value, initially set at i when election starts
  while (r < D)
      send L to all neighbors;
      receive message from each neighbor; // blocking operation
      // until Pi gets msg from each of its neighbors
      L=max_of( L and all integer values received from neighbors) ;
      r = r+1 ;
  end while
```

Apply this algorithm to the following network where the diameter is 4. You can simulate the execution of the algorithm using rounds where in each round you tell what each process does (alas in practice processes are totally asynchronous).
Generalize: At which round number is the process elected. Does this algorithm requires a proclamation phase additionally ? What is the parallel time complexity, and the messages complexity; do we have to consider best, average and worst cases ?
Compare: How this approach compares to a probe-echo approach for the election (Chang&Roberts and Tel like examples) ? How this approach compares to a wave approach for the election (Franklin like example) ?

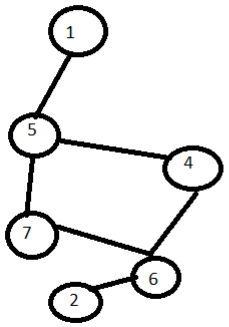

Figure 1: Topology example for leader election

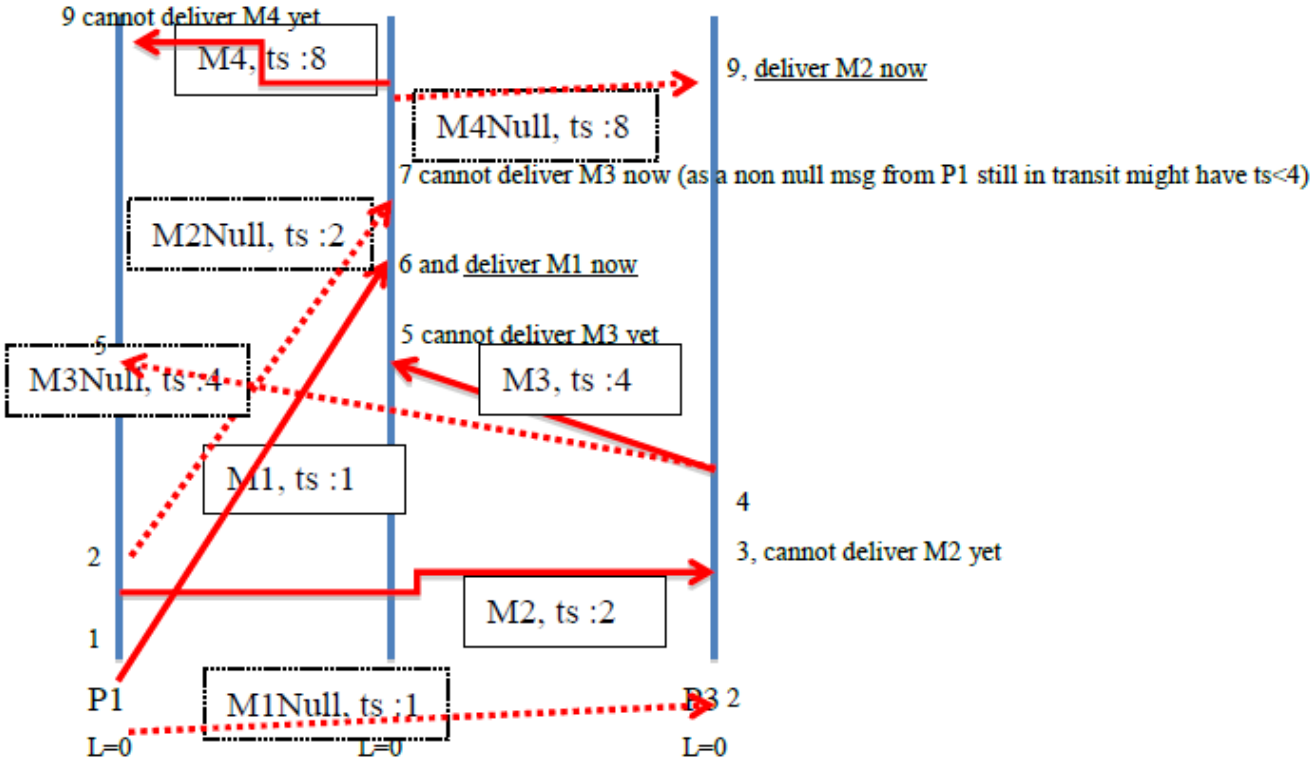

Figure 2: Broadcasted messages between the 3 processes

2 Physical Time (approx 2 pts)

Cristian' method allows any process **P** to synchronize its physical clock against a recognized source of time **S**.

If the bidirectional communication link between P and S is featuring same performances in the direction P to S and from S to P, explain why, after the synchronization, the clock deviation of P compared to S' clock is null (assuming it takes no time at S to read its clock).

Assume now that it takes a given delay d for S once it has read its clock value, in order to prepare the message to send back to P; in this case, explain what is the clock deviation of P' clock compared to S' clock after the synchronization. Is now P' clock in advance, or later compared to official S' clock ?

3 Group communication (approx 5 pts)

The solution to the exercise 4 given on the wiki (TD2 section) that you were allowed to try to answer two times, is not far from being the same as a group communication "broadcast" algorithm.

Question 1 Reproduce on your copy the message exchanges between P1, P2, P3 as you can read on the figure 2: do not make distinction between null messages and non-null ones. And, it is implicit that each process also delivers the broadcast message to itself. Then apply the CBCAST instead of the exo4-TD2 solution as shown on figure 2.

Question 2 Now explain what happens for the message M3 arrived on the figure 2 at time 7 on P2. Can it be delivered if we use CBCAST instead of the exo4-TD2 proposed algorithm ? And why.

Question 3 Wrap-up: if we used exo4-TD2 solution as a causal-order broadcast algorithm, why the given assumption that "processes send data infinitely often" would be so important ? And why such assumption isn't required in CBCAST ?

Also, why CBCAST first propagated message from any process can hold Vector=(0,0,0) as timestamp ? Should we have also done the same for the exo4-TD2 solution (i.e. that would mean: consider that we send the message with the current timestamp before incrementing the local logical clock), or isn't it so necessary ?

Question 4 This exo4-TD2 solution is thus very close to Reliable-Causal Order. But does it also ensures Total order or not ? And does CBCAST ensures total order ?

4 Problem: Mutual Exclusion + Fault-tolerance(approx 9pt)

Consider the algorithm in Figure 3 (studied during the course – Suzuki-Kasami). It is a slightly improved version. <> means "is different from". This algorithm ensures mutual exclusion (see the course for an example of execution).

Part 1 (approx 3 points) Compared to the algorithm of the course, the initialization is more precise and a new variable `awaiting` has been added.

Question 1.1 How is the variable *awaiting* used in the algorithm of the figure?

Question 1.2 Show that if awaiting=true on process j then a) j has the token, and b) Q is empty.

Question 1.3 Consider the beginning of the execution (for example), we have awaiting=true. Suppose a first request from process k arrives and is handled and executed up to line 32 (included). Suppose additionally another request from process l is handled but the token is not sent yet. What happens? is there a request lost? How many tokens are sent? What is the value(s) of the sent token(s)?

Question 1.4 same question if we exchange the two last lines of the algorithm

To simplify the rest of the problem, we suppose now that we cannot execute two events "Upon receiving a request" at the same time on the same process.

Part 2: Lost token (approx 6 points) We suppose now that the transmission of the token is not reliable (all other communications are still reliable). We suppose that the distributed system is synchronous (each message is received within bounded time). We denote T the maximal communication delay. We want to add an acknowledgement message to ensure that the token is eventually received:

Question 2.1 How does this work and how much time do you need to wait?

Question 2.2 Define the new algorithm (you can use line numbers to explain your changes and avoid writing everything). You should send and receive Ack messages. To implement the waiting for acknowledgement you can use something of the form:

```
Wait until Message is received or Timeout(Time) occurs Then
  if Message received do ...
  if Timeout occurs do ...
```

You should choose message and Time conveniently.

```
{Program of process j}
Initialization
  for all i do req[i] :=  0
  req[0]:=1
  awaiting=false
  if j=0 then
    for all i do last[i] :=  0
    last[0]:=1
    awaiting:=true

while (some critical section to execute)
  * Run Non-critical section
  // Entry protocol
    req[j] := req[j] + 1
    Send (j, req[j]) to all
    Wait until token (Q, last) arrives from a process k'
    * RUN Critical Section *
  // Exit protocol
    last[j] := req[j]
    for all k <> j do
      if (k not in Q) and (req[k] = last[k] + 1) then  append k to Q
    if (Q is not empty) then send (tail-of-Q, last) to head-of-Q
      else awaiting:=true
end while

Upon receiving a request (k, num)
    req[k] := max(req[k], num)
    if (awaiting) then
      for all k  do
        if (k not in Q) and (req[k] = last[k] + 1) then  append k to Q
      if (Q is not empty) then
        awaiting:=false
        send (tail-of-Q, last) to head-of-Q
```

Figure 3: Suzuki-Kasami algorithm

Question 2.3 What can happen if the system is not synchronous or the chosen bound is wrong? In this case one of the property (ME1-ME3) of mutual exclusion is lost, which one?

Question 2.4 Can you add information in the token and/or in the acknowledgement so that the algorithm also works for an asynchronous system. (be careful not to have a process stuck or forget sending/receiving an acknowledgement message)? Suggest a new algorithm.

Question 2.5 Finally, we consider a partially synchronous system: there is a maximal bound T on the communication time but it is not known initially. Can you improve the algorithm so that eventually the token is never received twice. Note that this should be done in addition to the previous solution because during an unknown time, some duplicate tokens might be received.

5 Termination (approx 1 pt)

Explain how many rounds the Misra termination detection algorithm needs in order for the initiator to detect that the application is indeed terminated. One, two, more rounds ?