Master Ubinet
Distributed Algorithms
2013

F. Baude, L. Henrio

The marks for each exercise is only an indication.

1 Election (synthesis question) (3pts)

Except for the algorithm entitled 'Bully' for leader election, none of the other algorithms we have studied (in particular approximation based election) assume the total number N of participating processes in the election or the identifiers of participating processes is known.

Discuss what is the impact of this assumption on the proposed solutions. Say another way, how easy it could be to solve the election (without failure) in case N and identifiers were known ? Sketch how such solution would proceed to elect a leader.

On the contrary, discuss how the Bully algorithm strongly relies upon knowing both N and process identifiers.

2 Clocks (2pts)

If events corresponding to vector timestamps Vt1, Vt2, ... , Vtn are mutually concurrent, then prove that Vt1 [1] , Vt2[2], ... Vtn[n] = max (Vt1,Vt2, ..., Vtn)

3 Group communications (5pts)

We studied an exercise in details regarding the use of a sequencer for implementing an almost fully reliable Total Order broadcast primitive (3rd homework, see in Annex the proposed solution for this algorithm in case you had not printed it). In this implementation, we did'nt had to rely explicitly upon the knowledge of N, the number of involved processes in the group, nor the identities of these processes. However, the Reliable Broadcast primitive needs to know this number and more importantly, processes identities, because it regularily pings them to detect if they have crashed.

a) (0.5 pts) Recall briefly how the provided solution for the TO-bcast already supports crash failures of processes (except the failure of the sequencer process)

Taking this knowledge about which are (and thus, how many) the involved correct processes in the group, the **goal of this exercise is to extend the TO-bcast primitive implementation so that it can now also recover the sequencer process failure**. You can imagine that the sequencer is simply the leader in the group. For leader election, you can assume that it handles the case when many candidates are proposing to be elected concurrently. Say another way, once an election has started, the module will eventually deliver to each correct process in the group, the information about the elected leader.

b) (1 pts) First explain which is the particular treatment the sequencer is responsible of, and consequently, what is the impact on the TO-broadcast protocol if it fails. And consequently, what needs to be recovered to replace the sequencer process.

c) (1 pts) Then provide the general principle of your solution

d) (2.5 pts) Give the precise code (in the module oriented approach we used in the Group Communication course, like the way the Reliable Broadcast for instance is presented). To this aim, you can assume for example that both failure detector and leader election are available modules. However, give precisely which are their interfaces.

4 Problem: Fault-tolerance and mutual exclusion (10 points)

This problem considers an adaptation of the mutual exclusion token-based algorithm based on two imbricated rings. It consists of 2 independent sub-parts; 4.1 on mutual exclusion, and 4.2 on recovery protocols. Figure 1 shows a set of processes organised in a double ring, each ring of length $N = 3$. The labels on the edges indicate how each process is referenced from the previous one (e.g. when **P** sends a message to **Left**, the message will be sent to **P1**).

The algorithm considered is detailed below. The process that has the token can execute its critical section if necessary, it then passes the token to the next process. The process **P** initializes the algorithm by sending the token to itself.

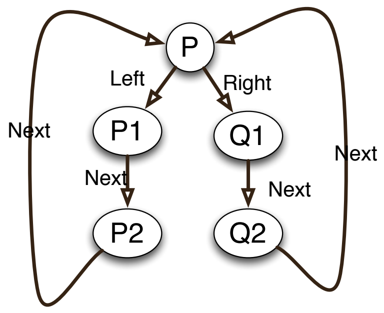

Figure 1: Double ring topology

```
Algorithm for ME-doubleRing on each process
CurrentProcess is the name of the current process, i.e., P or Pi or Qi
--------------------------------------
initialization:
   TokDestination:=Left
   CSRequested:=False
   if CurrentProcess = P then send <token> to P

Upon event RequestCriticalSection
   CSRequested:=True // remember that a critical section is to be run

Upon reception of a message <Token> from process Q
   if CSRequested then
        CSRequested:=False
        RunCriticalSection
   if currentProcess = P then // for central process send to left or right alternatively
        send message <Token> to process TokDestination
        if TokDestination = Left then
             TokDestination:=Right
        else
             TokDestination:=Left
   else                      // for other processes simlpy pass the token
        send message <Token> to process Next

Upon RunCriticalSection
   .... // code of the critical section
```

4.1 Mutual exclusion (6 points)

a) (approx 1.5 points) *Recall properties (ME1-ME3) of mutual exclusion that we have seen in the course, and prove that the new algorithm satisfies them.*

Concerning ME3, you will evaluate the maximal number of messages that have to be transmitted before a process can enter its critical section (Hint: this correspond to the number of messages exchanged between the moment a process exits from its critical section, and the moment when the process enters it again). Give this bound (maximal number of message) for process P and for others, in the case of Figure 1, i.e. when $N = 3$, but also depending on N in the general case.

Compare this bound to the case of a normal token ring (of the course).

To improve this bound, one decides that the process that wants to enter its critical section will send a request for critical section to process **P**. The token should still follow the arrows of Figure 1, but will take the branch of the process that requires the token. If no critical section is requested, the token stays in **P**. If both branches request the token, the token is sent alternatively to one branch and to the other. Note that when the token is sent to one branch, all the processes of the branch will have the opportunity to run their critical section (a process can never require a new critical section before executing the previous one).

b) (approx 2.5 points) *Modify the algorithm above to implement this new token strategy.*

You can add internal variables or messages between processes if necessary; you can also wait until a condition is satisfied, e.g. `Wait until P≠0`, but there are also solutions that do not use such a wait statement.

c) (approx 1.5 points, difficult) *Show that your new algorithm still verifies ME1 - ME3; also show that, in case communications are asynchronous with no bound on communication time the bound on number of communications to get the token can be worse than in the first algorithm. (NB:*

to find such an exampole, you must consider a worst-case scenario considering communication time)

d) (approx 1 point, difficult) *Suppose now that communications are synchronous, i.e. communications are considered as instantaneous. Suppose that the token is in **P** with no other request. What is, depending on N, the number of communications needed for a process to get the token (in the worst case).*

Conclude briefly on the advantages and drawback of the new algorithm.

4.2 Fault-tolerance (4 points)

We consider again the simple algorithm ME-DoubleRing suggested at the beginning of the problem (not the one you proposed in the previous sections). Now, each process takes a local checkpoint when entering its critical section. We consider that the processes enter their critical sections each time they receive the token (i.e. they immediately require a new critical section when the preceding one is finished). In appendix you will find an example of execution of the algorithm that you can use to answer questions. Do not forget to put a legend and explain where you answer to which question!

We consider an execution as shown in appendix (the drawing stops when **P2** is in its critical section for the second time).

a) 0.5 point *In this particular example, what would be the consequence of an orphan message? in other words What would be the consequence if one would recover the system from an inconsistent cut?*

Hint: take a recovery line with only one orphan message (you can use checkpoints that are not marked on the figure for the questions **a) and b)**).

b) 0.5 point *Same question concerning in-transit messages. (i.e. what would be the consequence if you recover from a consistent but not strongly consistent cut).*

In the next questions, we do not care about in-transit messages (we suppose they are automatically recovered by the system; for example the processes could detect that the message has been lost).

c) (1 point) *Suppose process P2 crashes just after completing its critical section (and its local checkpoint) for the second time (as drawn). Use a dependency graph to compute the best recovery line consisting only of the checkpoints marked (plus potentially the current state of the non-crashed processes).*

d) (0.5 point) *Suppose now that process Q1 crashes (at the point of the drawing, i.e. just after P2 finished its second critical section) find the new recovery line; verify that it is a consistent cut!*

e) (1 points) *To improve the system, one decides to implement a CIC protocol which will piggyback checkpoint numbers to trigger forced checkpoints and avoid unnnecessary rollbacks. Draw the new execution and find the recovery line in the two cases studied above (P2 and Q1 crashes).*

f) (0.5 point) *Explain briefly what happened in the CIC protocol in this particular case, and why are there a lot of forced checkpoints ?*

Annex: the TO bcast protocol

Here is the sketch of a TOTAL order (reliable) broadcast algorithm relying upon a sequencer, using the Reliable Broadcast (RB) group communication module. If a process wants to broadcast a message, it triggers the RB-multicast(¡m,i¿) instruction, with m, and for each m, an identifier i (it could be a combination of process identifier emitter and a number chosen by it, so that each i is unique over all the sent messages).

```
Algorithm for TO-Bcast on each process
--------------------------------------
On initialization, r:=0

To TO-Bcast message m:
        RB-multicast(<m,i>) // bcast the message to all, including to the sequencer process

On RB-deliver(<m,i>)
        Place <m,i> in hold-back queue

On RB-deliver(<''order'',i,S>)
        wait until <m,i> in hold-back queue and S=r
        // the wait does not prevent the process to react to another event
        // like, typically, another RB-deliver(...) concerning another i value.
        TO-deliver(m)
        r:=S+1

Algorithm for sequencer
-----------------------
On initialization s:=0
On RB-deliver(<m,i>)
        RB-multicast(<''order'',i,s>) // as order messages are not relevant for
        //the sequencer process, we can consider that this RB-multicast
        //broadcasts this message order to all processes except the sequencer
        s:=s+1
```

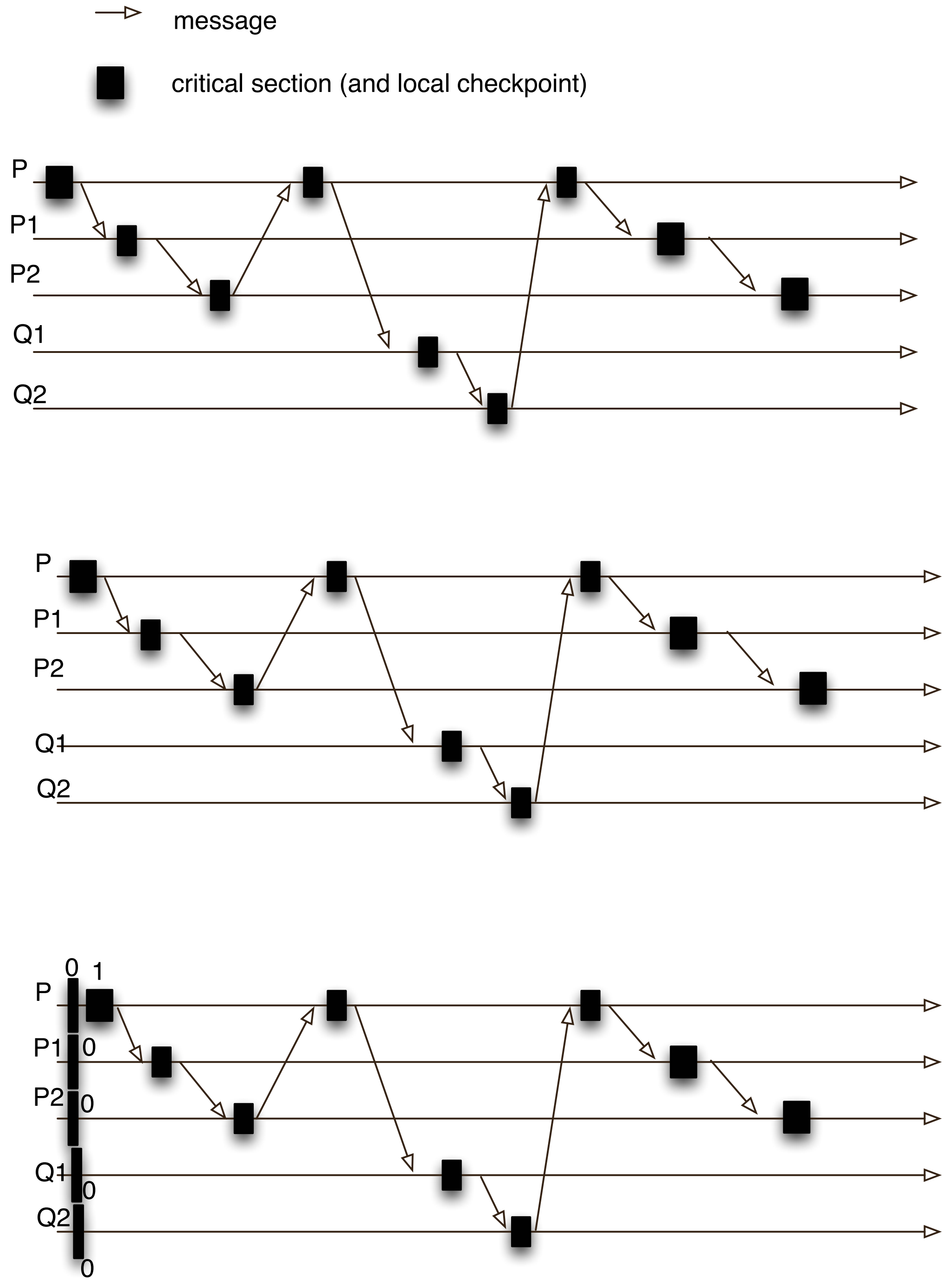

Figure 2: Executions for fault-tolerance questions