Master IFI
Distributed Algorithms
2014

F. Baude, L. Henrio

The marks for each exercise is only an indication.

1 Election (4 pts)

We have studied Franklin algorithm for electing a leader on a bi-directional ring, assuming implicitly that all processes are candidates, thus getting O(logN) rounds, each round taking at most a O(N) parallel time complexity.

The goal of this exercise is to study this algorithm in the case of a uni-directional ring.

1- Describe the algorithm briefly (you can use phrases)

2- Execute it on the graph of slide 23, supposing links are going in the clockwise direction (ie. $0- > 2- > 1- > 3- > 9- > 6- > 7- > 5- > 0$). How many rounds do you need to elect the highest numbered process as leader, ie 9. Sketch briefly what happens in each round

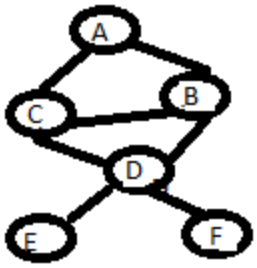
3- Generalize: what is the **worst case situation**, and in this case how many rounds would be needed; and to evaluate the parallel time complexity given the unit of time is considered being the transmission from one message to its neighbour, what is the time complexity; what is the **best case situation**, and in this case how many rounds would be needed and what is the parallel time complexity

4- Discuss this approach using a unidirectional ring, against the one proposed by Chang and Roberts for a unidirectional ring also. Compare them briefly first on their underlying principle, second on their theoretical parallel performances. Would you advise to select one or another, and in case yes, which one.

2 Probe Echo *mysterious* algorithm (3pts)

Given this algorithm, and the network sketched below.

```
Algorithm on each process Pi, initiator being A (REVISED VERSION, as Ri was incomplete, Sorry!)
-----------------------------------------------------------
local variables:
boolean initiator = false, except on process A where it is true
```




```
boolean started = false
set neighbours = ids of all my neighbours
int Nl // number of received messages
float X=0 // the mysterious variable
Ii: { initiator AND NOT started}
     started = true
     send Pr to neighbours
     Nl=0
Mi: { NOT started AND <message of type Pr has arrived> }
     started = true
     Nl = 1
     parent = Pr message's sender
     send Pr to <neighbours \ parent> // no effect if no neighbour except parent
     X = 1
     if (Nl == |neighbours| )
         send <Echo, X> to parent
Ri: { started AND <message of type Pr has arrived> }
     Nl = Nl + 1
     X = X + 0.5
     if (Nl == |neighbours| )  // indeed, it could happen that echo is received but it is not
         if ( NOT initiator) // from the 'last' neighbour. So, last message arriving could be a Pr.
             send <Echo, X> to parent
         else print "final result is " + X
              // algorithm is terminated
Ei: { <message of type Echo has arrived> }
     Nl = Nl + 1
     X = X + Echo.X // Echo.X is the field that holds the number in the mess.
     if (Nl == |neighbours| )
         if ( NOT initiator)
             send <Echo, X> to parent
         else print "final result is " + X
              // algorithm is terminated
```

Explain what information is gathered by this algorithm and consequently what is printed at the initiator side when the final result is ready to be printed. To help in your explanation, use the provided graph as an illustrative example.

More importantly, discuss why you think this algorithm is designed this way. Hint. Remember from home work 1 that on a given bi-directionnal link, you may have two probes and not simply one probe and one echo message that are travelling.

3 Group communications (3pts)

We assume there are no failures (even if RB is used later in the algorithm).

The goal of this exercise is to study the behaviour of the TO (total order) broadcast algorithm provided below. Does it allow to ensure all messages are delivered according to the **happens before** relation ? (remember that this relation addresses also consecutive local messages, not only messages exchanged between processes; i.e. if m1 is created by Pi before another message m2 created later on Pi, m1 *happens before m2.*)

```
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
```

Notice that the interactions with/from the sequencer are achieved using broadcast communication, and that the topology can be any: so a message from a process to the sequencer and vice versa can take any route as the process is not mandatorily connected in a direct point-to-point manner with the sequencer.

Explain synthetically, using diagrams showing message exchanged between processes to illustrate your arguments, if the provided TO broadcast algorithm ensures Causal Order property.

4 Mutual exclusion problem and fault tolerance (10 pts)

We want to provide a mutual exclusion protocol for a system where the process $P0$ is considered as prioritary and should be able to enter its critical section as soon as possible, possibly overtaking the other processes.

First part: token ring Consider the token ring approach with $n+1$ processes: $P0$ .. Pn but with all processes able to communicate with $P0$. The idea is to modify the token ring protocol so that the process $P0$ (i.e. the prioritary one) can enter its CS as soon as possible: You can use a broadcast message so that the process Pi directly sends the token to $P0$. $P0$ should wait

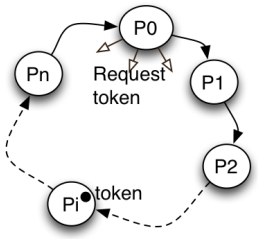


Figure 1: Illustration for a token ring

at maximum that one other process does its critical section if the broadcast was instantaneous. After its critical section $P0$ starts again the normal algorithm, i.e. gives the token to $P1$.

1. Explain how the new protocol works in two or 3 sentences.
2. Write the algorithm executed in $P0$; in each process Pi with $i \neq 0$ (Be careful with the special cases, e.g. when Pn sends the token).
3. Recall the 3 properties that should be ensured by mutual exclusion algorithm, is ME1 and ME2 guaranteed by the algorithm? Why?
4. Consider now ME3.
   - Does $P0$ verify ME3? how long does it have to wait before entering its critical section (in the worst case, and counting in number of critical sections executed).
   - Consider a process other than $P0$, how long does it have to wait before entering its critical section (in the worst case, and counting in number of critical sections executed).
   - Conclude: Is ME3 guaranteed?

Second part: improvement of the algorithm We now try to improve this algorithm in order to ensure ME3. Write an algorithm that includes in the token the number of the last processs visited.

- Use this to write a better algorithm that returns the token to the process $P(i+1)$ if $P0$ received the token from Pi. Be careful with the special cases: when Pn sends the token, and when P0 received a token with number n
- Prove that your algorithm ensures ME1, ME2, and ME3

Fault-tolerance Consider the second solution (this should not change anything except for the bonus question). We want to apply a message logging fault-tolerance algorithm for this application. We suppose that communications are reliable, only processes might fail. We consider two cases:

1. Each process takes a checkpoint after sending the token.
2. Each process takes a checkpoint before sending the token.

Questions:

- Explain the principle of message logging applied to the example. Number your lines of code and indicate which lines are impacted by the message logging algorithm and how (you should differentiate the two cases).
- What can happen at recovery in the two cases? Explain a possible recovery scenario for each case. Can the token be lost in case of failure in case 1? in case 2? (explain why the token is not lost or give an example if the token can be lost)

Bonus question (difficult): Suppose now that the communications are not reliable and that we do not use message logging (because we do not have any reliable storage at hand). Design an algorithm that detects when the token is lost and restore it if necessary. Explain your algorithm and write its pseudo-code. You can restore the token: in P0, or where it was lost (more difficult).

You can add as many informations as necessary inside the token or inside the processes.