Department of Computer Science
COMP212 - 2018 - CA Assignment 1
Coordination and Leader Election
Simulating and Evaluating Distributed Protocols in Java

Assessment Information

Assignment Number	1 (of 3)
Weighting	7%
Assignment Circulated	20th February 2018
Deadline	16th March 2018, 17:00 UK Time (UTC)
Submission Mode	Electronic via Departmental submission system
Learning outcomes assessed	(1) An appreciation of the main principles underlying distributed systems: processes, communication, naming, synchronisation, consistency, fault tolerance, and security. (3) Knowledge and understanding of the essential facts, concepts, principles and theories relating to Computer Science in general, and Distributed Computing in particular. (4) A sound knowledge of the criteria and mechanisms whereby traditional and distributed systems can be critically evaluated and analysed to determine the extent to which they meet the criteria defined for their current and future development.
Purpose of assessment Marking criteria	This assignment assesses the understanding of coordination and leader election in distributed systems and implementing, simulating, and evaluating distributed protocols by using the Java programming language. Marks for each question are indicated under the corresponding question.
Submission necessary in order to satisfy Module requirements?	No
Late Submission Penalty	Standard UoL Policy.

1 Overall marking scheme

The coursework for COMP212 consists of two assignments and a class test contributing altogether 20% of the final mark. The contribution of the individual assignments is as follows:

Assignment 1	7%
Assignment 2	7%
Class test	6%
TOTAL	20%

2 Objectives

This assignment requires you to implement in Java two distributed algorithms for leader election in a ring network.

3 Description of coursework

Throughout this coursework, the network on which our algorithms are to be executed is a bidirectional ring, as depicted in Figure 1.

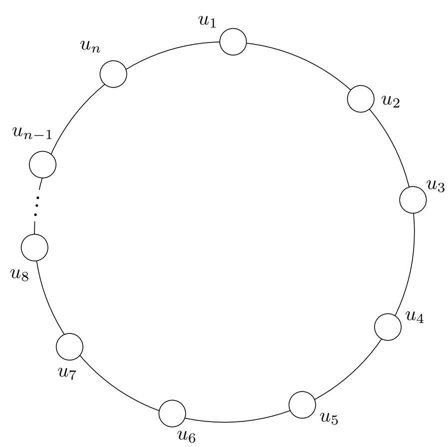


Figure 1: A bidirectional ring network on n processors.

In our setting, all processors execute the same algorithm, do not know the number n of processors in the system in advance, but they do know the structure of the network and are equipped with *unique* ids. The ids are not necessarily consecutive and for simplicity you can assume that they are chosen from $\{1, 2, \ldots, \alpha n\}$, where $\alpha \geq 1$ is a small constant (e.g., for $\alpha = 3$, the n processors will be every time assigned unique ids from $\{1, 2, \ldots, 3n - 1, 3n\}$). Additionally, every processor can distinguish its clockwise from its counterclockwise neighbour, so that, for example, it can choose to send to only one of them or to send a different message to each of them. Processors execute in synchronous rounds, as in every example we have discussed so far in class.

3.1 Implementing the LCR Algorithm—30% of the assignment mark

As a first step, you are required to implement the LCR algorithm for leader election in a ring. The pseudocode of the non-terminating version of LCR can be found in Lecture 7 and is also given here for convenience (Algorithm 1).

Algorithm 1 LCR (non-terminating version)

Code for processor u_i, $i \in \{1, 2, \ldots, n\}$:

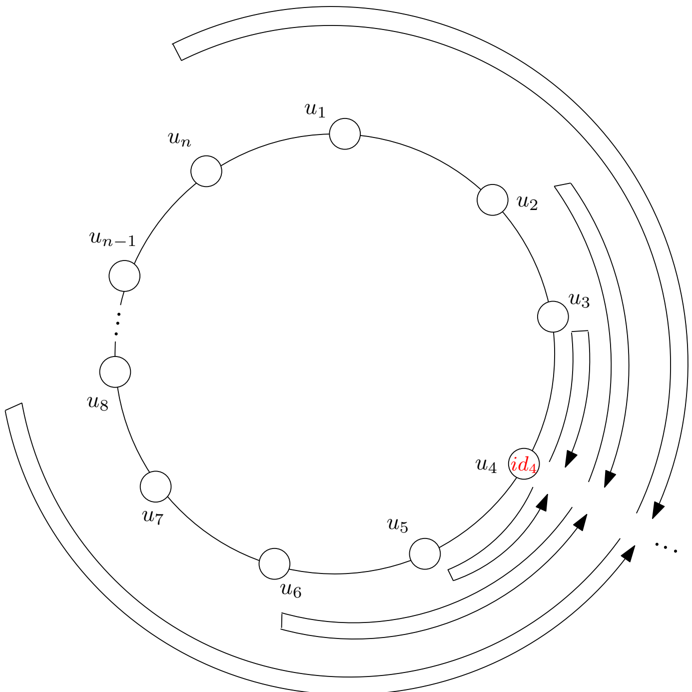
Initially:
u_i knows its own unique id stored in $myID_i$
$sendID_i := myID_i$
$status_i :=$ "*unknown*"

if $round = 1$ **then**
    send $\langle sendID_i \rangle$ to clockwise neighbour
else// $round > 1$
    upon receiving $\langle inID \rangle$ from counterclockwise neighbour
    if $inID > myID_i$ **then**
        $sendID_i := inID$
        send $\langle sendID_i \rangle$ to clockwise neighbour
    else if $inID = myID_i$ **then**
        $status_i :=$ "*leader*"
    else if $inID < myID_i$ **then**
        do nothing
    end if
end if

You are required to implement a *terminating version* of the LCR algorithm in which all processors eventually terminate and know the id of the elected leader.

3.2 Implementing the HS Algorithm—30% of the assignment mark

Next, you are required to implement another algorithm for leader election on a ring, known as the HS algorithm. As LCR, HS also elects the processor with the maximum id. The main difference is that HS instead of trying to send ids all the way around in one direction (which is what LCR does), it has every processor trying to send its id in both directions some distance away (e.g., k) and then has the ids turn around and come back to the originating processor. As long as a processor succeeds, it does so repeatedly (in "phases") to successively greater distances (doubling the distance to be travelled each time, e.g., $2k$). See Figure 2 for an illustration.



轨道，轨线

Figure 2: Trajectories of successive "phases" originating at processor u_4 (imagine the rest of the processors doing something similar in parallel, but not depicted here). The id transmitted by u_4 aims to travel some distance out in both directions and then return back. If it succeeds, then u_4 doubles the aimed distance and repeats.

Informally, each processor u_i "operates in phases" $l = 0, 1, \ldots$ (where each phase l consists of one or more rounds). In each phase l, processor u_i sends out a "token" (i.e., a message) containing its id id_i in both directions. These are intended to travel distance 2^l (that is, as in Figure 2, distance $2^0 = 1$ for $l = 0$, distance $2^1 = 2$ for $l = 1$, distance $2^2 = 4$ for $l = 2$, and so on) and then return to their origin. If both tokens manage to return back then u_i goes to the next phase, otherwise it stops to produce its own tokens (and only performs from that point on the rest of the algorithm's operations). A token is discarded if it ever meets a processor with greater id while travelling outwards (away from its origin). While travelling inwards (back to its origin), a token is forwarded by all processors without any check. The termination criterion is as follows: If a token travelling outwards meets its own origin u_i (meaning that this token managed to perform a complete turn of the whole ring while travelling outwards), then u_i elects itself as the leader. Observe that in order for tokens to know how far they should travel each time and in which direction, this information has to be included inside the transmitted messages (that is, apart from the id being transmitted, the messages should also contain this auxiliary information).

The pseudocode of the non-terminating version of HS is given in Algorithm 2. As with LCR, **you are required to implement a *terminating version* of the HS algorithm in which all processors eventually terminate and know the id of the elected leader**.

3.3 Experimental Evaluation, Comparison & Report—40% of the assignment mark

After implementing the terminating LCR and HS algorithms, the next step is to conduct an experimental evaluation of their correctness and performance.

Correctness. Execute each algorithm in rings of varying size (e.g., $n = 3, 4, \dots, 1000, \dots$; actually, up to a point where simulation does take too much time to complete) and starting from various different id assignments for each given ring size. For instance, you could execute them on both specifically constructed id assignments (e.g., ids ascending clockwise or counterclockwise) and random id assignments. In each execution, your simulator should check that eventually precisely one leader is elected. Of course, this will not be a replacement of a formal proof that the algorithms are correct as you won't be able to test them on all possible combinations of ring sizes and id assignments, but at least it will be a first indication that they may do as intended.

Performance. Execute, as above, each algorithm in rings of varying size and starting from various different id assignments for each given ring size. For each execution, your simulator should record the number of rounds and the total number of messages transmitted until termination.

1. Execute both algorithms in rings of varying size for the case in which ids are always clockwise ordered.

2. Execute both algorithms in rings of varying size for the case in which ids are always counterclockwise ordered.

3. Execute both algorithms in rings of varying size and various random id assignments for each given ring size. Note here that both algorithms should be simulated (e.g., one after the other) on every given choice of ring size and id assignment, so that a comparison of their performance makes sense.

In Summary: For both correctness validation and performance evaluation a suggestion is to simulate both algorithms (for all types of id assignments mentioned above) in rings containing up to at least 1000 processors. Specifically in the case of random id assignments, for each ring size n repeat the simulation for many different id assignments (e.g., at least 100 distinct simulations) and record the correctness and the worst, the best, and the average performance so that you get meaningful results.

Algorithm 2 HS (non-terminating version)

Messages are triples of the form $\langle ID, direction, hopCount\rangle$, where $direction \in \{out, in\}$ and $hopCount$ positive integer.

Code for processor u_i, $i \in \{1, 2, \dots, n\}$:

Initially:
u_i knows its own unique id stored in $myID_i$
$sendClock_i$ containing a message to be forwarded clockwise or $null$, initially $sendClock_i := \langle myID_i, out, 1\rangle$
$sendCounterclock_i$ containing a message to be forwarded counterclockwise or $null$, initially $sendCounterclock_i := \langle myID_i, out, 1\rangle$
$status_i \in \{\text{“}unknown\text{”}, \text{“}leader\text{”}\}$, initially $status_i :=$ “$unknown$”
$phase_i$ recording the current phase number, nonnegative integer, initially $phase_i = 0$

前提

upon receiving $\langle inID, out, hopCount\rangle$ from counterclockwise neighbour
if $inID > myID_i$ and $hopCount > 1$ **then**
    $sendClock_i := \langle inID, out, hopCount - 1\rangle$
else if $inID > myID_i$ and $hopCount = 1$ **then**
    $sendCounterclock_i := \langle inID, in, 1\rangle$
else if $inID = myID_i$ **then**
    $status_i :=$ “$leader$”
end if

upon receiving $\langle inID, out, hopCount\rangle$ from clockwise neighbour
if $inID > myID_i$ and $hopCount > 1$ **then**
    $sendCounterclock_i := \langle inID, out, hopCount - 1\rangle$
else if $inID > myID_i$ and $hopCount = 1$ **then**
    $sendClock_i := \langle inID, in, 1\rangle$
else if $inID = myID_i$ **then**
    $status_i :=$ “$leader$”
end if
pass without check
upon receiving $\langle inID, in, 1\rangle$ from counterclockwise neighbour, in which $inID \neq myID_i$
$sendClock_i := \langle inID, in, 1\rangle$

upon receiving $\langle inID, in, 1\rangle$ from clockwise neighbour, in which $inID \neq myID_i$
$sendCounterclock_i := \langle inID, in, 1\rangle$

upon receiving $\langle inID, in, 1\rangle$ from both clockwise and counterclockwise neighbours, in both of which $inID = myID_i$ holds
$phase_i := phase_i + 1$
$sendClock_i := \langle myID_i, out, 2^{phase_i}\rangle$
$sendCounterclock_i := \langle myID_i, out, 2^{phase_i}\rangle$

// The following to be always executed by all processors, i.e.,
// also in round 1 in which no message has been received
send $sendClock_i$ to clockwise neighbour
send $sendCounterclock_i$ to counterclockwise neighbour

After gathering the simulation data, plot them as follows. In each plot, the x-axis will represent the (increasing) size of the ring and the y-axis will represent the complexity measure (e.g., number of rounds or number of messages). You may produce individual plots depicting the performance of each algorithm (possibly comparing against standard complexity functions, like n, $n \log n$, or n^2) and you are *required* to produce plots comparing the performance of both algorithms in identical settings. For example, when measuring the total number of messages in the case of counterclockwise increasing ids, a plot would show at the same time the performance of both algorithms for increasing ring size n, using curves of different colours and possibly also a legend with explanations. Then, for each given ring size, the corresponding point of each curve will represent the total number of messages generated by the algorithm (indicated on the y-axis). You can use gnuplot, JavaPlot or any other plotting software that you are familiar with.

The final *crucial* step is to prepare a concise report (at most 5 pages including plots) clearly describing your main implementation choices, the main functionality of your simulator, the set of experiments conducted, and the findings of your experimental evaluation of the above algorithms. In particular, in the latter part you should try to draw conclusions about (i) the algorithms' correctness and (ii) the performance (time and messages) of each algorithm individually (e.g., what was the worst/best/average performance of each algorithm as a function of n? For example, we know from the lectures that the worst-case communication complexity of LCR is $O(n^2)$: can you verify this experimentally?) and when the two algorithms are being compared against each other (e.g., which one performs better and in which settings?).

4 Deadline and Submission Instructions

- The deadline for submitting this assignment is **Friday, 16th March 2018, 17:00 UK time (UTC)**.

- Submit
  (a) The Java source code for all your programs,
  (b) A README file (plain text) describing how to compile/run your code to produce the various results required by the assignment, and
  (c) A concise self-contained report (at most 5 pages including everything) describing your implementation choices, experiments, and conclusions in PDF format.

  Compress all of the above files into **a single ZIP** file (the electronic submission system won't accept any other file formats) and **specify the filename** as *Surname-Name-ID*.zip. It is extremely important that you include in the archive all the files described above and not just the source code!

- Submission is via the departmental submission system accessible from `https://sam.csc.liv.ac.uk/COMP/Submissions.pl?strModule=COMP212`

Good luck to all!