Coursework: Distributed Systems 2014

January 2014 (v1.1)
(revised, February 7, 2014)

1 Administration

- Assigned Date: Monday January 27th 2014
- Due Date: 4pm Thursday March 13th 2014

The expectation is that you should spend approximately 2 hours per week on practical work. You have a little over six weeks, hence 8-12 hours work on this practical should be a reasonable amount of time, though of course this is subject to variability depending upon the student.

This is an individual coursework: work may be discussed but all of the work you submit must be your own.
This coursework has **two parts**:

1. A programming exercise (described in Section 2)
2. Written theory questions (described in Section 3).

For the programming part, you should submit a single source file containing your solution. For the theory part, please put your answers into a single PDF with questions numbers 1,2,3,4 clearly marked. Submission instructions are in section 4, and a summary of the grading criteria and marking scheme is in section 5. Some answers to common questions are collected at the end in Section 6.

2 Programming Exercise

2.1 Introduction

In this practical you will implement a simulation of two distributed algorithms. A simulator will allow you to reason about a distributed algorithm prior to development and deployment on the potentially large scale of a real distributed system. You are to write a program which:

1. implements **Lamport's clocks** as discussed in the Time and Global State part of the course. This is a useful service that underlies many distributed algorithms.
2. implement a **distributed mutual exclusion** algorithm for controlling shared access to a resource. For full credit the algorithm used must be *fair*.

Both parts rely on material covered in lectures in Weeks 3-5. This material can also be found in Coulouris (chapter 14-15) or other distributed systems textbooks listed in the course web page.

Parts 1 and 2 can be done independently if a mutual exclusion algorithm that does not rely on clocks is used, such as ring, token or Maekawa's voting algorithm; however, these algorithms are not fair and so will not receive full credit. To ensure fairness, Part 2 should build on part 1 (using Lamport clocks to implement the Ricart/Agrawala algorithm or to ensure fairness in Maekawa's voting algorithm). The Ricart/Agrawala algorithm discussed in class that uses Lamport clocks is probably the easiest choice of fair algorithm to implement in part 2; if you would like to try implementing some other fair algorithm, please check this with us.

Your program will need to take as input a description of several process schedules (i.e., lists of send, receive or print operations). The output of your program will be a linearization of these events in the order actually performed, annotated with Lamport clock values.

2.2 Input format and description of behavior

The input of the program will be a collection of processes, each with an associated *schedule*, or list of operations to perform. The processes are named p1...pn for some n (you may assume that n is at most 9.) The format of a process is:

```
begin process p1
operation
…
operation
end process
```

where each line contains an operation. Operations include *basic operations* and *mutex blocks*. The possible basic operations are:

- `send` *`pN msg`* (that is, send message *`msg`* to *`pN`*)
- `recv` *`pN msg`* (that is, receive message *`msg`* from *`pN`*)
- `print` *`msg`* (that is, print message *`msg`* to the terminal)

where *`msg`* is any alphanumeric string. We describe mutex blocks later.

The `send` operation simply sends a message and the process is free to continue executing the next operation. The `recv` operation *blocks* and waits to hear message *`msg`* from a given process. (This means that there can be deadlocks if all processes are waiting to receive and there are no messages in transit.)

The `print` operation provides us with a simple form of shared resource among processes. An individual print operation takes place atomically, but if two processes are concurrently printing multi-line messages then the results may be interleaved.

Messages can be sent and received, and printing can take place, in any order, provided causality is respected: that is, the order of events within a process is preserved, and a message is always sent before it is received. One approach is for your simulator to maintain a pool of messages including sender, receiver and payload. When a message `msg` is sent from `p1` to `p2`, we add message `m = (p1,msg, p2)` to the pool, and when the message is received by `p2`, `m` is removed from the pool. This can be done explicitly by maintaining a set or list of messages in the simulator, or implicitly by using one thread per process and setting up communication channels between each pair of threads; in this case, the "pool" is the combined contents of all message buffers between pairs of threads.

In addition, there is a compound operation called an mutex block. An mutex block is as follows:

```
begin mutex
  basic_operation …
  basic_operation
end mutex
```

where each of the enclosed operations is a basic operation (i.e. mutex blocks cannot be nested, they can only contain sequences of send, receive or print operations). Events in different mutex blocks should not execute concurrently. Thus, a process that wants exclusive access to the output terminal (for example to print a multi-line message without being interrupted) should wrap the print operations in a mutex block. Part of this coursework is to implement a mutual exclusion algorithm that ensures this.

For convenience, individual print operations occurring outside a mutex block are treated as if they each are in their own mutex block. Thus, even printing a single line requires exclusive access to the terminal.

Here is a small example illustrating the input format:

```
begin process p1
  send p2 m1
  begin mutex
    print abc
    print def
  end mutex
end process

begin process p2
  print x1
  recv p1 m1
  print x2
  send p1 m2
  print x3
end process p2
```

This example describes two processes. The first starts by sending a message, then prints "abc" and "def" to the terminal. The second prints x1, then receives from the first process, then prints x2, then sends to the first process, then prints x3. Because of the mutex block, the two print operations performed by p1 should not be interrupted by any other print operations (particularly x2). Note that the message sent from p2 to p1 is never received; this is fine.

2.3 Output format

The output format is a single *log* of the events that took place during the simulation run, one per line, including Lamport clock timestamps. The possible events are:

- `sent` *`pN msg pM T`* (that is, *`pN`* sent message *`msg`* to pM at local time *`T`*)
- `received` *`pN msg pM T`* (that is, *`pN`* received message *`msg`* from pM at local time *`T`*)
- `printed` *`pN msg T`* (that is, *`pN`* printed message *`msg`* to the shared terminal at time *`T`*)

Mutex blocks are not reflected explicitly in the log. Your implementation of mutual exclusion may involve additional special messages being exchanged by processes. These *should not* be reported in the log, only those events corresponding to operations in the schedules.

Continuing the example above, the following is a valid output:

```
printed p2 x1 1
sent p1 m1 p2 1
received p2 m1 p1 2
printed p1 abc 2
printed p1 def 3
printed p2 x2 3
sent p2 m2 p1 4
printed p2 x3 5
```

Several other outputs are also possible. However, because of the mutex block, it should not be possible for `p2` to print `x2` between the two prints performed by `p1`.

However, it is OK for the Lamport clock timestamps to be different, as long as they are correct with respect to the happens-before ordering - this could happen for example if Lamport clocks are used to support fair mutual exclusion, because this may lead to additional messages being sent/received (and timestamped) to support acquiring and releasing locks. If so, these messages should not be reported in the output.

2.4 Further details

The practical is designed such that your program will accept textual input and produce textual output. This means both that you are free to choose your implementation language and that we may automate evaluation of your solution to some extent. Whatever language you choose to implement your solution in you should make sure that it can be compiled and run on a standard DICE desktop.

2.5 Sample code for parsing input

Since parsing textual input is not really the point of the practical, we provide sample code in Java to show how this is to be done. The following code can be used verbatim, or you can adapt it to your language of choice. This is just one approach, but are are free to come up with your own parsing solution. The key point is that your program should accept input and produce output in the formats specified above.

```
// the name of the input file is passed as argument
// when running the programme
BufferedReader in = new BufferedReader(new FileReader(args[0]));
String line = null;

Process p = null;     // is a class that holds a group of instructions
                      // (send, recv, print, mutex..)
                      // it’s up to you how you want this to be


// for each line of the input file
while ((line = in.readLine()) != null) {

     // split the line into an array of words
     String[] arr = line.trim().split("[ ]+");

     // if the first word is “begin”
     if (arr[0].equals("begin")) {
          if (arr[1].equals("process")) {

          // the instructions for a new process will begin
          // after this command in the input file
```

```
            // TODO - create a new process for the next instructions

            } else

            if (arr[1].equals("mutex")) {

                  // after this command all the following instructions
                  // should be executed “atomically” - as one.

                  // TODO - treat the starting of an atomic block

            }

      } //  end of dealing with the begins

      // the end commands
      if (arr[0].equals("end")) {

            if (arr[1].equals("mutex")) {
                  // TODO - mark the end of an atomic block
            } else if (arr[1].equals("process")) {
                  // TODO - deal with the end of instructions for a
                  //process
            }

      } // end of ends

      // the send instructions
      if (arr[0].equals("send")) {
            // TODO - deal with the send instruction
            // arr[1] is the destination
            // arr[2] is the message
      }

      // the receive instructions
      if (arr[0].equals("recv")) {
            // TODO - deal with the receive instruction
            // arr[1] is the sender
            // arr[2] is the message
      }

      // the print instructions
      if (arr[0].equals("print")) {
          // TODO - deal with the print instruction
          // arr[1] is the payload of the task
     }

}    // end of while

in.close();

// now that the file is parsed, you can put to work your processes.
// TODO - execute the instructions associated to each process.
```

3 Written Theoretical Questions.

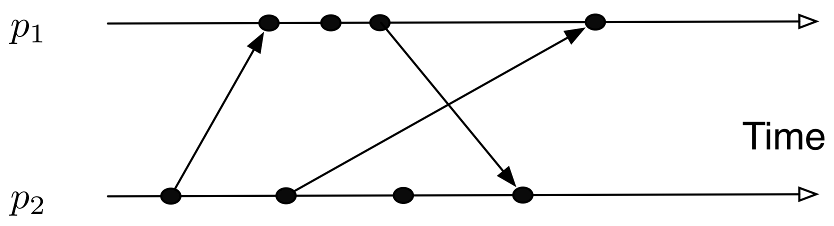

Figure 1: Events for p_1 and p_2.

Q1 [2 marks] Figure 1 above shows events occurring for each of two processes, p_1 and p_2. Arrows between processes denote message transmission. Draw and label the lattice of consistent states (p_1 state, p_2 state), beginning with the initial state $(0, 0)$.

Q2 [2 marks] Even without a deadlock, a poor mutual exclusion algorithm may lead to starvation. Give an example of a system which leads to starvation.

Q3 [3 marks] Suppose a distributed system is operating in synchronous rounds. The network communication graph is given by G. A BFS tree T has been constructed with root at node r. There is a function f whose value at any node p is given by $p.f$, and $p.f$ is known to p.

Give a distributed algorithm that computes the average and maximum of f over all nodes with $O(n)$ time and message complexity. The computed values must be known to all nodes.

Q4 [1 mark] Figure 2 shows a graph representing a network, with weights on edges. What is the weighted diameter of this network? (that is, diameter of the network, where the weight of an edge represents its length.) Which is

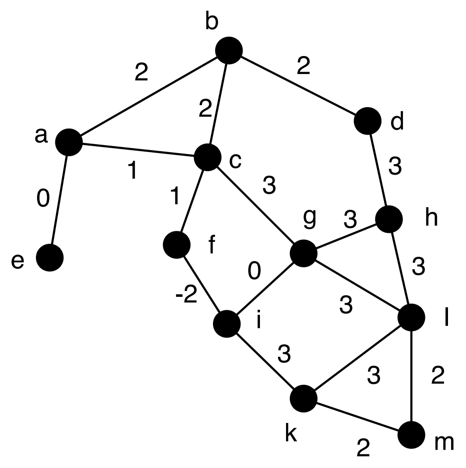

Figure 2: Weighted network graph

the path that realizes the diameter?

What will be the diameter if the graph was unweighted, and what will be the corresponding path?

4 What To Submit

You must submit the source code of your solution to the programming problem as a single source code file. The source code should compile and run on a standard DICE desktop. This practical is not intended to evaluate your coding ability, however the portion(s) of your code which do the logic itself should be well marked and documented.

Please also include a comment presenting your assumptions, implementation design, any notes on how to run your code, and anything else relevant for us to better understand your code (in a concise manner), marked README, at the beginning of your submission file.

Your solutions to the written theory questions should be submitted as a single PDF, with question numbers marked clearly. (As advertised in a previous version of this coursework, your answers to the additional questions could also be given as comments at the end of your source code. If so, please mark these clearly.)

4.1 Evaluation

Your solution will be evaluated primarily on how well your solution performs on various scenarios (including samples we will make available and test cases we will not provide in advance). The key points are:

1. No events should occur out of order, there is some scope for the order to be different as in a real network. However no event should violate the happens-before relation. In particular no process should receive a message before it is sent.
2. The Lamport clock timestamps must be correct with respect to happens-before ordering.
3. Events in mutex blocks should not occur concurrently. In particular, it should not be possible for “print” events from two mutex blocks in different processes to be interleaved.
4. Log file must not contain any entries other than the events from the schedule.
5. For full credit, your mutual exclusion solution should be *fair with respect to happens-before ordering* (we will assess this by inspection of the code.)
6. Your answers to the written questions

5 Grading

The entire course work is graded out of 25, and broken down as follows:

- Lamport clocks [7 marks]
- Mutual exclusion [10 marks] - **at most 5 marks if the mutual exclusion is not fair**
- Additional written questions [8 marks]

6 Clarifications

- Question: Must I code my solution in Java?
- Answer: No, you may choose which ever programming language you prefer. However you should make sure that your solution can compile and run on a standard DICE machine.

- Question: Should I use threads in my solution?
- Answer: Threads are one way to simulate multiple communicating processes. But you are free to simply have a queue of events and an object for each process. You can then simply have a queue manager that takes events from the queue and calls the appropriate method on the object representing the process at which that event takes place. For example you could have a receive method on a process object which then queues further send events.

- Question: Can I use Java's `synchronize` keyword to implement mutual exclusion?
- Answer: In a multithreaded simulator (see previous question), you will probably want to use `synchronize` and probably also Java's concurrency-safe data structure libraries, to manage access to the simulator's shared state. You could also use `synchronize` to implement mutex blocks. However, the point of this coursework is to simulate a **distributed** mutual exclusion algorithm, where processes can only communicate with each other using messages. Therefore, while it is OK to use Java concurrency features within the simulator, please don't use `synchronize` in the implementation of mutual exclusion, instead, implement one of the message-based algorithms.