Operating Systems
Jacobs University Bremen
Dr. Jürgen Schönwälder

Course: 320202
Date: 2015-04-24
Deadline: 2015-05-01

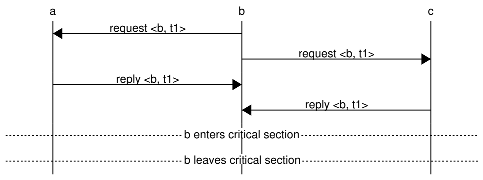
Problem Sheet[1] #5

Problem 5.1: *Ricart-Agrawala algorithm (distributed synchronization)* (10 points)

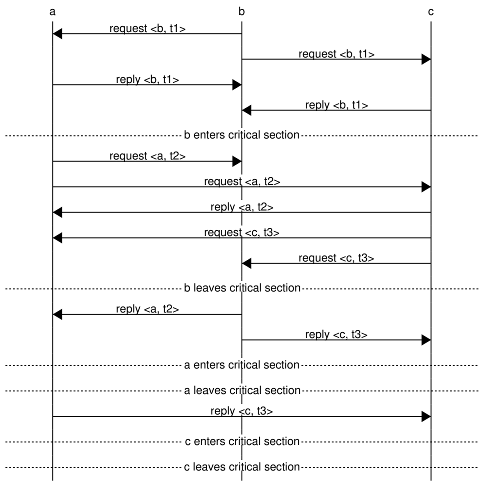
In a distributed system, it is sometimes necessary to synchronize between processes that do not share memory. One mutual exclusion algorithm that does not need shared memory was proposed by Ricart and Agrawala [1]:

- When process p_i wants to enter its critical section, it generates a new timestamp, t_i, and sends a request message containing $< p_i, t_i >$ to all other processes in the system.
- When process p_j receives a request message, it may reply immediately or it may defer sending a reply back. The decision whether process p_j replies immediately to a request message containing $< p_i, t_i >$ or defers its reply is based on three factors:
  1. If p_j is in its critical section, then it defers its reply to p_i.
  2. If p_j did not request itself to enter its critical section, then it sends a reply immediately to p_i.
  3. If p_j wants to enter its critical section but has not yet entered it, then it compares its own request timestamp t_j with the timestamp t_i:
     - If its own request timestamp t_j is greater than t_i, then it sends a reply immediately to p_i (p_i asked first).
     - Otherwise, the reply is deferred until p_j leaves the requested critical section.
- Once process p_i has received a reply message from all other processes in the system, it can enter its critical section.
- Upon exiting its critical section, the process p_i sends reply messages to all its deferred requests.

An example message exchange without contention:



An example message exchange with contention:



[1]See the course web page for submission instructions and grading details.

Write a C or C++ program that implements this algorithm. The main part of the program should conceptually execute this loop:

```
while (1) {
  enter_critical()
  sleep(2)
  leave_critical()
  sleep(5)
}
```

The main loop shown above is conceptual. For the assignment, it is sufficient to implement things in a purely event-driven way (e.g, by using `libevent`. This makes it easy to receive and process messages from peers while sleeping since sleeping can be simply implemented by scheduling a timer event at wakeup time.

Note that the programs that are involved need to startup and connect to each other before the main loop starts. This can be achieved as follows (the `-l` option specifies the listening endpoint and any remaining arguments the existing endpoints to connect to).

```
$ ./raa -l 1234
$ ./raa -l 1235 localhost:1234
$ ./raa -l 1236 localhost:1234 localhost:1235
```

To keep things simple, you can assume a fixed number of processes. The number of processes may be passed as another option (e.g., `-n`) to the processes.

References

[1] G. Ricart and A. K. Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM*, 21(2):9–17, 1981.