UNIVERSITÄT SIEGEN

Fakultät IV
Betriebssysteme und verteilte Systeme
Prof. Dr. rer. nat. Roland Wismüller

Excercise Sheet 9
Solution

Lecture Distributed Systems
Summer Term 2020

Exercise 1: Centralized Mutual Exclusion Algorithm

a) Requests could be assigned priority levels depending on their importance. The coordinator could then serve the highest priority request first.

b) Suppose the algorithm specifies that each request is answered immediately, either with a permission or with a denial. If there are neither processes in a critical section nor in the queue, a crash is not fatal.

The next process requesting a permission receives no response and can then initiate the election of a new coordinator. The system can be made even more robust by allowing the coordinator to save each incoming request to disk before returning a response. In this way, in the event of a crash, the new coordinator can reconstruct both the list of active critical sections and the queue by reading the file from disk.

Exercise 2: Algorithm of Ricart and Agrawala

Suppose a process denies permission and then crashes. The requesting process believes it is still alive, but the permission never arrives. A way out is to cause the requester not only to block, but to *sleep* for a certain amount of time, after which he queries all processes that have denied permission to check if they are still running.

Exercise 3: Deadlocks - Ricart and Agrawala

That depends on the fundamental rules. If the processes enter into critical sections strictly sequentially, i.e. a process in a critical section must not attempt to enter into another critical section, there is no way it can block while holding a resource (i.e. a critical section) that another process needs. The system is deadlock-free. On the other hand, if process 0 can enter critical section A and then attempt to enter critical section B, a deadlock may occur if another process attempts to enter the critical sections in reverse order. The algorithm of Ricart and Agrawala itself does not contribute to the deadlock because each critical section is treated independently of the others.

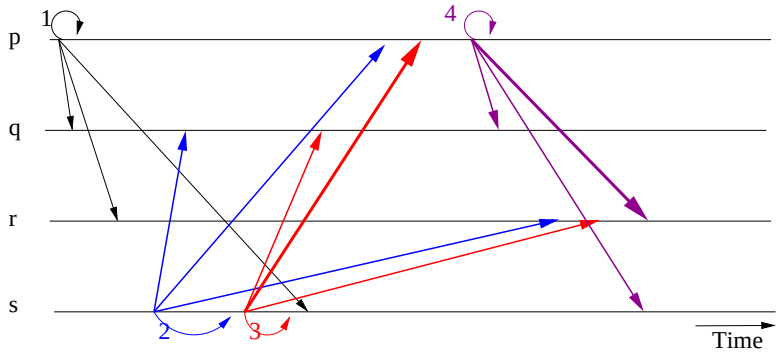
Exercise 4: Programming: Ricart/Agrawala

You will find the solution to this problem in the archive `l09eFiles.zip`[1] on the lecture's web page.

Exercise 5: Multicast Order Guarantees

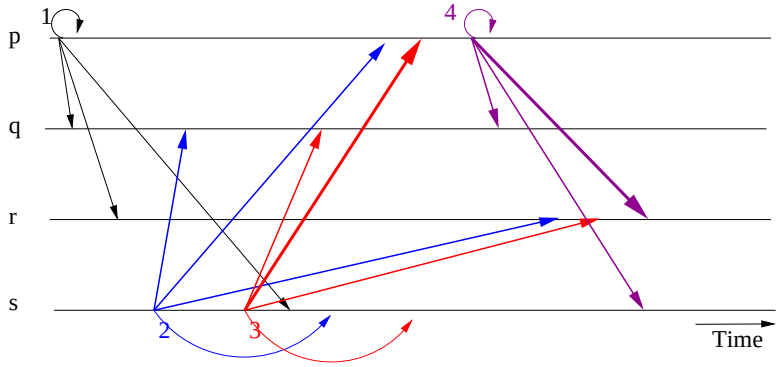
a) With a causally sorted multicast, message 3 must arrive in all processes (especially process p) after message 2 (since 3 causally depends on 2). Message 4 must also arrive in all processes (especially process r) after message 3.

[1] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/vs/l09eFiles.zip`



However, this does not guarantee total order (for example, process p: sequence 1,2,3,4; process s: sequence 2,3,1,4).

A total order would be provided if all messages from all processes were delivered in the same order.



b) For the user to receive all publications, a **reliable** multicast is required.

A **FIFO order** is required so that a user's posts, e.g. A. Bauer, are received everywhere in the same order. Users can then consistently discuss A. Bauer's "second posting". Also, the second post might refer to the first one.

A **causal order** is also needed because the messages whose topics begin with *Re:* should appear after the messages they refer to. Otherwise, delaying a message could cause the message *Re: RPC Principle* to appear before the original message *RPC Principle*.

If the multicast delivery were totally ordered, the numbering in the left column would be consistent between users. However, this is not absolutely necessary.

In practice, the USENET system, which emulates such a bulletin board, implements neither a causal nor a total order. The communication effort required for this would far outweigh its advantages.

Exercise 6: Causally Ordered Multicast

a) Causal order means if a message b depends (or could depend) causally on a message a, then all processes will receive a before b.

The multicast is causally ordered because:

- Each message is provided with the Lamport timestamp of the send event (which is incremented after each message)
- and the recipients will deliver incoming messages in the order of the Lamport timestamps.

So if $a \rightarrow b$ applies, then also $L(a) < L(b)$, i.e. b is delivered to the receiver after a (although b could arrive at the receiver node before a of course; b is then "simpl" withheld).

b) As indicated above, the recipient node may only deliver a message b if all messages with a time stamp smaller than $L(b)$ have already been delivered. But how should the recipient node know that no message m with $L(m) < L(b)$ is on the way anymore?

A straightforward solution would be to simply discard incoming messages with a timestamp smaller than that of the last message delivered. The multicast would then still be causally ordered, but no longer reliable.

Another solution is conceivable, if the individual channels have FIFO semantics: if a node has received messages with a time stamp greater than $L(b)$ from all other nodes, it can deliver b, since surely no message m with $L(m) < L(b)$ can arrive any more. The problem is that you have to wait for messages from all other processes (possibly for a very long time if a node has nothing to send). Possible solutions would be:

- sending periodic broadcasts (problem: still long delay, overhead)
- sending acknowledgements of receipt via multicast (problem: overhead)

An efficient solution is possible with vector clocks. In this case, the vector clocks only count the multicast events. From the vector timestamp of an incoming message, the recipient node then knows exactly which messages to wait for (because the i-th element of the vector timestamp specifies exactly how many events have occurred in process i that are causally before the sending of the message).

c) The problem is that two messages may have the same Lamport timestamp. The recipients would then not know unambiguously which message should be delivered first. Two recipients could therefore deliver the messages in different order.

This would have to be remedied by a linear ordering of the messages, e.g. by combining the Lamport time with a process ID of the sender (see the Ricart/Agrawala alogrithm).