The UNIVERSITY *of* EDINBURGH
SCHOOL *of* COMPUTER SCIENCE

CS4/MSc

Distributed Systems

Jane Hillston

jeh@dcs.ed.ac.uk

Room 1604

(Lecture 11: Distributed Mutual Exclusion,
11th February 2002)

Critical Sections and Mutual Exclusion

- The mutual exclusion problem is the problem of managing access to a shared resource (data, software or hardware) which can support only one process at a time. A printer is an example of such a resource.
- **Critical sections** (or critical regions) are used to program systems with multiple processes which require exclusive access to a shared resource.

```
                    // enter critical section,
enter();            // block if necessary
resourceAccess();   // access shared resources
exit();             // leave critical section,
                    // possibly unblock another process
```

- In a single-processor system constructs such as semaphores and monitors are used to protect critical sections. In a distributed system mutual exclusion needs to be achieved by means of message passing between processes.

Algorithms for Distributed Mutual Exclusion

The key properties of a useful mutual exclusion algorithm are:

Safety: At most one process may execute in the critical section at a time.

Liveness: Requests to enter and exit the critical section eventually succeed. This implies freedom from deadlock and starvation.

Ordering: If one request to enter the critical section *happened-before* another then entry to the critical section is granted in that order.

We will see that most algorithms satisfy the first two conditions but not all satisfy the ordering property.

Performance of the algorithms is also an important issue and is generally judged in terms of the number of messages needed for a process to gain access to the critical section. The **throughput** at the critical section will be determined by the **synchronisation delay**: the delay between one process leaving the critical section and the next entering.

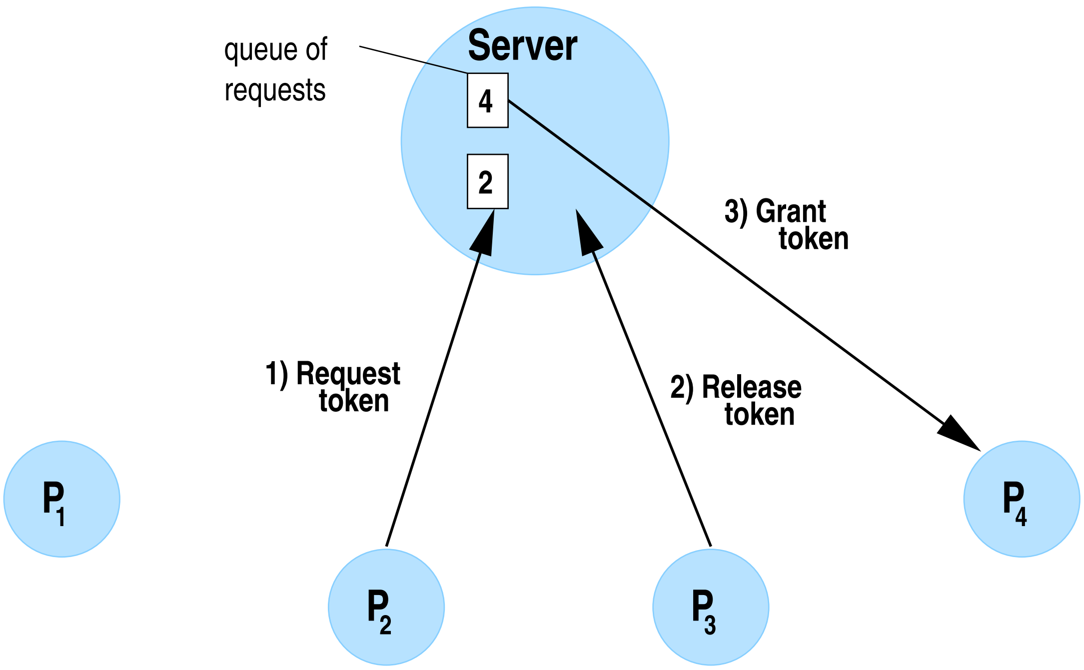
The Central Server Algorithm

One of the processes acts as the **coordinator**.

- Whenever a process wants to enter a critical section, it sends a **request** message to the coordinator seeking permission to enter.
  - If no other process is currently in the critical section the coordinator sends a **grant** message back to process.
  - If another process is currently in the critical section the request is queued.
- When a process leaves the critical section it sends a **release** message to the coordinator upon which the coordinator dequeues a pending request and gives permission to enter the critical section in a **grant** message back to the owner of the request.

This algorithm has the safety and liveness properties but not the ordering property. The synchronisation delay is the time for the **release** and **grant** messages to be sent.

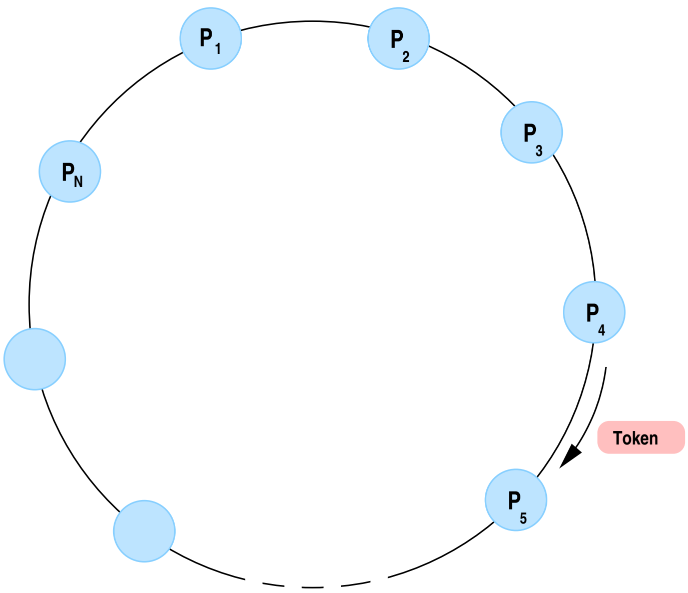
Central Server Algorithm Example



The Ring-Based Algorithm

- The ring-based algorithm removes the dependence on a single process to act as a coordinator.
- Instead of gaining access to the critical section on demand, each process has a regular opportunity to access the critical section.
- The processes are logically arranged in a ring topology so that there is a clear order in which they have the opportunity of access. (This may be unrelated to the physical topology.)
- Exclusion is maintained by requiring that each process must first obtain a **token**, which is simply a message passed from process to process in a single direction.
- If a process does not require access to the critical section when it receives the token it immediately forwards it to its neighbour.
- A process that requires the token waits for it and then retains it.
- To exit the critical section the process sends the token on to its neighbour.

Ring-Based Algorithm Example



This algorithm also has the safety and liveness properties but not the ordering property. The synchronisation delay may be N messages in the worst case.

Ricart and Agrawala's Algorithm (1)

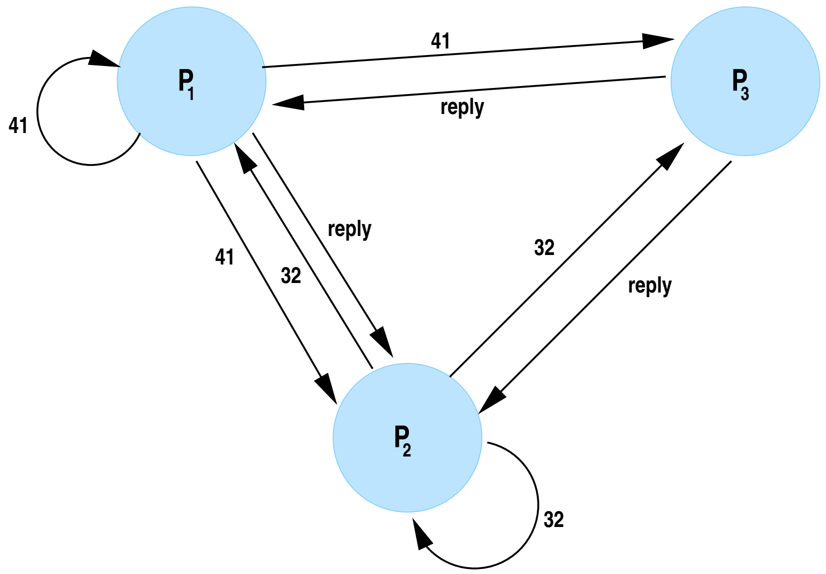
- This algorithm requires total ordering of all events in the system so that access to a critical section can be resolved based on the order requests for entering the critical section were generated.
- Lamport's logical clocks serve as a global clock on which to base the ordering required by the algorithm.
- It is assumed that each process has a distinct identifier and these are added to timestamps to create a total ordering. It is also assumed that the graph of processes and channels is fully connected.
- When a process wants to enter a critical section it sends a Lamport timestamped **request** message to all other processes including itself. It enters the critical section provided that a **reply** message is received from all the other processes. The processes which receive the request use the timestamps to judge whether the incoming request has higher-priority than their own.

Ricart and Agrawala's Algorithm (2)

- When a process receives a **request** message from another process
  - if it is not in the critical section and does not want to enter it, it sends back a **reply** message.
  - if it is already in the critical section, it queues the request and does not reply.
  - if it wants to enter the critical section, it compares the timestamp of the incoming **request** message and that of the message it sent to everyone.
    - □ if the incoming timestamp is smaller, the process sends a **reply** message to the sender.
    - □ otherwise it queues the request and does not reply.
- When a process leaves the critical section, it sends a **reply** message to all processes in its queue and deletes them from its queue.

Ricart and Agrawala's Algorithm Example

Both P_1 and P_2 request access to the critical section. P_2 is successful because it has the lower timestamp.



Ricart and Agrawala's Algorithm Characteristics

- The total ordering of the timestamps means that any process comparing several requests can uniquely determine which process to reply to. This ensures that only one process can receive $N - 1$ replies and thus enter the critical section so the safety property is satisfied.
- The liveness property holds because only request messages that happened-before this request will have priority and each takes a finite amount of time. Consequently all requests will eventually be satisfied.
- The ordering property holds because if one request happened-before another it will have a lower timestamp by the definition of the logical clock.
- Synchronisation delay is reduced to a single message but the number of messages needed to gain access can require $2(N - 1)$ messages if multicast is not supported in hardware.

Voting Algorithms

Ricart and Agrawala's algorithm can be improved by adopting a scheme whereby a process does not need the permission of **all** other processes to enter a critical section.

- Each process is assigned to a **voting set**. The optimal size of the voting sets for N processes is $\sqrt{N}$ and each process should belong to $\sqrt{N}$ sets.
- These sets are chosen to overlap and a process must collect replies from all members of its voting set.
- A member can only reply to one member of its voting set at a time, and will not reply to another request until receiving a **release** or a **relinquish** message.
- However it may try to change its vote if it receives a request with an earlier timestamp before receiving a **release** message. It does this by sending an **inquire** message to the process it last voted for.

Voting Algorithms (2)

- When a process receives an **inquire** message
  - if it has already entered the critical section it discards the message;
  - if it has not already entered the critical section, it decrements its number of votes by one and sends a **relinquish** message to the sender.
- When a process receives a **relinquish** message it puts the corresponding request on the queue of requests. It then sends a **vote** to the request with the earlier timestamp.
- To avoid deadlock outstanding requests are dequeued in *happened-before* order: without this condition the algorithm can fail to satisfy the liveness property.
- The client delay is the same as in Ricart and Agrawala's algorithm when no re-allocation of votes is needed but the synchronisation delay is two messages: **release** and **reply**. When votes need to be re-allocated the number of messages needed may double.

Fault Tolerance

All the algorithms described assume that message delivery is error-free: all messages are delivered in the same order in which they were sent.

- The central server algorithm cannot tolerate failure of the coordinator but can tolerate failure of other processes, as long as they are not holding or requesting the critical section at the time of the crash.
- The ring-based algorithm cannot tolerate the failure of any process.
- The Ricart and Agrawala algorithm can be modified to tolerate the crash of a process which is not holding or requesting the critical section. Such a crashed process is assumed to reply to all requests.
- In the voting algorithms the crash of a process outside the voting set of a requesting process will go undetected, and in other cases the algorithms can be modified as for Ricart and Agrawala's algorithm.