DISTRIBUTED MUTUAL EXCLUSION AND ELECTION

Mutual Exclusion

☞ Mutual exclusion ensures that concurrent processes make a serialized access to shared resources or data.

⇩

The well known *critical section* problem!

☞ In a distributed system neither shared variables (semaphores) nor a local kernel can be used in order to implement mutual exclusion!
Thus, mutual exclusion has to be based exclusively on message passing, in the context of unpredictable message delays and no complete knowledge of the state of the system.

☞ Sometimes the resource is managed by a server which implements its own *lock* together with the mechanisms needed to synchronize access to the resource ⇒ mutual exclusion and the related synchronization are transparent for the process accessing the resource.
This is typically the case for database systems with *transaction processing* (see *concurrency controll* in Database course!)

Mutual Exclusion (cont'd)

☞ Often there is no synchronization built in which implicitely protects the resource (files, display windows, peripheral devices, etc.).

⇩

A mechanism has to be implemented at the level of the process requesting for access.

☞ Basic requirements for a mutual exclusion mechanism:

- **safety**: at most one process may execute in the critical section (CS) at a time;
- **liveness**: a process requesting entry to the CS is eventually granted it (so long as any process executing the CS eventually leaves it).
  Liveness implies freedom of *deadlock* and *starvation*.

☞ There are two basic approaches to distributed mutual exclusion:

1. Non-token-based: each process freely and equally competes for the right to use the shared resource; requests are arbitrated by a central control site or by distributed agreement.
2. Token-based: a logical token representing the access right to the shared resource is passed in a regulated fashion among the processes; whoever holds the token is allowed to enter the critical section.

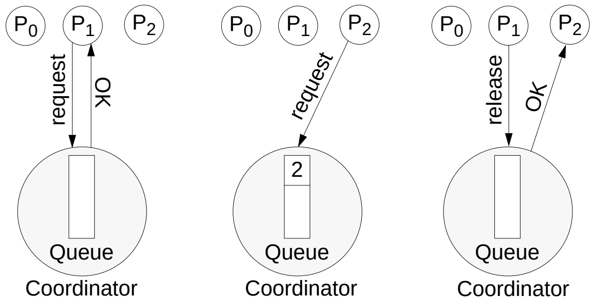
Non-Token-Based Mutual Exclusion

- Central Coordinator Algorithm

- Ricart-Agrawala Algorithm

Central Coordinator Algorithm

☞ A central coordinator grants permission to enter a CS.



- To enter a CS, a process sends a request message to the coordinator and then waits for a reply (during this waiting period the process can continue with other work).
- The reply from the coordinator gives the right to enter the CS.
- After finishing work in the CS the process notifies the coordinator with a release message.

Central Coordinator Algorithm (cont'd)

☞ The scheme is simple and easy to implement.

☞ The strategy requires only three messages per use of a CS (*request*, *OK*, *release*).

Problems

- The coordinator can become a performance bottleneck.
- The coordinator is a critical point of failure:
  - If the coordinator crashes, a new coordinator must be created.
  - *The coordinator can be one of the processes competing for access*; an *election algorithm* (see later) has to be run in order to choose one and only one new coordinator.

Ricart-Agrawala Algorithm

☞ In a distributed environment it seems more natural to implement mutual exclusion, based upon distributed agreement - not on a central coordinator.

☞ It is assumed that all processes keep a (Lamport's) logical clock which is updated according to the rules in Fö. 5, slide 8.

The algorithm requires a total ordering of requests ⇒ requests are ordered according to their global logical timestamps; if timestamps are equal, process identifiers are compared to order them (see Fö. 5, slide 10).

☞ The process that requires entry to a CS multicasts the request message to all other processes competing for the same resource; it is allowed to enter the CS when all processes have replied to this message. The request message consists of the requesting process' timestamp (logical clock) and its identifier.

☞ Each process keeps its state with respect to the CS: *released*, *requested*, or *held*.

Ricart-Agrawala Algorithm (cont'd)

The Algorithm

☞ Rule for process initialization

/* performed by each process P_i at initialization */

[RI1]: $state_{P_i}$:= RELEASED.

☞ Rule for access request to CS

/* performed whenever process P_i requests an access to the CS */

[RA1]: $state_{P_i}$:= REQUESTED.
T_{P_i} := the value of the local logical clock corresponding to this request.

[RA2]: P_i sends a request message to all processes; the message is of the form (T_{P_i}, i), where i is an identifier of P_i.

[RA3]: P_i waits until it has received replies from all other n-1 processes.

☞ Rule for executing the CS

/* performed by P_i after it received the n-1 replies */

[RE1]: $state_{P_i}$:= HELD.
P_i enters the CS.

Ricart-Agrawala Algorithm (cont’d)

☞ Rule for handling incoming requests

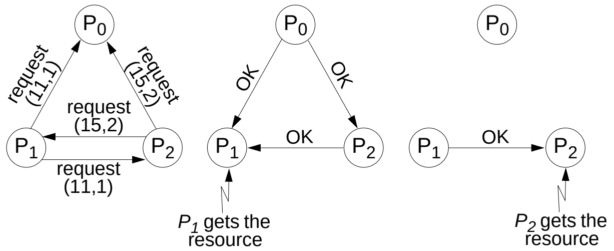
```
    /* performed by $P_i$ whenever it received a request
       ($T_{P_j}$, $j$) from $P_j$ */
[RH1]: if $state_{P_i}$ = HELD or (($state_{P_i}$ = REQUESTED)
                        and (($T_{P_i}$, $i$) < ($T_{P_j}$, $j$))) then
            Queue the request from $P_j$ without replying.
       else
            Reply immediately to $P_j$.
       end if.
```

☞ Rule for releasing a CS

```
    /* performed by $P_i$ after it finished work in a CS */
[RR1]: $state_{P_i}$ := RELEASED.
       $P_i$ replies to all queued requests.
```

Ricart-Agrawala Algorithm (cont’d)

- A request issued by a process P_j is blocked by another process P_i only if P_i is holding the resource or if it is requesting the resource with a higher priority (this means a smaller timestamp) then P_j.



Problems

- The algorithm is expensive in terms of message traffic; it requires 2(n-1) messages for entering a CS: (n-1) requests and (n-1) replies.
- The failure of any process involved makes progress impossible if no special recovery measures are taken.

Token-Based Mutual Exclusion

- Ricart-Agrawala Second Algorithm
- Token Ring Algorithm

Ricart-Agrawala Second Algorithm

☞ A process is allowed to enter the critical section when it got the token. In order to get the token it sends a request to all other processes competing for the same resource. The request message consists of the requesting process’ timestamp (logical clock) and its identifier.
Initially the token is assigned arbitrarily to one of the processes.

☞ When a process P_i leaves a critical section it passes the token to one of the processes which are waiting for it; this will be the first process P_j, where j is searched in order [i+1, i+2, ..., n, 1, 2, ..., i-2, i-1] for which there is a pending request.
If no process is waiting, P_i retains the token (and is allowed to enter the CS if it needs); it will pass over the token as result of an incoming request.

☞ How does P_i find out if there is a pending request?
Each process P_i records the timestamp corresponding to the last request it got from process P_j, in $request_{P_i}[j]$. In the token itself, $token[j]$ records the timestamp (logical clock) of P_j’s last holding of the token. If $request_{P_i}[j] > token[j]$ then P_j has a pending request.

Ricart-Agrawala Second Algorithm (cont'd)

The Algorithm

☞ Rule for process initialization

```
/* performed at initialization */
[RI1]: $state_{Pi}$ := NO-TOKEN for all processes $P_i$, except
       one single process $P_x$ for which
       $state_{Px}$ := TOKEN-PRESENT.
[RI2]: $token[k]$ initialized 0 for all elements $k$ = 1 .. $n$.
       $request_{Pi}[k]$ initialized 0 for all processes $P_i$ and
       all elements $k$ = 1 .. $n$.
```

☞ Rule for access request and execution of the CS

```
/* performed whenever process $P_i$ requests an
   access to the CS and when it finally gets it;
   in particular $P_i$ can already possess the token */
[RA1]: if $state_{Pi}$ = NO-TOKEN then
           $P_i$ sends a request message to all processes;
           the message is of the form ($T_{Pi}$, $i$), where
           $T_{Pi} = C_{Pi}$ is the value of the local logical clock,
           and $i$ is an identifier of $P_i$.
           $P_i$ waits until it receives the token.
       end if.
       $state_{Pi}$ := TOKEN-HELD.
       $P_i$ enters the CS.
```

Ricart-Agrawala Second Algorithm (cont'd)

☞ Rule for handling incoming requests

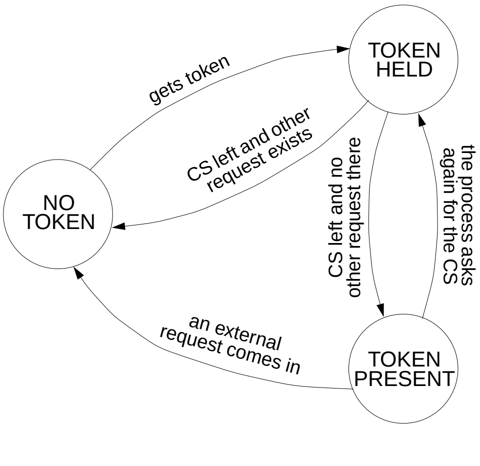
```
/* performed by $P_i$ whenever it received a request
   ($T_{Pj}$, $j$) from $P_j$ */
[RH1]: $request_{Pi}[j]$ := max($request_{Pi}[j]$, $T_{Pj}$).
[RH2]: if $state_{Pi}$ = TOKEN-PRESENT then
           $P_i$ releases the resource (see rule RR2).
       end if.
```

☞ Rule for releasing a CS

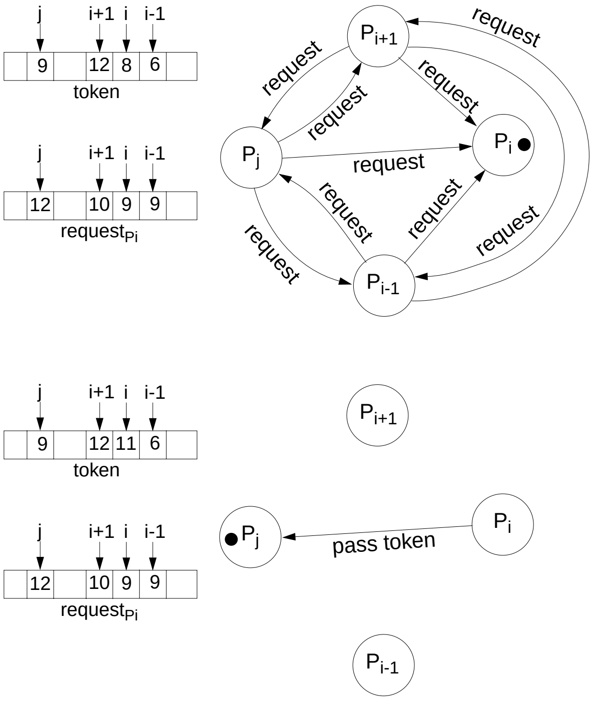
```
/* performed by $P_i$ after it finished work in a CS or
   when it holds a token without using it and it got a
   request */
[RR1]: $state_{Pi}$ = TOKEN-PRESENT.
[RR2]: for $k$ = [$i$+1, $i$+2, ..., $n$, 1, 2, ..., $i$-2, $i$-1] do
           if $request_{Pi}[k]$ > $token[k]$ then
               $state_{Pi}$ := NO-TOKEN.
               $token[i]$ := $C_{Pi}$, the value of the local
                                            logical clock.
               $P_i$ sends the token to $P_k$.
               break.  /* leave the for loop */
           end if.
       end for.
```

Ricart-Agrawala Second Algorithm (cont'd)

☞ Each process keeps its state with respect to the token: NO-TOKEN, TOKEN-PRESENT, TOKEN-HOLD.


TOKEN HELD
NO TOKEN
TOKEN PRESENT
gets token
CS left and other request exists
CS left and no other request there
the process asks again for the CS
an external request comes in

Ricart-Agrawala Second Algorithm (cont'd)



Ricart-Agrawala Second Algorithm (cont’d)

☞ The complexity is reduced compared to the (first) Ricart-Agrawala algorithm: it requires *n* messages for entering a CS: (*n*-1) requests and one reply.

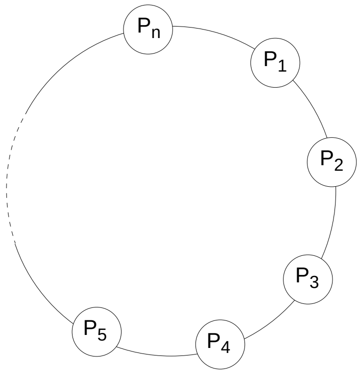
☞ The failure of a process, except the one which holds the token, doesn’t prevent progress.

Token Ring Algorithm

☞ A very simple way to solve mutual exclusion ⇒ arrange the *n* processes P_1, P_2, ... P_n in a logical ring.

☞ The logical ring topology is created by giving each process the address of one other process which is its neighbour in the clockwise direction.

☞ The logical ring topology is unrelated to the physical interconnections between the computers.



Token Ring Algorithm (cont’d)

The algorithm

- The token is initially given to one process.
- The token is passed from one process to its neighbour round the ring.
- When a process requires to enter the CS, it waits until it receives the token from its left neighbour and then it retains it; after it got the token it enters the CS; after it left the CS it passes the token to its neighbour in clockwise direction.
- When a process receives the token but does not require to enter the critical section, it immediately passes the token over along the ring.

Token Ring Algorithm (cont’d)

☞ It can take from 1 to *n*-1 messages to obtain a token. Messages are sent around the ring even when no process requires the token ⇒ additional load on the network.

⇩

The algorithm works well in heavily loaded situations, when there is a high probability that the process which gets the token wants to enter the CS. It works poorly in lightly loaded cases.

☞ If a process fails, no progress can be made until a reconfiguration is applied to extract the process from the ring.

☞ If the process holding the token fails, a unique process has to be picked, which will regenerate the token and pass it along the ring; an *election algorithm* (see later) has to be run for this purpose.

Election

- Many distributed algorithms require one process to act as a coordinator or, in general, perform some special role.

Examples with mutual exclusion

- Central coordinator algorithm: at initialisation or whenever the coordinator crashes, a new coordinator has to be elected (see slide 6).
- Token ring algorithm: when the process holding the token fails, a new process has to be elected which generates the new token (see slide 20).

Election (cont'd)

- We consider that it doesn't matter which process is elected; what is important is that one and only one process is chosen (we call this process the coordinator) and all processes agree on this decision.
- We assume that each process has a unique number (identifier); in general, election algorithms attempt to locate the process with the highest number, among those which currently are up.
- Election is typically started after a failure occurs. The detection of a failure (e.g. the crash of the current coordinator) is normally based on time-out ⇒ a process that gets no response for a period of time suspects a failure and initiates an election process.
- An election process is typically performed in two phases:
  1. Select a leader with the highest priority.
  2. Inform all processes about the winner.

The Bully Algorithm

- A process has to know the identifier of all other processes (it doesn't know, however, which one is still up); the process with the highest identifier, among those which are up, is selected.
- Any process could fail during the election procedure.
- When a process P_i detects a failure and a coordinator has to be elected, it sends an *election message* to all the processes with a higher identifier and then waits for an *answer message*:
  - If no response arrives within a time limit, P_i becomes the coordinator (all processes with higher identifier are down) ⇒ it broadcasts a *coordinator message* to all processes to let them know.
  - If an *answer message* arrives, P_i knows that another process has to become the coordinator ⇒ it waits in order to receive the *coordinator message*. If this message fails to arrive within a time limit (which means that a potential coordinator crashed after sending the *answer message*) P_i resends the *election message*.
- When receiving an *election message* from P_i, a process P_j replys with an *answer message* to P_i and then starts an election procedure itself, unless it has already started one ⇒ it sends an *election message* to all processes with higher identifier.
- Finlly all processes get an *answer message*, except the one which becomes the coordinator.

The Bully Algorithm (cont'd)

The Algorithm

- By default, the state of a process is ELECTION-OFF
- Rule for election process initiator

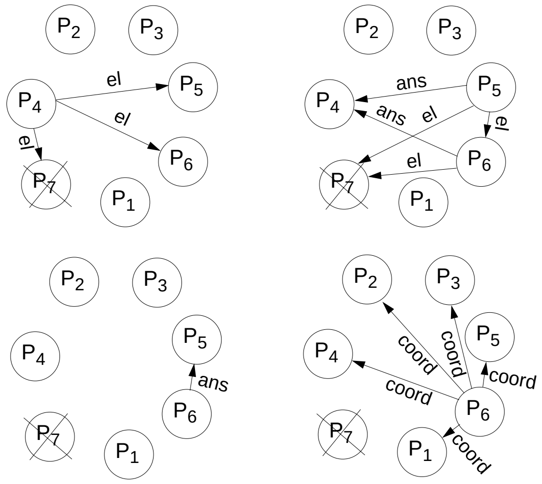
```
/* performed by a process $P_i$, which triggers the
   election procedure, or which starts an election after
   receiving itself an election message */
[RE1]: $state_{P_i}$ := ELECTION-ON.
       $P_i$ sends an election message to all processes
       with a higher identifier.
       $P_i$ waits for answer message.
       if no answer message arrives before time-out then
           $P_i$ is the coordinator and sends a coordinator
                                   message to all processes.
       else
           $P_i$ waits for a coordinator message to arrive.
           if no coordinator message arrives before
                                             time-out then
               restart election procedure according to RE1
           end if
       end if.
```

The Bully Algorithm (cont'd)

☞ Rule for handling an incoming *election message*

```
/* performed by a process $P_i$ at reception of an
   election message coming from $P_j$ */
[RH1]: $P_i$ replies with an answer message to $P_j$.
[RH2]: if $state_{Pi}$ := ELECTION-OFF then
           start election procedure according to RE1
       end if
```

The Bully Algorithm (cont'd)



- If P_6 crashes before sending the coordinator message, P_4 and P_5 restart the election process.

☞ The best case: the process with the second highest identifier notices the coordinator's failure. It can immediately select itself and then send *n*-2 coordinator messages.

☞ The worst case: the process with the lowest identifier initiates the election; it sends *n*-1 election messages to processes which themselves initiate each one an election ⇒ O(n^2) messages.

The Ring-Based Algorithm

☞ We assume that the processes are arranged in a logical ring; each process knows the address of one other process, which is its neighbour in the clockwise direction.

☞ The algorithm elects a single coordinator, which is the process with the highest identifier.

☞ Election is started by a process which has noticed that the current coordinator has failed. The process places its identifier in an *election message* that is passed to the following process.

☞ When a process receives an *election message* it compares the identifier in the message with its own. If the arrived identifier is greater, it forwards the received *election message* to its neighbour; if the arrived identifier is smaller it substitutes its own identifier in the *election message* before forwarding it.

☞ If the received identifier is that of the receiver itself ⇒ this will be the coordinator. The new coordinator sends an *elected message* through the ring.

The Ring-Based Algorithm (cont'd)

☞ An optimization

- Several elections can be active at the same time. Messages generated by later elections should be killed as soon as possible.

⇩

Processes can be in one of two states: *participant* and *non-participant*. Initially, a process is *non-participant*.

- The process initiating an election marks itself *participant*.
- A *participant* process, in the case that the identifier in the *election message* is smaller than the own, does not forward any message (it has already forwarded it, or a larger one, as part of another simultaneously ongoing election).
- When forwarding an *election message*, a process marks itself *participant*.
- When sending (forwarding) an *elected message*, a process marks itself *non-participant*.

The Ring-Based Algorithm (cont'd)

The Algorithm

☞ By default, the state of a process is NON-PARTICIPANT

☞ Rule for election process initiator

```
/* performed by a process $P_i$, which triggers the
   election procedure */
[RE1]: $state_{P_i}$ := PARTICIPANT.
[RE2]: $P_i$ sends an election message with message.id := $i$
       to its neighbour.
```

☞ Rule for handling an incoming *election message*

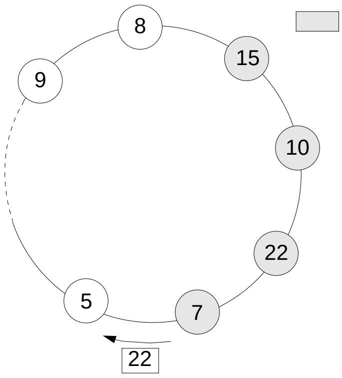
```
/* performed by a process $P_j$, which receives an
   election message */
[RH1]: if message.id > $j$ then
           $P_j$ forwards the received election message.
           $state_{P_j}$ := PARTICIPANT.
       elseif message.id < $j$ then
           if $state_{P_j}$ = NON-PARTICIPANT then
               $P_j$ forwards an election message with
               message.id := $j$.
               $state_{P_j}$ := PARTICIPANT
           end if
       else
           $P_j$ is the coordinator and sends an elected
           message with message.id := $j$ to its neighbour.
           $state_{P_j}$ := NON-PARTICIPANT.
       end if.
```

The Ring-Based Algorithm (cont'd)

☞ Rule for handling an incoming *elected message*

```
/* performed by a process $P_i$, which receives an
   elected message */
[RD1]: if message.id $\neq$ $i$ then
           $P_i$ forwards the received elected message.
           $state_{P_i}$ := NON-PARTICIPANT.
       end if.
```

The Ring-Based Algorithm (cont'd)



▨ *state* = PARTICIPANT

If, in this stage, process 8 initiates another election, the *election message* will be killed by process 15

☞ With one single election started:

- On average: $n/2$ (election) messages needed to reach maximal node; n (election) messages to return to maximal node; n messages to rotate *elected message*.
  Number of messages: $2n + n/2$.
- Worst case: n-1 messages needed to reach maximal node;
  Number of messages: $3n - 1$.

☞ The ring algorithm is more efficient on average then the bully algorithm.

Summary

- In a distributed environment no shared variables (semaphores) and local kernels can be used to enforce mutual exclusion. Mutual exclusion has to be based only on message passing.
- There are two basic approaches to mutual exclusion: non-token-based and token-based.
- The central coordinator algorithm is based on the availability of a coordinator process which handles all the requests and provides exclusive access to the resource. The coordinator is a performance bottleneck and a critical point of failure. However, the number of messages exchanged per use of a CS is small.
- The Ricart-Agrawala algorithm is based on fully distributed agreement for mutual exclusion. A request is multicast to all processes competing for a resource and access is provided when all processes have replied to the request. The algorithm is expensive in terms of message traffic, and failure of any process prevents progress.
- Ricart-Agrawala's second algorithm is token-based. Requests are sent to all processes competing for a resource but a reply is expected only from the process holding the token. The complexity in terms of message traffic is reduced compared to the first algorithm. Failure of a process (except the one holding the token) does not prevent progress.

Summary (cont'd)

- The token-ring algorithm very simply solves mutual exclusion. It is requested that processes are logically arranged in a ring. The token is permanently passed from one process to the other and the process currently holding the token has exclusive right to the resource. The algorithm is efficient in heavily loaded situations.
- For many distributed applications it is needed that one process acts as a coordinator. An election algorithm has to choose one and only one process from a group, to become the coordinator. All group members have to agree on the decision.
- The bully algorithm requires the processes to know the identifier of all other processes; the process with the highest identifier, among those which are up, is selected. Processes are allowed to fail during the election procedure.
- The ring-based algorithm requires processes to be arranged in a logical ring.The process with the highest identifier is selected. On average, the ring-based algorithm is more efficient then the bully algorithm.