Sistemas Distribuídos

Group Communication

António Rui Borges

Summary

- *Characterization of the problem*
- *Access to a shared object in mutual exclusion*
  - *Centralized access permission*
  - *Logic ring*
  - *Total ordering of events*
  - *Minimizing the numbers of messages*
- *Elective procedure*
  - *Election in a logic ring*
  - *Election in an unstructured group*
- *Suggested reading*

Characterization of the problem

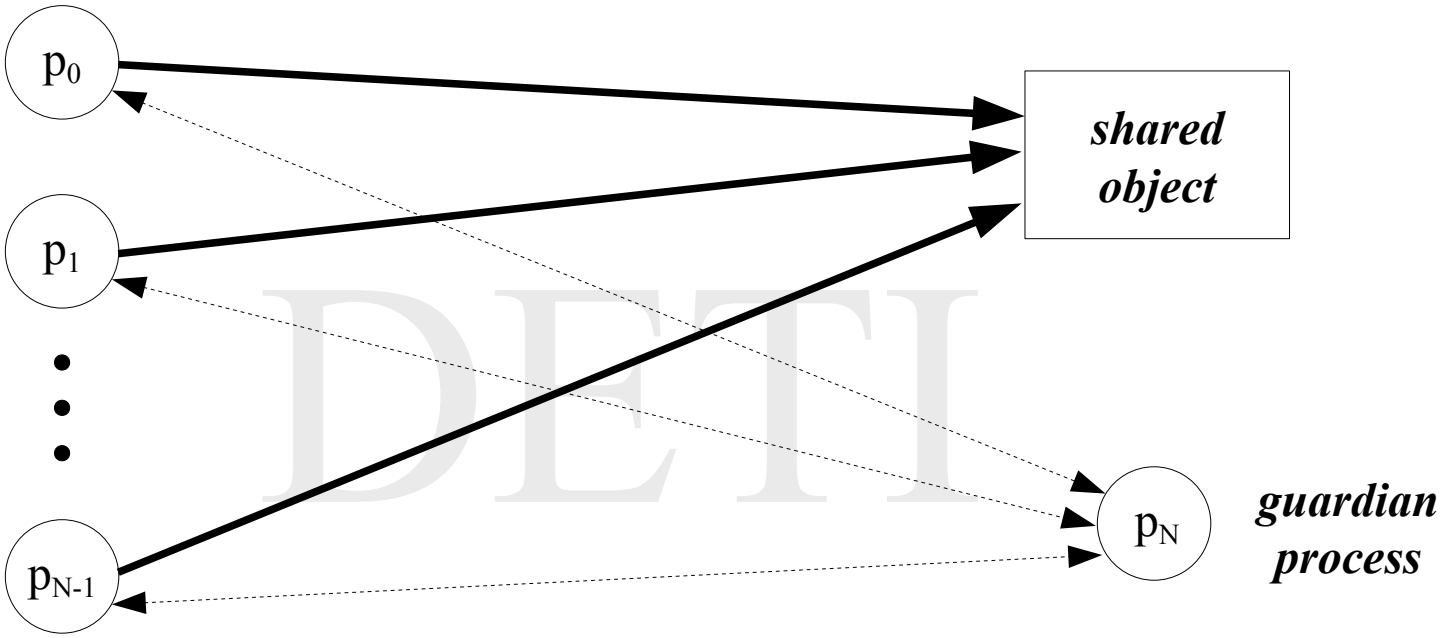
In *group communication*, all involved processes have the same standing. One may assume there is no relevant feature that distinguishes one from the others. We then say we are in a situation of *communication among equals*.

Thus, when they access a common resource, means have to be devised to prevent racing conditions which may lead to information inconsistency. Since they do not share in general an addressing space, synchronization among them must be carried out through message passing.

We may assume for the time being that

- the exchange messages have a finite transmission time, but without an upper limit
- there is no message loss.

Centralized access permission - 1



It is an almost straightforward adaptation of the client-server model with *request serialization*.

There is a special process, the *guardian process*, that keeps track of the accesses to the shared object and allows access to it on per request basis.

Centralized access permission - 2

Protocol

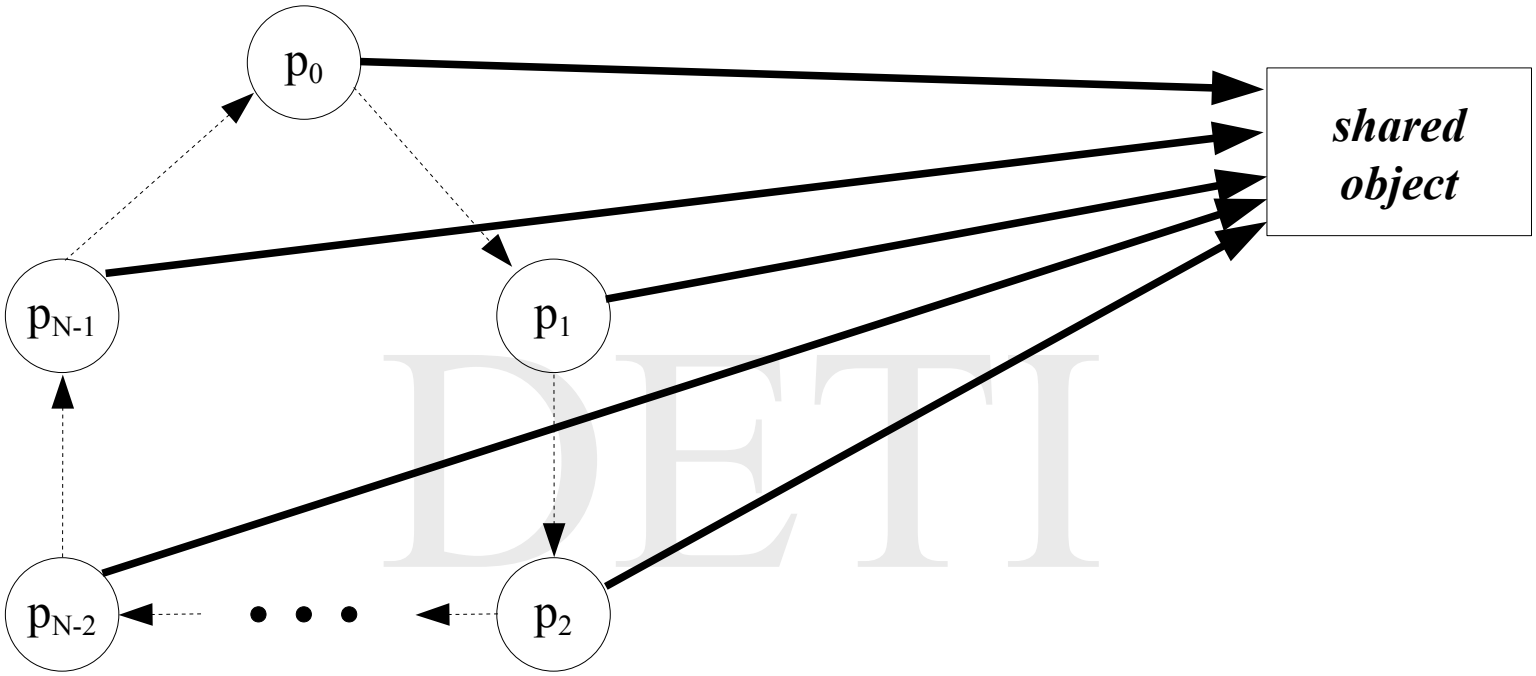
- whenever one of the peer processes p_i, with $i = 0, 1, \cdots, \text{N-1}$, wants access to the shared object, it sends a message of *request access* to p_N , requesting permission, and waits for the reception of a message of *grant access* from it
- if, at the time, there is no other process accessing the shared object, process p_N answers immediately; otherwise, it inserts the request in a waiting queue
- when the message of *grant access* is received, processes p_i may access the shared object
- when access to the shared object is terminated, process p_i sends a message of *release access* to p_N, signaling it no longer requires the object
- if there are pending requests in the waiting queue, process p_N retrieves the first and replies to the referenced process with *grant access*.

Centralized access permission - 3

Comments

- three messages are exchanged per access
- it is not really a true peer-to-peer solution, because it supposes the existence of a special process, the *guardian process*, to control the access to the shared object
- thus, it has a *single point* of failure, if there is a malfunction in process p_N, all the operation comes to a halt.

Logic ring - 1



A restriction is imposed on the processes that form the group: they are organized in a closed loop as far as communication is concerned. Process p_i, with $i = 0, 1, \cdots, N\text{-}1$, can only receive messages from process $p_{(i-1) \bmod N}$ and send messages to process $p_{(i+1) \bmod N}$.

A *token message* is continuously circulated among them. Access to the shared object can only be done by the process that has taken possession of the message.

Logic ring - 2

Protocol

- if the process p_i needs access to the shared object, it waits for the reception of the token message and once it holds it, it accesses the object; upon access termination, it sends the token message to the next process in the ring
- if the process p_i does not need access to the shared object, upon receiving the token message, it sends it immediately to the next process in the ring.

Logic ring - 3

Comments

- one message is always exchanged, either there is access, or not
- it is very efficient if the process group is small
- however, if it is very large, a given process may have to wait a long time to be able to access the object, even if no process is currently doing it.

Total ordering of events - 1

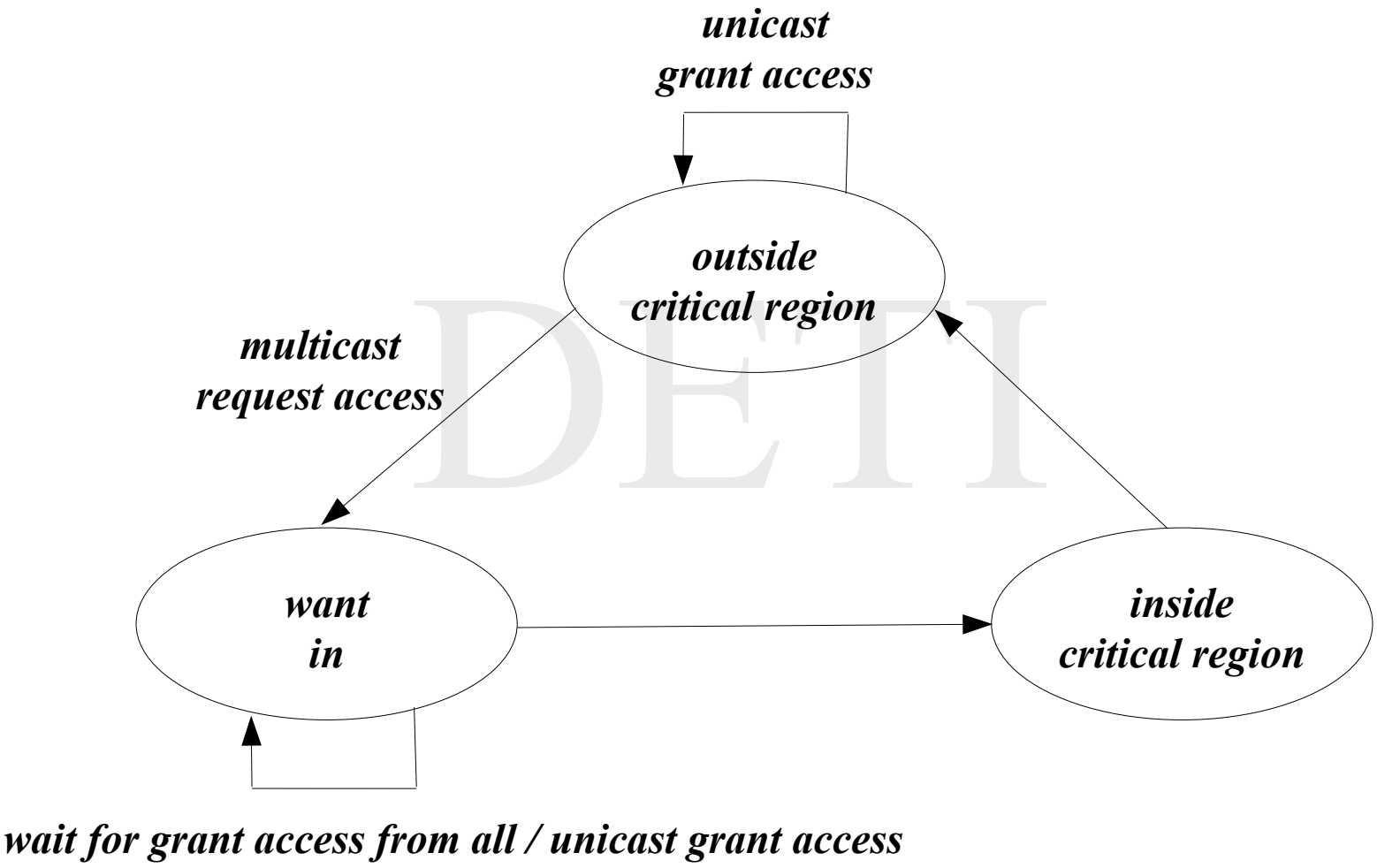
Ricart and Agrawala (1981) have proposed a general method to ensure that N processes access a shared object with mutual exclusion by total ordering their access requests through a Lamport logic clock.

Thus, all processes order in the same way the events associated with the access requests to the shared object and an overall consensus is attained.

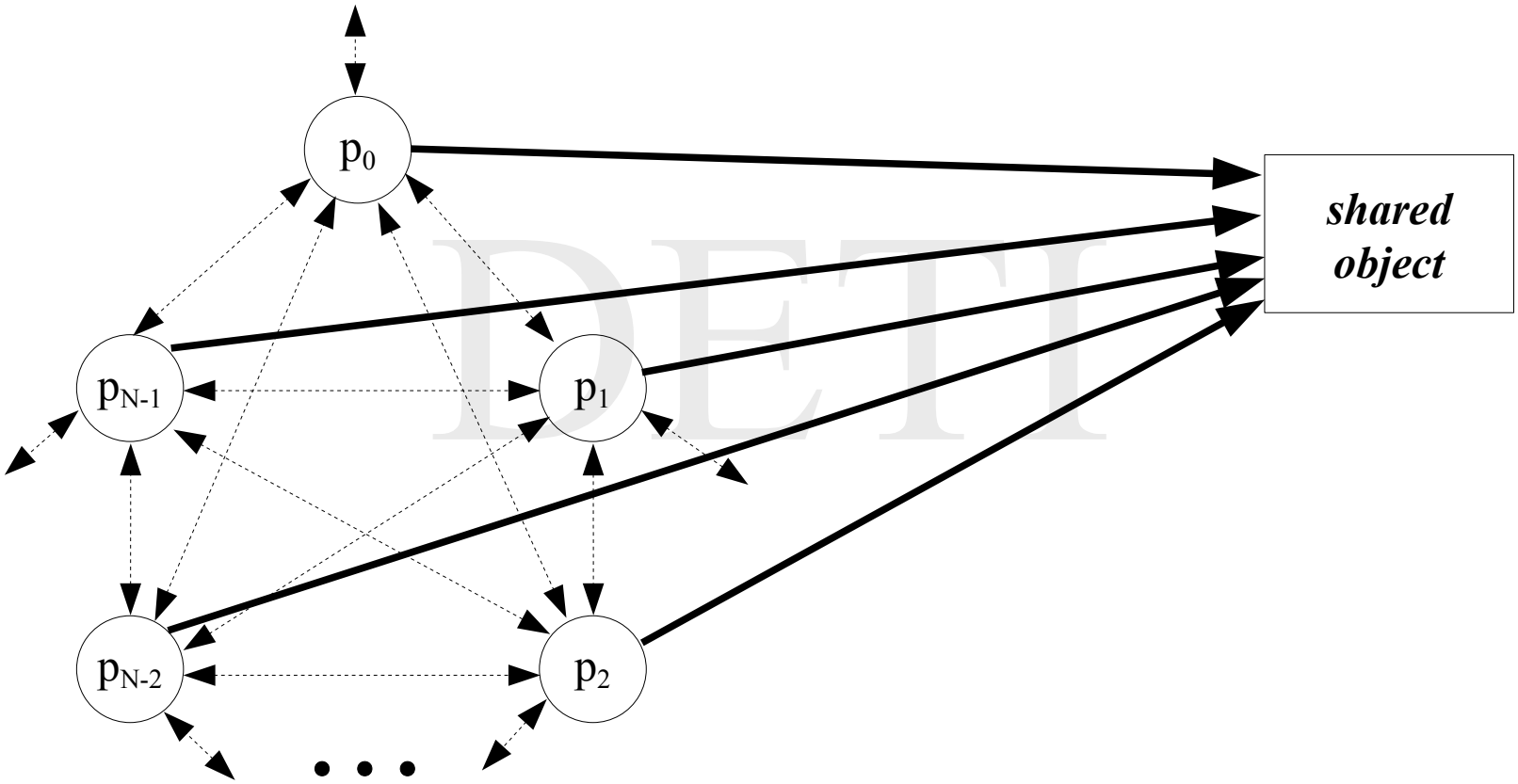
The exchanged messages include the time stamp associated with the sending event, which enables the receiving process, upon message reception, to adjust its local clock according to the rules prescribed by Lamport. Therefore, the potential causality between events can be made explicit.

To generate the total ordering of access request events, an extended time stamp containing the ordered pair $(ts(m_{ar}),id(m_{ar}))$, where $ts(m_{ar})$ is the message time stamp and $id(m_{ar})$ is the sender identification, is constructed.

Total ordering of events - 2



Total ordering of events - 3



Total ordering of events - 4

initialization

```
state = outsideCR;
requestMessageReady = false;
```

p_i enters the critical region

```
state = wantIn;
numberOfRequestsGranted = 0;
myRequestMessage = multicast (requestAccess);
requestMessageReady = true;
wait until (numberOfRequestsGranted == N-1);
state = insideCR;
```

p_i exits the critical region

```
state = outsideCR;
requestMessageReady = false;
while (!empty (requestQueue))
{ id = getId (queueOut (requestQueue));
  unicast (id, accessGranted);
}
```

Total ordering of events - 5

p_i receives an access request message from p_j

```
if (state == outsideCR)
   { id = getId (requestMessage));
     unicast (id, accessGranted);
   }
   else if (state == insideCR)
           queueIn (requestQueue, requestMessage);
           else { wait until (requestMessageReady);
                  if (getExtTimeStamp (myRequestMessage) <
                        getExtTimeStamp (requestMessage))
                     queueIn (requestQueue, requestMessage);
                     else { id = getId (requestMessage));
                            unicast (id, accessGranted);
                          }
                }
```

p_i processes access permissions

```
numberOfRequestsGranted += 1;
```

Total ordering of events - 6

Comments

- 2(N-1) messages are exchanged per access
- it is very efficient if the process group is small
- however, if it is very large, a lot of messages are exchanged.

Minimizing the number of messages – 1

Maekawa (1985) has shown that the access in mutual exclusion to a shared object by any of the peer processes that exist, does not require permission of all of them. The peer processes can be organized in partial groups and only get permission from all the processes belonging to their group.

Mutual exclusion and, therefore, the elimination of racing conditions on access to the shared object are ensured as long as the groups are not mutual exclusive.

The underlying principle is to impose that a process only accesses the shared object if and only if it has permission from all the processes belonging to its group. The permission is given through a voting process.

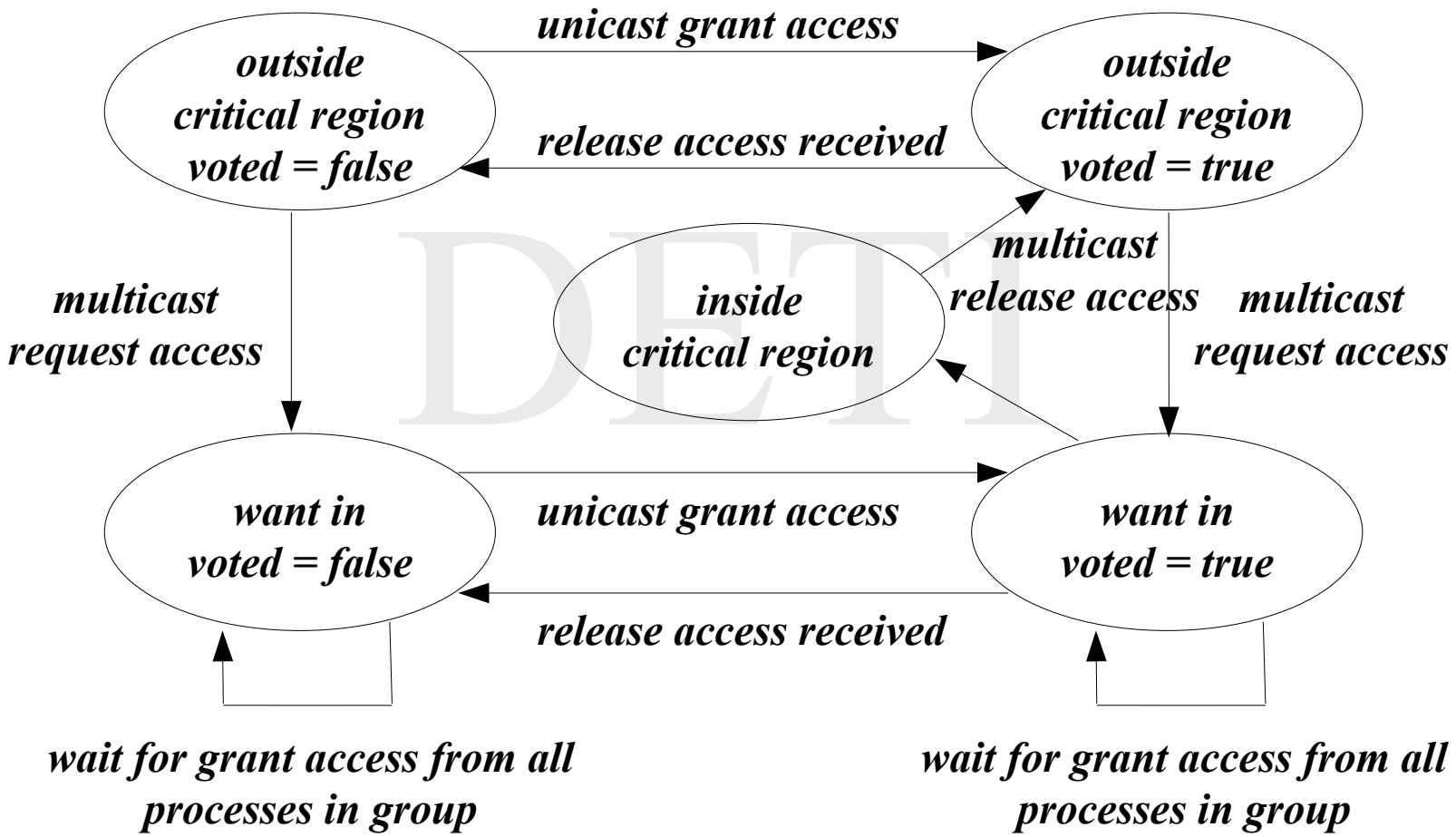
Minimizing the number of messages – 2

Each peer process p_i, with $i = 0, 1, \cdots, N-1$, belong to the *voting group* V_i.

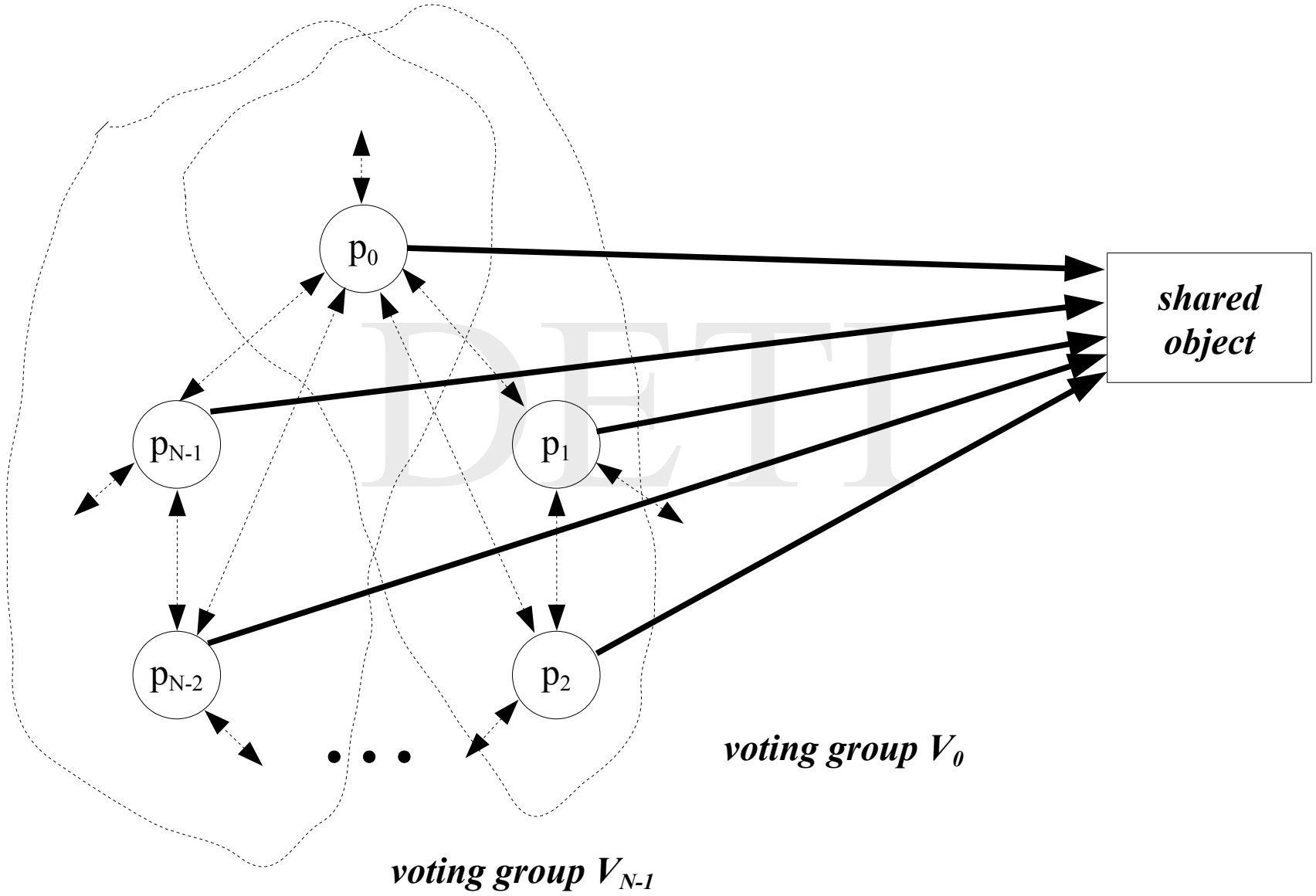
The elements of each voting group are chosen such that

- $\forall_{0 \le i < N} \quad V_i \subseteq \{p_0, p_1, \cdots, p_{N-1}\}$
- $\forall_{0 \le i < N} \quad p_i \in V_i$
- $\forall_{0 \le i, j < N} \quad V_i \cap V_j \neq \emptyset$
- $\forall_{0 \le i, j < N} \quad \#(V_i) \approx \#(V_j)$
- $\exists_{M \in \mathbb{N}} \ \forall_{0 \le i < N} \quad p_i$ belongs to M groups V_* .

Minimizing the number of messages – 3



Minimizing the number of messages – 4



Minimizing the number of messages – 5

The exact determination of the contents of the different voting groups V_i, with $i = 0, 1, \cdots, N-1$, is not a simple procedure. There is, however, an approximation which makes $\#(V_i) \simeq \#(V_j)$, $M \simeq O(\sqrt{N})$ and is trivial.

0	1	$\cdots$	$K-2$	$K-1$
K	$K+1$	$\cdots$	$2K-2$	$2K-1$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$(R-1)K$	$(R-1)K+1$	$\cdots$	$RK-1$	0

$$V_1 = \{0,\ 1,\ \cdots,\ K-1,\ K+1,\ \cdots,(R-1)K+1\}$$

Minimizing the number of messages – 6

initialization

```
state = outsideCR;
voted = false;
```

p_i enters the critical region

```
state = wantIn;
numberOfRequestsGranted = 0;
multicast (requestAccess)to all processes in V_i;
wait until (numberOfRequestsGranted == #(V_i));
state = insideCR;
```

p_i exits the critical region

```
state = outsideCR;
multicast (releaseAccess)to all processes in V_i;
```

Minimizing the number of messages – 7

p_i receives an access request message from p_j

```
if ((state != insideCR) && !voted)
   { id = getId (requestMessage));
     unicast (id, accessGranted);
     voted = true;
   }
   else queueIn (requestQueue, requestMessage);
```

p_i receives a release access message from p_j

```
if (!empty (requestQueue))
   { id = getId (queueOut (requestQueue));
     unicast (id, accessGranted);
     voted = true;
   }
   else voted = false;
```

p_i processes access permissions

```
numberOfRequestsGranted += 1;
```

Minimizing the number of messages – 8

Comments

- 3 $O(\sqrt{N})$ messages are exchanged per access
- it is very efficient when the process group is very large.

Minimizing the number of messages – 9

This algorithm is, however, incorrect! There are instances that, due to the prevailing racing conditions, *deadlock* may occur.

$$V_0 = \{0, 1\}$$
$$V_1 = \{1, 2\}$$
$$V_2 = \{2, 0\}$$

Processes p_0, p_1 e p_2 try to access the shared object about the same time. It may happen that, due to the moments of message arrival, p_0 votes for its own access, p_1 votes for its own access and p_2 votes for its own access. Thus, all processes receive a first permission, but never a second.

How to solve the problem?

Minimizing the number of messages – 10

Saunders (1987) has shown that a possible solution is to total order the access requests to prevent contention. Another one is to deny the condition of *circular waiting*, ordering the access requests by the process identification.

Adapt Maekawa algorithm so that deadlock is prevented!

Elective procedure - 1

There are cases which require the selection a process from a group to carry out a well-defined task at some specific moment. Since all processes are supposed to be conceptually similar and can not be distinguished one from the other, any of them can be in principle selected.

Important points to stress are

- the elective procedure has to be performed in a finite number of steps
- there can be no ambiguity, that is, only one process is selected in the end
- the decision must be consensual, that is, all the involved processes will accept the selection.

Elective procedure - 2

It will be assumed that

- the number of processes in the group is fixed and previously known
- their state at any given time is *in execution* or *catastrophic failure*
- the transmission time of the exchanged messages is finite, but has an upper bound, that is, *time-outs* may be defined
- messages may be lost.

Election in a logic ring - 1

- to begin with, no election is taking place and all processes are in the state of *no participant*
- any process may initiate the elective procedure; it changes its state to *participant* and sends to the next process in the ring a message of type *start election* with its own identification
- when a process receives a message of type *start election*, it carries out the following actions
  - if the process identification in the message is less than its own, it sends the message to the next process in the ring and changes its state to *participant*, if it was not yet changed
  - if the process identification in the message is greater than its own and it is not yet a participant, it replaces the process identification with its own, sends the message to the next process in the ring and changes its state to *participant*; however, if it is already in the state of *participant*, it discards the message to reduce the number of circulating messages
  - if the process identification in the message is equal to its own, then the elective procedure has terminated and the process itself was elected as *leader* (**why?**)

Election in a logic ring - 2

- when a process is elected as *leader*, it sends a message of type *elected* with its own identification
- any process receiving a message of type *elected* carries out the following actions
  - it changes its state to *no participant*
  - if the process identification in the message is different from its own, it saves the identification of the *leader* process and sends the message to the next process in the ring
  - if the process identification in the message is equal to its own, it discards the message leading to the end of the elective procedure.

Election in a logic ring - 3

This algorithm was proposed by Chang e Roberts (1979) and solves the problem if there are no failures.

What should be done if the logic ring requires a dynamic reconfiguration due to

- a process changing its state from *in execution* to *catastrophic failure*, or vice-versa?
- message loss?

Election in an unstructured group – 1

- to begin with, no election is taking place and all processes are in the state of *no participant*
- any process may initiate the elective procedure; it changes its state to *participant* and sends to all the processes having a lower identification number a message of type *start election* with its own identification
- when a process receives a message of type *start election*, it carries out the following actions
  - it replies to the sender process with a message of type *acknowledge*
  - if its state was *no participant*, it changes its state to *participant* and sends to all the processes having a lower identification number a message of type *start election* with its own identification
- when a process receives a message of type *acknowledge*, it waits for a message of type *elected* with the identification of the leader
- if, during a prescribed period of time, a process having the *participant* state does not receive any message of type *acknowledge*, it considers itself the process with the lower identification that is alive and assumes the role of *leader*, it then sends a message of type *elected* with its own identification to all the processes of the group to inform them of the fact.

Election in an unstructured group – 2

This algorithm was proposed by Garcia-Molina (1982) and solves the problem if there are no failures.

What should be done if the unstructured group requires a dynamic reconfiguration due to

- a process changing its state from *in execution* to *catastrophic failure*, or vice-versa?
- message loss?

Suggested reading

- *Distributed Systems: Concepts and Design, 4th Edition, Coulouris*, Dollimore, Kindberg, Addison-Wesley
  - Chapter 12: *Coordination and agreement*
    - Sections 12.1 to 12.3
- *Distributed Systems: Principles and Paradigms, 2nd Edition,* Tanenbaum, van Steen, Pearson Education Inc.
  - Chapter 6: *Synchronization*
    - Sections 6.3 to 6.5