Distributed mutual exclusion

Distributed processes often need to coordinate their activities. I

f a collection of processes share a resource or collection of resources, then often mutual exclusion is required to prevent interference and ensure consistency when accessing the resources.

This is the *critical section* problem, familiar in the domain of operating systems.

In a distributed system, however, neither shared variables nor facilities supplied by a single local kernel can be used to solve it, in general.

We require a solution to *distributed mutual exclusion*: one that is based solely on message passing.

Algorithms for mutual exclusion

We consider a system of N processes p_i, $i = 1, 2, \ldots, N$, that do not share variables. The processes access common resources, but they do so in a critical section. For the sake of simplicity, we assume that there is only one critical section. It is straightforward to extend the algorithms we present to more than one critical section.

We assume that the system is asynchronous, that processes do not fail and that message delivery is reliable, so that any message sent is eventually delivered intact, exactly once.

The application-level protocol for executing a critical section is as follows:

```
enter()              // enter critical section – block if necessary
resourceAccesses()   // access shared resources in critical section
exit()               // leave critical section – other processes may now enter
```

Our essential requirements for mutual exclusion are as follows:

ME1: (safety)	At most one process may execute in the critical section (CS) at a time.
ME2: (liveness)	Requests to enter and exit the critical section eventually succeed.

Condition ME2 implies freedom from both deadlock and starvation. A deadlock would involve two or more of the processes becoming stuck indefinitely while attempting to enter or exit the critical section, by virtue of their mutual interdependence. But even without a deadlock, a poor algorithm might lead to *starvation*: the indefinite postponement of entry for a process that has requested it.

ME3: ($\rightarrow$ ordering) If one request to enter the CS happened-before another, then entry to the CS is granted in that order.

If a solution grants entry to the critical section in happened-before order, and if all requests are related by happened-before, then it is not possible for a process to enter the critical section more than once while another waits to enter. This ordering also allows processes to coordinate their accesses to the critical section. A multi-threaded process may continue with other processing while a thread waits to be granted entry to a critical section. During this time, it might send a message to another process, which consequently also tries to enter the critical section. ME3 specifies that the first process be granted access before the second.

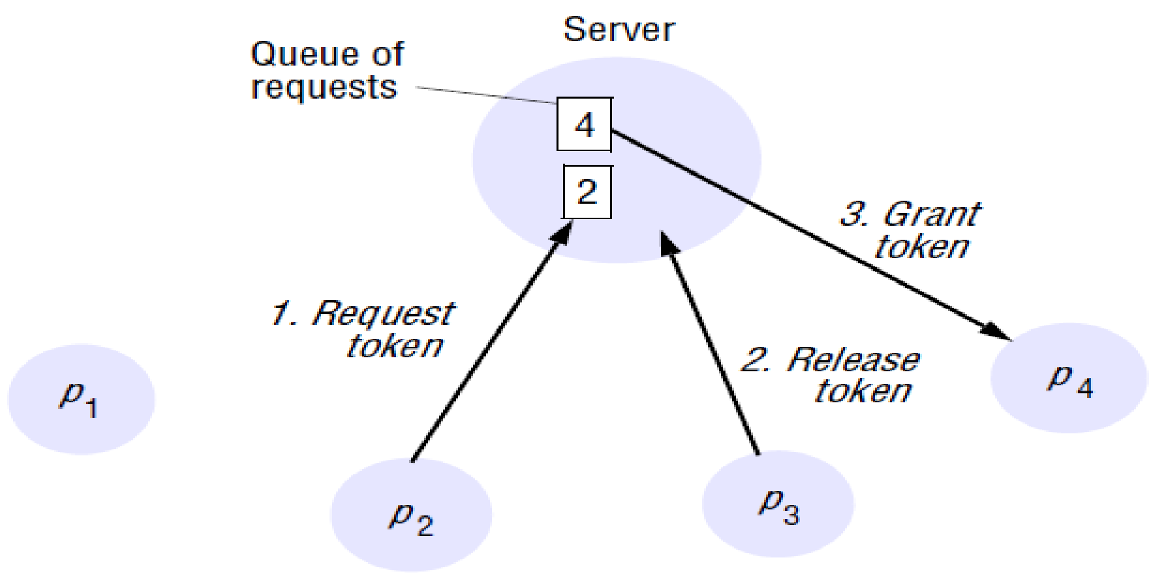
We evaluate the performance of algorithms for mutual exclusion according to the following criteria:

- the *bandwidth* consumed, which is proportional to the number of messages sent in each *entry* and *exit* operation;
- the *client delay* incurred by a process at each *entry* and *exit* operation;
- the algorithm's effect upon the *throughput* of the system. This is the rate at which

The central server algorithm •

The simplest way to achieve mutual exclusion is to employ a server that grants permission to enter the critical section.

Figure 15.2 shows the use of this server. To enter a critical section, a process sends a request message to the server and awaits a reply from it. Conceptually, the reply constitutes a token signifying permission to enter the critical section. If no other process has the token at the time of the request, then the server replies immediately, granting the token. If the token is currently held by another process, then the server does not reply, but queues the request. When a process exits the critical section, it sends a message to the server, giving it back the token.



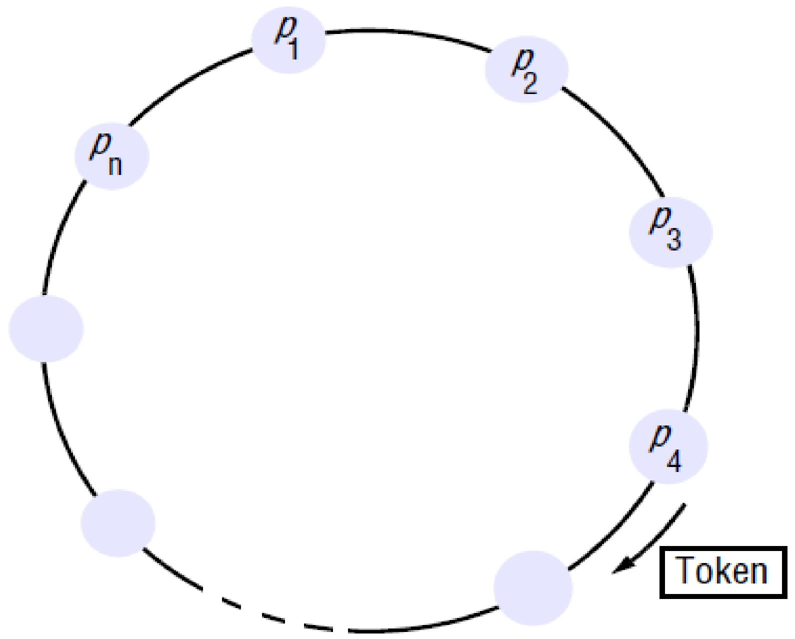
The server may become a performance bottleneck for the system as a whole.
The synchronization delay is the time taken for a round-trip: a *release* message to the server, followed by a *grant* message to the next process to enter the critical section

A ring-based algorithm • One of the simplest ways to arrange mutual exclusion between the N processes without requiring an additional process is to arrange them in a logical ring. This requires only that each process p_i has a communication channel to the next process in the ring, $p_{(i+1) mod N}$. The idea is that exclusion is conferred by obtaining a token in the form of a message passed from process to process in a single direction –

If a process does not require to enter the critical section when it receives the token, then it immediately forwards the token to its neighbour. A process that requires the token waits until it receives it, but retains it. To exit the critical section, the process sends the token on to its neighbour.

The arrangement of processes is shown in Figure 15.3. It is straightforward to verify that the conditions ME1 and ME2 are met by this algorithm, but that the token is not necessarily obtained in happened-before order. (Recall that the processes may exchange messages independently of the rotation of the token.)

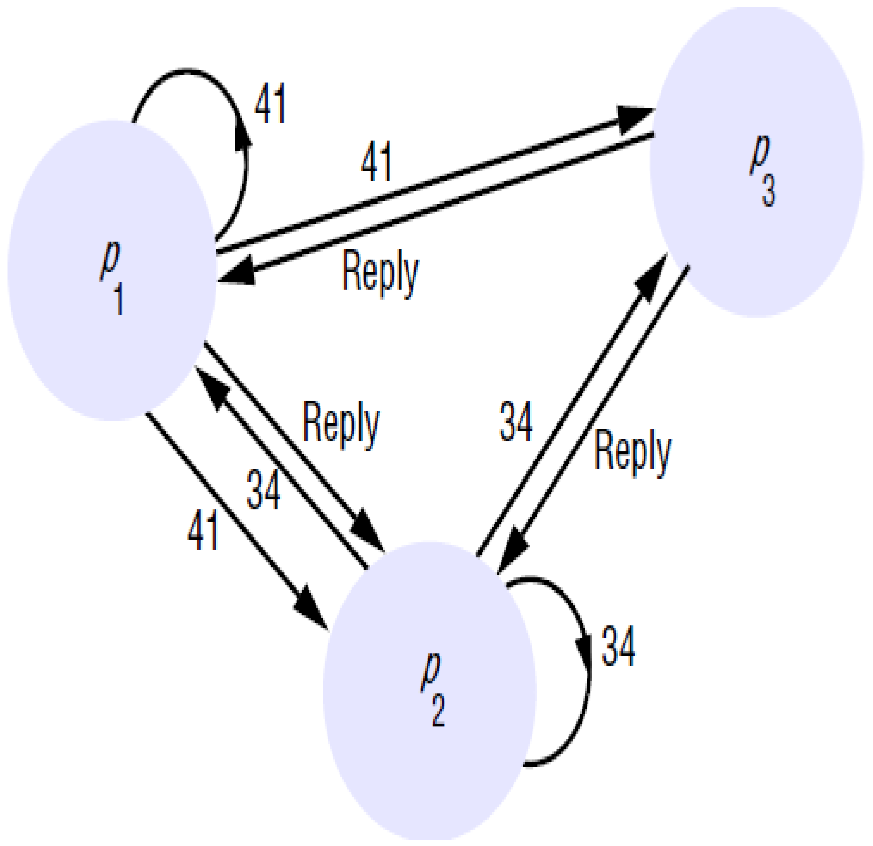
A ring of processes transferring a mutual exclusion token



This algorithm continuously consumes network bandwidth (except when a process is inside the critical section): the processes send messages around the ring even when no process requires entry to the critical section. The delay experienced by a process requesting entry to the critical section is between 0 messages (when it has just received the token) and *N* messages (when it has just passed on the token).

An algorithm using multicast and logical clocks • Ricart and Agrawala [1981] developed an algorithm to implement mutual exclusion between N peer processes that is based upon multicast. The basic idea is that processes that require entry to a critical section multicast a request message, and can enter it only when all the other processes have replied to this message. The conditions under which a process replies to a request are designed to ensure that conditions ME1–ME3 are met.

The processes $p_1, p_2, ..., p_N$ bear distinct numeric identifiers. They are assumed to possess communication channels to one another, and each process p_i keeps a Lamport clock, updated according to the rules LC1 and LC2 of Section 14.4. Messages requesting entry are of the form $<T, p_i>$, where T is the sender's timestamp and p_i is the sender's identifier.



Ricart and Agrawala's algorithm

```
On initialization
    state := RELEASED;

To enter the section
    state := WANTED;                                  ⎫
    Multicast request to all processes;               ⎬ Request processing deferred here
    T := request's timestamp;                         ⎭
    Wait until (number of replies received = $(N - 1)$);
    state := HELD;

On receipt of a request $<T_i, p_i>$ at $p_j$ $(i \neq j)$
    if (state = HELD or (state = WANTED and $(T, p_j) < (T_i, p_i)$))
    then
                queue request from $p_i$ without replying;
    else
                reply immediately to $p_i$;
    end if

To exit the critical section
    state := RELEASED;
    reply to any queued requests;
```

Each process records its state of being outside the critical section (*RELEASED*), wanting entry (*WANTED*) or being in the critical section (*HELD*) in a variable *state*. The protocol is given in Figure 15.4.
If a process requests entry and the state of all other processes is *RELEASED*, then all processes will reply immediately to the request and the requester will obtain entry. If some process is in the state *HELD*, then that process will not reply to requests until it has finished with the critical section, and so the requester cannot gain entry in the meantime.
If two or more processes request entry at the same time, then whichever process's request bears the lowest timestamp will be the first to collect $N - 1$ replies, granting it entry next.

The advantage of this algorithm is that its synchronization delay is only one message transmission time. Both the previous algorithms incurred a round-trip synchronization delay.

Maekawa's voting algorithm • Maekawa [1985] observed that in order for a process to enter a critical section, it is not necessary for all of its peers to grant it access. Processes need only obtain permission to enter from *subsets* of their peers, as long as the subsets used by any two processes overlap. We can think of processes as voting for one another to enter the critical section. A 'candidate' process must collect sufficient votes to enter. Processes in the intersection of two sets of voters ensure the safety property ME1, that at most one process can enter the critical section, by casting their votes for only one candidate.

Maekawa associated a *voting set* V_i with each process p_i $(i = 1, 2, \ldots, N)$, where $V_i \subseteq \{p_1, p_1, \ldots, p_N\}$. The sets V_i are chosen so that, for all $i, j = 1, 2, \ldots, N$:

- $p_i \in V_i$
- $V_i \cap V_j \neq \varnothing$ – there is at least one common member of any two voting sets
- $|V_i| = K$ – to be fair, each process has a voting set of the same size
- Each process p_j is contained in M of the voting sets V_i.

Maekawa showed that the optimal solution, which minimizes K and allows the processes to achieve mutual exclusion, has $K \sim \sqrt{N}$ and $M = K$ (so that each process is in as many of the voting sets as there are elements in each one of those sets). It is non-

Figure 15.6 Maekawa's algorithm

```
On initialization
    state := RELEASED;
    voted := FALSE;

For $p_i$ to enter the critical section
    state := WANTED;
    Multicast request to all processes in $V_i$;
    Wait until (number of replies received = $K$);
    state := HELD;

On receipt of a request from $p_i$ at $p_j$
    if (state = HELD or voted = TRUE)
    then
                queue request from $p_i$ without replying;
    else
                send reply to $p_i$;
                voted := TRUE;
    end if

For $p_i$ to exit the critical section
    state := RELEASED;
    Multicast release to all processes in $V_i$;

On receipt of a release from $p_i$ at $p_j$
    if (queue of requests is non-empty)
    then
                remove head of queue – from $p_k$, say;
                send reply to $p_k$;
                voted := TRUE;
    else
                voted := FALSE;
    end if
```

Fault tolerance • The main points to consider when evaluating the above algorithms with respect to fault tolerance are:

- What happens when messages are lost?
- What happens when a process crashes?

None of the algorithms that we have described would tolerate the loss of messages, if the channels were unreliable.

The ring-based algorithm cannot tolerate a crash failure of any single process.

As it stands, Maekawa's algorithm can tolerate some process crash failures: if a crashed process is not in a voting set that is required, then its failure will not affect the other processes.

The central server algorithm can tolerate the crash failure of a client process that neither holds nor has requested the token.

The Ricart and Agrawala algorithm as we have described it can be adapted to tolerate the crash failure of such a process, by taking it to grant all requests implicitly.

15.4 In the central server algorithm for mutual exclusion, describe a situation in which two requests are not processed in happened-before order.

15.4 Ans. Process A sends a request Ra for entry then sends a message m to B. On receipt of m, B sends request Rb for entry. To satisfy happened-before order, Ra should be granted before Rb. However, due to the vagaries of message propagation delay, Rb arrives at the server before Ra, and they are serviced in the opposite order.

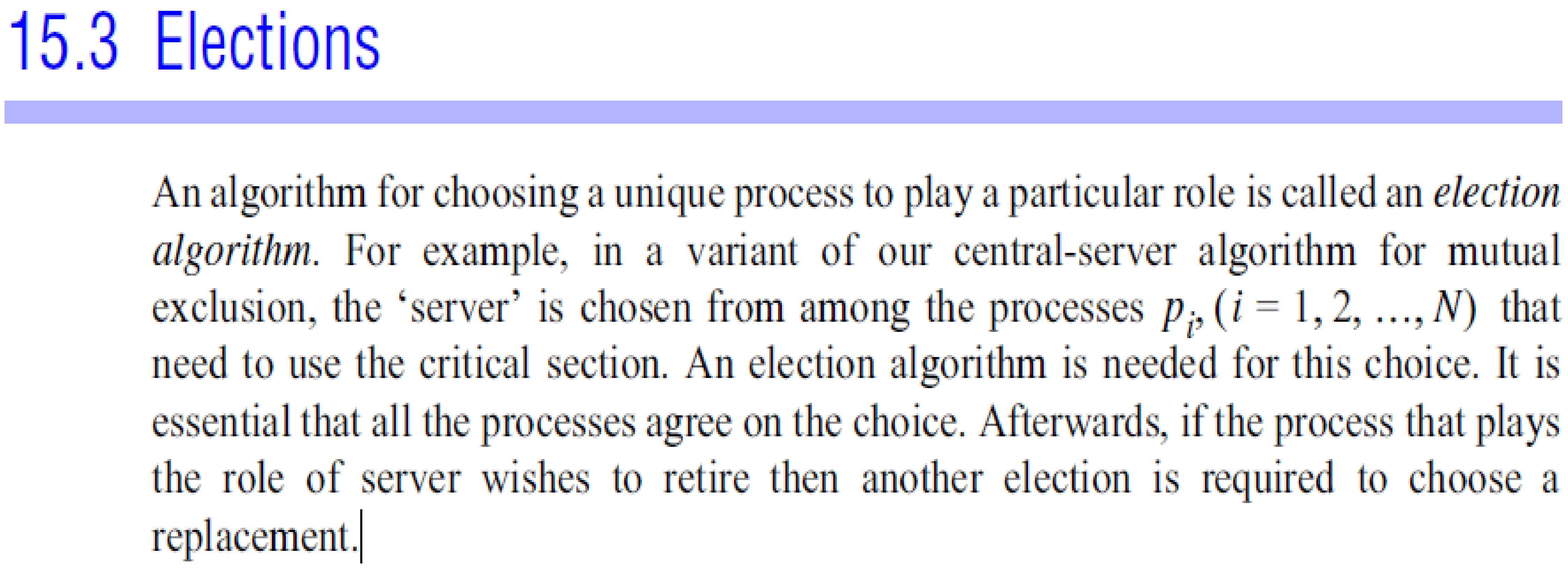
15.5 Adapt the central server algorithm for mutual exclusion to handle the crash failure of any client (in any state), assuming that the server is correct and given a reliable failure detector. Comment on whether the resultant system is fault-tolerant. What would happen if a client that possesses the token is wrongly suspected to have failed?

11.5 Ans. The server uses the reliable failure detector to determine whether any client has crashed. If the client has been granted the token then the server acts as if the client had returned the token.
In case it subsequently receives the token from the client (which may have sent it before crashing), it ignores it. The resultant system is not fault-tolerant.
If a token-holding client crashed then the application-specific data protected by the critical section (whose consistency is at stake) may be in an unknown state at the point when another client starts to access it.
If a client that possesses the token is wrongly suspected to have failed then there is a danger that two processes will be allowed to execute in the critical section concurrently.

15.7 In a certain system, each process typically uses a critical section many times before another process requires it. Explain why Ricart and Agrawala’s multicast-based mutual exclusion algorithm is inefficient for this case, and describe how to improve its performance. Does your adaptation satisfy liveness condition ME2?
15.7 Ans: In Ricart and Agrawala’s multicast-based mutual exclusion algorithm, a client issues a multicast request every time it requires entry. This is inefficient in the case described, of a client that repeatedly enters the critical section before another needs entry. Instead, a client that finishes with a critical section and which has received no outstanding requests could mark the token as JUST_RELEASED, meaning that it has not conveyed any information to other processes that it has finished with the critical section.
If the client attempts to enter the critical section and finds the token to be JUST_RELEASED, it can change the state to HELD and re-enter the critical section.
To meet liveness condition ME2, a JUST_RELEASED token should become RELEASED if a request for entry is received.

Give an example execution of the ring-based algorithm to show that processes are not necessarily granted entry to the critical section in happened-before order.

Ans: The token may be grabbed by B first then A. However, A happens before B.



15.3 Elections

An algorithm for choosing a unique process to play a particular role is called an *election algorithm*. For example, in a variant of our central-server algorithm for mutual exclusion, the 'server' is chosen from among the processes p_i, ($i = 1, 2, \ldots, N$) that need to use the critical section. An election algorithm is needed for this choice. It is essential that all the processes agree on the choice. Afterwards, if the process that plays the role of server wishes to retire then another election is required to choose a replacement.

Each process p_i ($i = 1, 2, \ldots, N$) has a variable $elected_i$, which will contain the identifier of the elected process. When the process first becomes a participant in an election it sets this variable to the special value '$\perp$' to denote that it is not yet defined.

Our requirements are that, during any particular run of the algorithm:

E1: (safety)	A participant process p_i has $elected_i = \perp$ or $elected_i = P$, where P is chosen as the non-crashed process at the end of the run with the largest identifier.
E2: (liveness)	All processes p_i participate and eventually either set $elected_i \neq \perp$ – or crash.

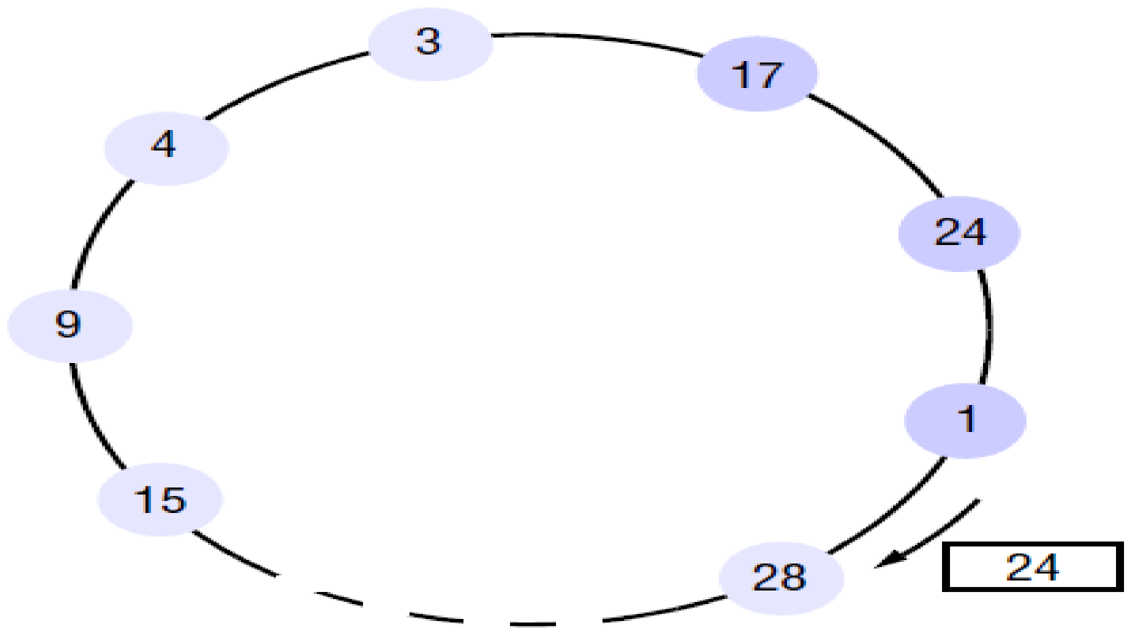
Note that there may be processes p_j that are not yet participants, which record in $elected_j$ the identifier of the previous elected process.

A ring-based election algorithm • The algorithm of Chang and Roberts [1979] is suitable for a collection of processes arranged in a logical ring. Each process p_i has a communication channel to the next process in the ring, $p_{(i+1) mod N}$, and all messages are sent clockwise around the ring. We assume that no failures occur, and that the system is asynchronous. The goal of this algorithm is to elect a single process called the *coordinator*, which is the process with the largest identifier.

Initially, every process is marked as a *non-participant* in an election. Any process can begin an election. It proceeds by marking itself as a *participant*, placing its identifier in an *election* message and sending it to its clockwise neighbour.

When a process receives an *election* message, it compares the identifier in the message with its own. If the arrived identifier is greater, then it forwards the message to its neighbour. If the arrived identifier is smaller and the receiver is not a *participant*, then it substitutes its own identifier in the message and forwards it; but it does not forward the message if it is already a *participant*. On forwarding an *election* message in any case, the process marks itself as a *participant*.

A ring-based election in progress



Note: The election was started by process 17. The highest process identifier encountered so far is 24. Participant processes are shown in a darker tint.

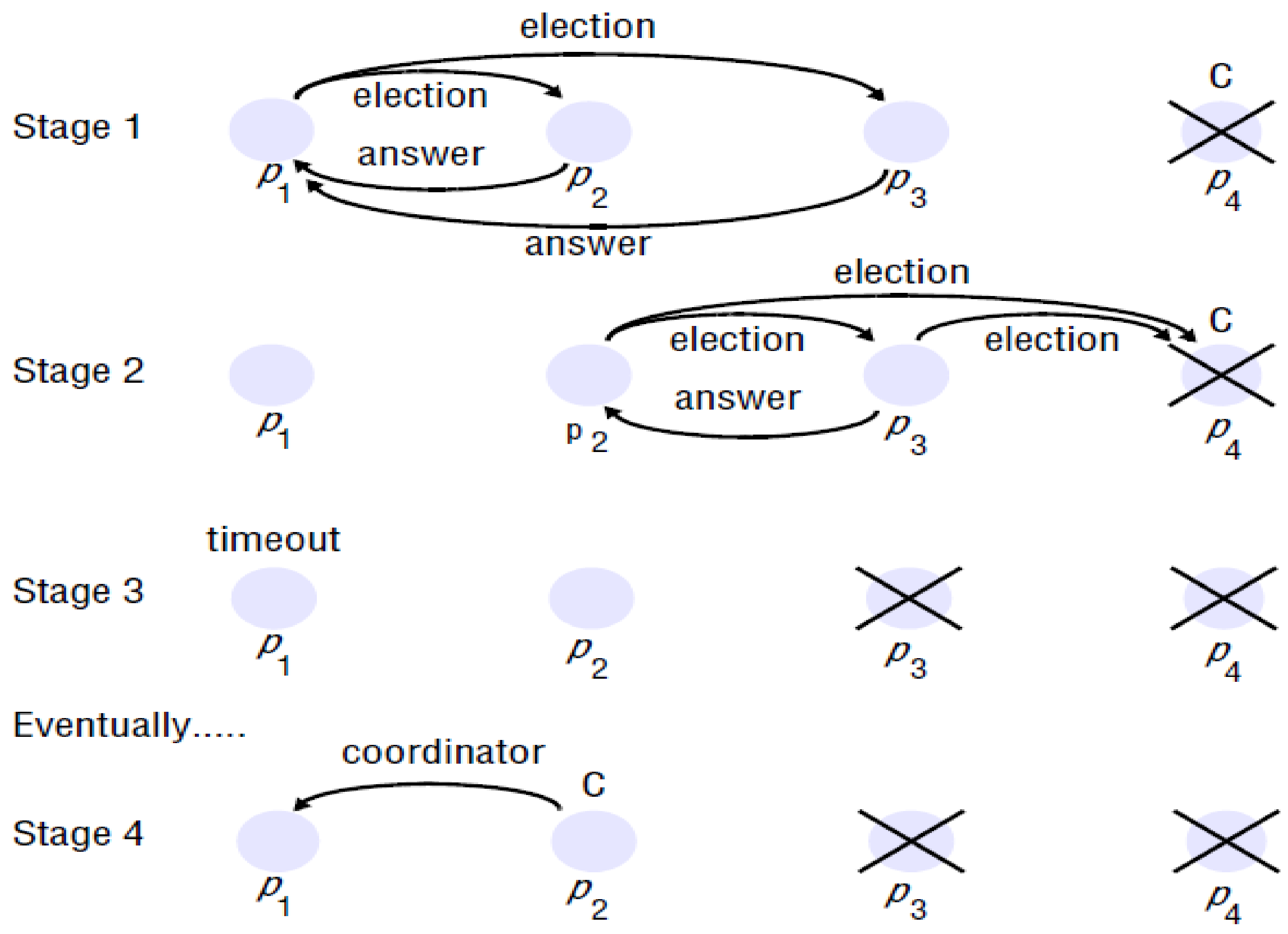
If, however, the received identifier is that of the receiver itself, then this process's identifier must be the greatest, and it becomes the coordinator. The coordinator marks itself as a *non-participant* once more and sends an *elected* message to its neighbour, announcing its election and enclosing its identity.

When a process p_i receives an *elected* message, it marks itself as a *non-participant*, sets its variable $elected_i$ to the identifier in the message and, unless it is the new coordinator, forwards the message to its neighbour.

The bully algorithm • The bully algorithm [Garcia-Molina 1982] allows processes to crash during an election, although it assumes that message delivery between processes is reliable. Unlike the ring-based algorithm, this algorithm assumes that the system is synchronous: it uses timeouts to detect a process failure. Another difference is that the ring-based algorithm assumed that processes have minimal *a priori* knowledge of one another: each knows only how to communicate with its neighbour, and none knows the identifiers of the other processes. The bully algorithm, on the other hand, assumes that each process knows which processes have higher identifiers, and that it can communicate with all such processes.

There are three types of message in this algorithm: an *election* message is sent to announce an election; an *answer* message is sent in response to an election message and a *coordinator* message is sent to announce the identity of the elected process – the new

The bully algorithm



The election of coordinator p_2, after the failure of p_4 and then p_3

The process that knows it has the highest identifier can elect itself as the coordinator simply by sending a *coordinator* message to all processes with lower identifiers. On the other hand, a process with a lower identifier can begin an election by sending an *election* message to those processes that have a higher identifier and awaiting *answer* messages in response. If none arrives within time T, the process considers itself the coordinator and sends a *coordinator* message to all processes with lower identifiers announcing this. Otherwise, the process waits a further period T' for a *coordinator* message to arrive from the new coordinator. If none arrives, it begins another election.

If a process p_i receives a *coordinator* message, it sets its variable $elected_i$ to the identifier of the coordinator contained within it and treats that process as the coordinator.

If a process receives an *election* message, it sends back an *answer* message and begins another election – unless it has begun one already.

When a process is started to replace a crashed process, it begins an election. If it has the highest process identifier, then it will decide that it is the coordinator and announce this to the other processes. Thus it will become the coordinator, even though the current coordinator is functioning. It is for this reason that the algorithm is called the 'bully' algorithm.

The operation of the algorithm is shown in Figure 15.8. There are four processes, $p_1 - p_4$. Process p_1 detects the failure of the coordinator p_4 and announces an election (stage 1 in the figure). On receiving an *election* message from p_1, processes p_2 and p_3 send *answer* messages to p_1 and begin their own elections; p_3 sends an *answer* message to p_2, but p_3 receives no *answer* message from the failed process p_4 (stage 2). It therefore decides that it is the coordinator. But before it can send out the *coordinator* message, it too fails (stage 3). When p_1's timeout period T' expires (which we assume occurs before p_2's timeout expires), it deduces the absence of a *coordinator* message and begins another election. Eventually, p_2 is elected coordinator (stage 4).

11.8 In the Bully algorithm, a recovering process starts an election and will become the new coordinator if it has a higher identifier than the current incumbent. Is this a necessary feature of the algorithm?

11.8 Ans.

First note that this is an undesirable feature if there is no advantage to using a higher-numbered process: the re-election is wasteful. However, the numbering of processes may reflect their relative advantage (for example, with higher-numbered processes executing at faster machines). In this case, the advantage may be worth the re-election costs. Re-election costs include the message rounds needed to implement the election; they also may include application-specific state transfer from the old coordinator to the new coordinator.
To avoid a re-election, a recovering process could merely send a *requestStatus* message to successive lower-numbered processes to discover whether another process is already elected, and elect itself only if it receives a negative response. Thereafter, the algorithm can operate as before: if the newly-recovered process discovers the coordinator to have failed, or if it receives an *election* message, it sends a *coordinator* message to the remaining processes.