2

ELECTION AND MUTUAL EXCLUSION ALGORITHMS

1 INTRODUCTION

A problem that often arises in connection with communicating processes that co-operate to achieve a common goal is that of deciding, at any given instant, to which of the processes a particular privilege should be assigned. The requirement can take either of two forms according to the nature of the goal:

(a) bringing about a mutual exclusion, when the choice of the privileged process must be made equitably and that process must not hold the privilege for an indefinite period;
(b) election by all the processes of one to which the privilege shall be assigned permanently.

We study these two aspects of the problem in this chapter.

2 THE MUTUAL EXCLUSION PROBLEM

This is one of the first problems met in parallel programming: the several processes, operating in parallel, compete for resources that cannot be shared and therefore constraints have to be applied to ensure that when one process is using such a resource none of the others can gain access to it. The first software solution to this

problem of mutual exclusion was due to Dekker [DIJ 65] and used only the primitive operations of reading and writing a word in memory; many more have been proposed since then, differing in their properties (e.g. equitable or not) and in the number and the nature of the variables they need.

In a distributed system the specification of a mutual exclusion algorithm cannot be made to depend on access to central memory but must be in terms of message exchanges; the protocol must provide both equitability and freedom from mutual blocking, that is, any process that wishes to enter the critical section must be able to do so within a finite time.

In an earlier book [RAY 86] I have dealt exclusively with this problem and its solution and a number of algorithms are given. We must recall here the distinction made between distributed algorithms based on the use of state variables (such a variable being 'distributed' in the sense that at any instant it can be read by any process but written to by only a designated one) and those based on communication of messages. The activities involving the distributed variables of the algorithms of the first class can of course be expressed in terms of the sending and receiving of messages, but the effect of this is to mix the logic of the algorithm with that of its implementation, and consequently to obscure the presentation and make understanding more difficult. Such algorithms therefore are best suited to distributing control in a centralized architecture, resulting in reliable algorithms that are resilient to failures — properties not possessed by centralized algorithms.

We shall study here distributed algorithms based on messages. [RAY 86] gives a number of these: Lamport's algorithm [LAM 78] which uses a distributed queue, a full account of which is given in [PLO 84], in Ada; Lelann's [LEL 77], using a token circulating round a ring; Ricart and Agrawala's [RIC 81], which aims to minimize the number of messages; and that of Carvallo and Roucairol [CAR 83] which also seeks to minimize the number of messages but uses a different definition of symmetry. We shall not deal with these but will give a mutual exclusion algorithm that is described in two separate publications ([RIC 83] and [SUZ 82]); its importance is that it needs only n messages or none at all to bring about the desired exclusion in a set of n processes.

3 THE RICART AND AGRAWALA/SUZUKI KASAMI ALGORITHM

3.1 Overview of other algorithms

Lamport's algorithm [LAM 78] needs $3(n - 1)$ messages to ensure mutual exclusion in a set of n processes: $n-1$ each for distributing the request for access to the critical section by one of the processes, for conveying the agreement to this of the other $(n-1)$ and for broadcasting the news when the critical section becomes free again. Ricart and Agrawala's algorithm [RIC 81] needs only $2(n - 1)$; $n-1$ for a process P_i to inform the others of its wish to enter the critical section, and $n-1$ for conveying the agreements, as before; no explicit messages are sent to say that this section is free, any process that was using it being deemed to have released it when agreeing to its use by another. Carvallo and Roucairol's algorithm [CAR 83] can need any number between zero and $2(n - 1)$, using a different definition of symmetry to optimize Ricart and Agrawala's [CAR 83, RIC 83]: if a process P_i has used the critical section and asks for access again after a period during which no other process has asked for access, then P_i considers that it has permanent access and does not make a further request when it next needs access; this enables the number of messages to be reduced.

All these algorithms ensure the desired exclusion, are equitable and prevent mutual blocking by the processes; the equitability results from the use of time-stamping, which enables an ordering to be established among the messages and any conflicts to be resolved. All these properties are discussed by Raynal [RAY 86].

3.2 Assumptions

We now describe the algorithm devised by Ricart and Agrawala as a result of their attempt to reduce the number of messages needed. It makes the following assumptions about the communication network:

(a) The network is fully linked.
(b) The transmission is error-free.

(c) The delay is variable.
(d) Desequencing is possible — i.e. messages may be received in an order different from that in which they were sent.

3.3 Principles of the algorithm

The privilege that allows a process to enter the critical section is represented by a *token*, which must be held by any process using this section; and any process that holds the token can enter this section without requesting permission of the other processes (cf. [CAR 83]). Initially the token is assigned arbitrarily to one of the processes; a process P_i ($i = 1,2 \ldots, n$) that wishes to use the critical section will not know which of the other processes holds the token at that instant and will request it by issuing a message that is time-stamped and broadcast to all the others. The essential difference between this algorithm and those just described lies in the definition and implementation of the token: this takes the form of an object manipulated by the processes, consisting of an array whose kth element records the time-stamp of the last time it was assigned to the process P_k. When the process holding the token, P_j say, no longer needs to use the critical section it searches this array in the order $j+1$, $j+2$, . . ., n, 1, 2, . . ., $j-1$ for the first value of l such that the time-stamp of P_l's last request for the token is greater than the value recorded in the token for the time-stamp of P_l's last holding of the token. P_j then transfers the token to P_l.

3.4 The algorithm

The following variables are declared in each process P_i:

clock: 0, 1, . . . **initialized** 0; (logical clock)
token_present: **boolean**;
token_held: **boolean initialize** F;
token: **array** (1, 2, . . ., n) **of** (0, 1, . . .) **initialized** $\emptyset$;
requests: **array** (1, 2, . . ., n) **of** (0, 1, . . .) **initialized** $\emptyset$;

The boolean 'token_present' is initialized to F (false) in every process except one, this one holding the token at the start; the texts of all the processes are identical at the start, each identified as i.

The operation **wait** (access, token) causes the process to wait until a message of the type 'access' is received, which is then put into the variable 'token'; after such a wait other request messages can be received and processed.

The algorithm is in two parts; the first part deals with the use of the critical section and consists of a prelude, followed by the critical section and ending with a postlude, the second part deals with the actions to be performed when messages are received. The texts are as follows, where the notation is that of [RAY 86].

```
if ¬ token_present then begin clock ← clock + 1;                    [Prelude]
                              broadcast (requests, clock, i);
                              wait (access, token);
                              token_present ← T;
                        end;
end if;
token_held ← T;
     ⟨critical section⟩
token (i) ← clock;                                                   [Postlude]
token_held ← F;
for j from i+1 to n, 1 to i−1 do
                  if request (j) > token (j) ∧ token_present then
    begin token_present ← F;
        send (access, token) j;
                   end;
                  end if;

when received (request, k, j) do
                  request (j) ← max(request (j), k);
                  if token_present ∧ ¬ token_held then
                     ⟨text of postlude⟩
                  end if;
end do;
```

Notes: 1. If the token is not present, the prelude consists in broadcasting the request to all the other processes and waits until the token arrives.

2. The postlude first records in the token the time of its last holding by P_i, looks to see if any process has requested the token and if so transfers it appropriately.

3. The receipt of a request from P_j has the effect of updating the local variable 'request (j)' which records the time of P_j's last request, followed by the transfer of the token if it is neither held nor being used by any other process.

3.5 Proof of the algorithm

Mutual exclusion

Proving that this is ensured is equivalent to showing that at any one time the maximum number of variables token_present that can have the value T (true) is 1; and since this is the initial value it suffices to show that the value is conserved throughout the procedure.

Consider first the prelude. The variable for P_i, which we write token_present$_i$, changes its value from F to T when P_i receives the token. If we now consider the postlude for the process P_j that has issued the token, we see that P_j has been able to do so only if token_present $_j$ had the value T and P_j had changed this to F before sending the token. This establishes the exclusion property.

Notice that all the token_present variables have the value F when and only when the token is in the process of being transferred.

Fairness and absence of deadlock

The absence of deadlock can be established by a *reductio ad absurdum* argument. Suppose that all the processes wish to enter the critical section but none of them has the token, so they are all halted, awaiting its arrival. The token is therefore in transit, it will after some finite time arrive at one of the processes (recall that, by hypothesis, the transmission delay is variable and desequencing is possible) and so unblock it.

The fairness follows from the fact that all messages are delivered within a finite time of issue. The postlude requires that P_i transfers the token to the first process P_l, found in scanning the set in the order $l=i+1, i+2, \ldots, n, 1, \ldots, i-1$, whose request has reached P_i; if the transmission delays for all messages are finite (i.e. no message is lost) all the processes will learn of the wish of some P_j to enter the critical section and will agree to this when its turn comes.

3.6 Messages and time-stamping

Number and size of messages

The algorithm requires either n messages ($n-1$ to broadcast the request and 1 to transfer the token) when the requesting process does not already hold the token, or 0 when it does.

The messages are of two types, the requests and the token itself. Those of the first type consist of three elements: one defines the type, one the value of the clock and one the identity of the process. Those of the second type have $n+1$ elements: the type and n clock values. Notwithstanding the large size of the messages of type 'token', this algorithm is preferable, so far as the number of messages needed is concerned, to the others mentioned.

Use of time-stamping

In this algorithm the time-stampings of the request messages are not used, as they are in Lamport's, to reset the clocks and so correct any drifts; they are used simply as counters that record the number of times the various processes have asked to use the critical section, and so to find whether or not the number of times that P_i has been given this access, recorded as the value of token(i), is less than the number of requests it has made, known to P_j by the value of $\text{request}_j(i)$. The function 'max', used in the processing associated with the reception of requests, results in only the last request from P_j being considered if several had been delivered out of sequence.

If the additional assumption has to be made that messages can be lost, two problems arise. If the lost message is a request, then, unless there is a recovery procedure based on a time-out mechanism, the process making the request is removed from the competition for the critical section and is halted, no longer able to access this; in this case the algorithm continues to operate as before but with only the remaining processes. But if the loss is of the message conveying the token, the algorithm loses many of its properties: mutual exclusion is maintained but deadlock begins to appear. This last problem also arises if the process holding the token fails; we now study a distributed algorithm for regenerating a lost token.

4 AN ALGORITHM FOR REGENERATING THE TOKEN

4.1 A token circulating on a logical ring

There is an exceedingly simple algorithm for mutual exclusion when the topology of the communication between processes is that of a ring, meaning that each process can communicate only with one of its neighbours or with both, according as the ring is unidirectional or bidirectional. Again, the privilege is represented by a special message, the token, which the processes hand from one to the other around the ring, so the protocol controlling the use of the critical section by the process P_i (i in 0, 1, . . ., $n-1$ for n processes) is as follows.

```
wait (token) of P(i-1)mod n;
  ⟨critical section⟩
send (token) P(i+1)mod n;
```

This algorithm ensures mutual exclusion because there is only one message of the type 'token'; further, if the ring is unidirectional there can be neither deadlock nor starving, as the token circulates from process to process around the ring.

However, problems can arise concerning resilience to failures, and these are of two sorts, those resulting from the failure of a process and those from loss of the token. The first are dealt with by identifying the failed process and reconfiguring the ring; methods, based on the principle of 'mutual distrust', are described in [LEL 77] and [COR 81]. Here we consider the loss and regeneration of the token.

4.2 Loss of token: Misra's algorithm

The usual methods for detecting the loss of the token and setting in motion the actions needed to recover from this loss are described

in [LEL 77] and [PES 83] – the 'bully algorithm'; these involve use of the time-outs, and require the identities of the individual processes to be known. The reason for this last requirement is that if the delays are not measured accurately enough a lost token can be regenerated by several processes but only one copy must be retained; the identities, all of which are different, are used to bring this about.

Misra [MIS 83] has proposed a method that requires knowledge neither of delays nor of process identities. It uses two tokens, each of which serves to detect the possible loss of the other, by this means: a token T1 arriving at the process P_i can guarantee that the other token T2 has been lost — and can therefore regenerate it — if neither it nor P_i has encountered T2 since T1's last passage through P_i. Whilst the behaviours of the two tokens are symmetrical, the privilege with which the mutual exclusion is concerned is, of course, associated with only one.

The loss of a token is detected by the other in one passage round the ring; and the algorithm works only when one token having been lost, the other makes a complete turn round the ring without itself being lost. To deal with this possibility the algorithm can be generalized to use any number of tokens, when it will work so long as one remains on the ring.

The algorithm

Let us call the tokens 'ping' and 'pong', and with these associate numbers 'nbping', 'nbpong' respectively, equal in absolute value but opposite in sign, that record the number of times the tokens have met; these numbers are therefore related by the constraint

$$\text{nbping} + \text{nbpong} = 0$$

Initially the two tokens are both in an arbitrarily chosen process and the values are

$$\text{nbping} = 1, \text{nbpong} = -1$$

Each process P_i carries an integer variable m_i, initialized to 0, that records the number, nbping or nbpong, associated with the token that last passed through P_i. The behaviour of P_i as follows.

```
when received (ping, nbping) do
                if m_i = nbping then
                   begin ⟨pong is lost, regenerate it⟩
                         nbping ← nbping + 1;
                         nbpong ← −nbping;
                   end;
                   else m_i ← nbping;
                end if;
                end do;
when received (pong, nbpong) do
                ⟨as before, interchanging ping and pong⟩
                end do;
when meeting (ping, pong) do
                nbping ← nbping + 1;
                nbpong ← nbpong − 1;
                end do;
```

Essentially, the algorithm conserves the relation nbping + nbpong = 0, changing these numbers appropriately each time the tokens meet. When P_i receives one of the tokens it compares m_i with the associated number. Suppose this is nbping and $m_i \neq$ nbping; then either m_i has been changed since the last passage of ping, meaning that pong has passed through P_i in the meantime, or the two tokens have met and consequently their numbers have been changed to record this: the passage of ping is then recorded by P_i setting $m_i :=$ nbping. If P_i finds that $m_i =$ nbping then pong was not the last token to pass through P_i (for otherwise m_i would have been modified appropriately and the negative values recognized by P_i) nor have the tokens met in their passage round the ring (for then nbping would have been increased by 1 to record the fact). Therefore pong must have been lost and P_i can regenerate it, setting the values of nbping and nbpong appropriately.

Values of nbping, nbpong

As the procedure has been described, there is no limit in principle to these counter values, and this would constitute a major disadvantage of the algorithm. They can, however, be bounded, by a device that requires only comparisons of the equal/unequal type and no greater than/less than discrimination, and therefore a simple hardware implementation.

Suppose there are n processes P_i. When the counters are updated, as a result either of meeting or of one having been lost and regenerated, their absolute values must be different from all the values m_i. It is thus sufficient to increment them modulo $(n+1)$:

nbping, for example, cannot then have the same value as any of m_i, for this could result only from its having been updated $n+1$ times since ping's previous passage through P_i, which is impossible since there are only n processes and the tokens meet only once in a process.

5 ELECTIVE ALGORITHMS

5.1 Introduction

From the point of view of control, many algorithms that are called distributed are in fact centralized: there is a single co-ordinating process that performs certain functions on behalf of the others when they ask for these — the classical 'client/server' protocol. Any algorithm for which a centralized expression is known can be implemented thus — mutual exclusion, detection of mutual blocking, etc. — with the messages serving only to convey the requests for service and the results obtained; so far as the using process is concerned everything has the appearance of a call to a distant procedure.

Such an algorithm, like any centralized algorithm or indeed any algorithm for which the texts of its processes are not symmetrical, suffers from the great disadvantage that a failure of the co-ordinating process results in the failure of the algorithm as a whole. This can be overcome by the remaining processes conducting a negotiation among themselves with the aim of electing one of their number to take over the role of co-ordinator. The protocols associated with such negotiations are called elective algorithms. These are usually based on the assumption that each process has a unique identifier, taking the form of a number; if the convention is adopted that at any instance the co-ordinating process is the one with the largest identifier (or alternatively, the smallest), then after a failure of this process the elective protocol consists in choosing the process with the largest (or smallest) of the remaining numbers.

Elective protocols can be useful in distributed algorithms in which one process plays a role that is not necessarily that of co-ordinator but is special in some other way (cf. Dijkstra's algorithm for mutual exclusion [DIJ 74]).

There are several elective algorithms. Two which we shall not describe here are given in [PES 83]: the 'bully algorithm' and an algorithm for ring topology due to Lelann [LEL 77]; if there are n processes each of these algorithms requires n^2 messages for the election. We shall give three algorithms, all assuming ring topology; the first is distinguished by its simplicity and by its methodology – it uses the principle of 'selective extinction' – and the others by requiring only $O(n \log n)$ messages.

5.2 The Chang and Roberts algorithm [CHA 79]

This applies to the case where the processes, the total number of which is not necessarily known, are linked in a unidirectional ring; as above, each process P_i has a unique numerical identifier. The principle is simply one of finding the maximum of a set of numbers asociated with the processes.

The processes can be arranged in any order around the ring. Each process P_i 'knows' the value of its own identifier i, which it transmits to its neighbour on the left, Pj say. P_j then compares this with its own identifier j and transmits the greater to its left-hand neighbour: this is the 'selective extinction' method. The election starts with one process sending such a message and marking itself as taking part in the election – at this stage it will be the only one so marked. The unmarked process receiving the message will act as just described and mark itself as a participant. When a marked (i.e. participating) process receives its own identifier it knows that this must be the greatest and that therefore it is elected to play the special role; and if this is necessary to the working of the algorithm for which the election is being conducted, it must then transmit its identity to all the other processes.

The algorithm

Each process P_i carries the following declarations.

```
constant   my_number:   value i;
variables  participant: boolean initialized F;
           co-ordinator: integer;
```

The variable 'co-ordinator' is used only if it is necessary for all the processes to know which has been elected.

The messages circulating around the ring are of two types: 'election', indicating that the value transmitted with the message is a candidate for election; and 'elected', indicating that the value is the identifier of the elected process. When 'co-ordinator' is not used, neither are these latter messages (which are used to update the variables). The communication primitive **sendL**, used here, is tailored to the particular ring of processes; it enables a process to send a message to the process designated as its neighbour on the left.

```
when decision (initiate_election) do
              participant ← T;
              sendL (election, my_number);
              end do;
when received (election, j) do
              case j > my_number then begin
                        sendL (election, j);
                        participant ← T;
                        end case;
              case j < my_number and participant then begin
                                   sendL (election, my_number);
                                   participant ← T;
                                   end case;
              case j = my_number then sendL (elected, i);
                                   end case;
when received (elected, j) do
                     co-ordinator ← j;
                     participant ← F;
                     if j ≠ my_number then sendL (elected, j);
                     end if;
end do;
```

The principle of selective extinction is applied when a process, P_i say, a participant in the election, receives a message from another participant carrying a number less than its own: P_i transmits no message on this occasion, so the received message is 'extinguished' and by continuation all the messages.

Proof of validity of the algorithm

What has to be shown is that the algorithm picks out one and only one number, the maximum.

Because of the unidirectional nature of the ring, a message issued by any process must be received by each of the others before returning to its sender. Only the message carrying the greatest number, once this has been issued, will not be replaced

by one carrying a different (greater) number at any stage and only this, therefore, can make a complete circuit of the ring. This message must make a complete circuit because either the corresponding process P_{max} is the one that started the election and therefore issued this message, or it was not the initiator but then would receive a message at some later stage, in response to which it would issue the message carrying this greatest number.

Timing

There are two extreme cases. In the first, every process initiates an election at the same time, with the result that the largest number is found in one turn round the ring; in this case the time required is proportional to the number n of processes. At the other extreme, only the left-hand neighbour of P_{max} initiates the election; the message has then to travel through $n - 1$ processes before reaching P_{max} which then issues the largest number, which in turn has to travel through $n - 1$ processes before returning to P_{max}; so this case requires a time proportional to $2(n - 1)$. In either case, and therefore in general, the time required is $0(n)$.

Number of messages of type 'election'

The most favourable situation is that in which the processes are arranged round the ring in increasing order of their identifying numbers and the election is initiated by P_{max}. The message has then to undergo n transfers before returning to P_{max}, so the number involved here is $0(n)$.

In the least favourable situation the processes are arranged in decreasing order and all initiate an election simultaneously; then the message issued by P_i undergoes i transfers, so the total number is

$$\sum_{i=1}^{n} i = \tfrac{1}{2}n(n + 1)$$

Thus the maximum number of messages is $0(n^2)$.

To deal with the cases intermediate between these we use the fact that the probability $P(i,k)$ that the message i is transferred k times is the probability that the first $k-1$ left-hand neighbours of P_i have identifiers that are less than i and that that of the kth is greater. If we write $C(a,b)$ for the number of combinations of b items from a set of a this is

$$P(i,k) = \frac{C(i-1, k-1)}{C(n-1, k-1)} \times \frac{n-i}{n-k}$$

The message carrying the number n is transferred n times; the mean number of transfers from that of number i is

$$E_i = \sum_{k=1}^{n-1} k\, P(i,k)$$

and therefore the mean total number is

$$E = n + \sum_{i=1}^{n-1} E_i$$

which can be shown [CHA 79] to be $0\,(n \log n)$.

5.3 The Hirschberg and Sinclair algorithm (HIR 80]

This makes a number of the assumptions of the previous algorithm: each process has a unique numerical identifier and is not necessarily aware of the total number of processes; the processes are arranged around a ring in arbitrary order, but now the ring is bidirectional, so that messages can circulate in either direction. The communication primitives are as follows, all relating to a typical process P_i:

sendLR send the same message to both the left-hand and right-hand neighbour

pass pass the message received from the right-hand neighbour to the left (or conversely) without modification

respond send a response to the neighbour from which a message has just been received

These are closely tied to the topology of the bidirectional ring; they could all be expressed in terms of a single primitive **send**, but this would confuse the internal logic of the algorithm with that of the communication system; so we shall prefer to keep to the higher level forms.

The algorithm is based on the idea of conducting a sequence of elections on increasingly large subsets of the processes until all have been included. If P_i initiates an election it declares itself a candidate and seeks to find if its own identifier (whose value it knows) is greater than that of either of its two neighbours (which it does not know) by sending this value to each of these neighbours, P_k and P_l say. These perform the necessary comparisons and if either k or l is found to be greater than i the corresponding process replaces P_i as candidate; otherwise P_i remains the candidate and tests this against a larger set of processes, in fact doubling the number of processes consulted. This continues until all the processes have been consulted. The subsequent role of a candidate defeated at any election is simply to pass any message received from one hand to the other, using **pass**.

The algorithm

Each process P_i carries the following declarations:

constant my_number: **value** *i*;
variables state: **position initialized** not_involved;
lgmax: **integer**;
winner: **integer**;
nbresp: 0, 1, **or** 2 **initialized** 0;
respOK: **boolean initialized** T;

The type 'position' of the variable 'state' defines the possible states of a process in relation to the election in progress:

type position = (not_involved, candidate, lost, elected);

The messages relating to the election are of two types, declaring a candidature and responding to a candidature declaration respectively. Those of the first type are associated with the primitives **sendLR**, **pass** and have the form

(candidate, number, lg, lgmax)

where number is the identifier of the process sending the candidature message
lg is the distance (length) around the ring already travelled by this message
lgmax is the maximum distance the message must travel

The second type is associated with **pass** and **respond** and has the form

(response, bool, number);

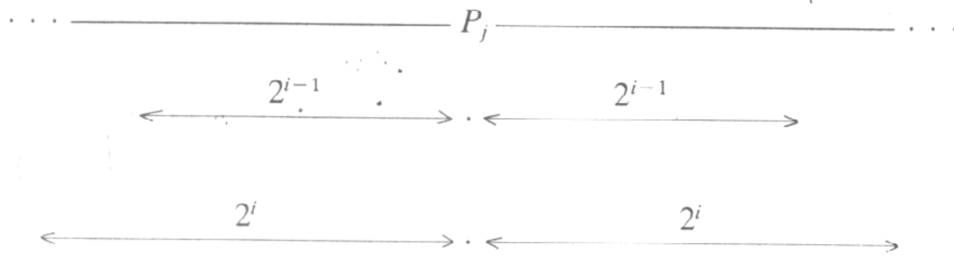
where bool is T if the resonse is favourable ('respOK'), F otherwise
number is the identifier of the destination process (and is therefore a means of addressing on the ring, since all the identifiers are different)

The text for P_i is as follows.

```
when decision (initiate_election) do
                              state ← candidate;
                              lgmax ← 1;
                              while state = candidate do
                                 nbresp ← 0; respOK ← T;
                                  sendLR (candidature, my_number, 0,
                              lgmax);
                                 wait nbresp = 2;
                                 if respOK then state ← lost)
                                 end if;
                                 lgmax ← lgmax * 2;
                                 end while;
                              end do;
when received (response, bool, number) do
              if number = my_number then nbresp:= nbresp + 1;
                                  respOK;= respOK∧bool;
              else pass (response, bool, number);
              end if;
              end do;
when received (candidature, number, lgmax) do
              case number < my_number then respond (response, F, number);
             if state = not_involved then initiate_election;
             end if;
             end case;
              case number > my_number then state ← lost;
                               lg ← lg + 1;
                               if lg < lgmax then pass (candidature, number, lg,
                               lmax);
                               else respond (response, T, number);
                               end if;
                               end case
              case number = my_number then if state ≠ elected then state ←
                                 elected;
                                 winner ← my_number;
                                 pass (ended, my_number);
                                 end if;
                                 end case;
              end do;
when received (ended, number) do
              if winner ≠ number then pass (ended, number);
                            winner ← number;
                            state ← not_involved;
                            end if;
              end do;
```

Addressing
This is done implicitly in the algorithm. Taking a candidate process as origin, any other process is addressed by means of the length of the path from the origin to that process; and the destination of a response message, having a unique identifier, is recognized by the number carried by the message.

Number of messages
The length of the path travelled by any candidature message is of the form 2^i where i is some positive integer. A process P_j will launch its candidature on a path of length 2^i only if it has not been defeated in the election by a process whose distance from itself is at most 2^{i-1}:



Further, in any set of $2^{i-1} + 1$ consecutive processes only one can launch a candidature on paths of length 2^i. Therefore, at the start of the procedure, although in the worst case all n processes can be candidates with paths of length 1, there are

at most $[n/2]$ that are candidates with paths of length 2
$[n/3]$ that are candidates with paths of length 4
$[n/(2^{i-1} + 1)]$ that are candidates with paths of length 2^i

Now a process that launches its candidature on a path of length 2^i will give rise to 2 messages (a candidature and a response) in each direction between each pair of processes on the path, at most; that is, to at most 4×2^i messages. So the total number of messages exchanged is at most

$$4 \times \left[1.n + 2\left(\frac{n}{2}\right) + 4\left(\frac{n}{3}\right) + \ldots + 2^i\left(\frac{n}{2^{i-1} + 1}\right) + \ldots \right]$$

Each term in this sum is the product of the number of paths and the length of those paths; none can exceed $2n$ in value and there are at most $(1 + \log n)$ terms, so an upper bound for the total number of messages is $8n(1 + \log n)$, which is $0(n \log n)$.

Thus this algorithm, although more complex than those mentioned at the end of Section 5.1, is more attractive because of the reduced number of messages.

5.4 The algorithm of Dolev, Klawe and Rodeh [DOL 82]

At the end of their paper (HIR 80] Hirschberg and Sinclair make the following conjecture: the number of messages needed by an elective algorithm on a *unidirectional* ring is $\Omega(n^2)$. If this were true it would imply that a bidirectional ring was needed to reduce the number of messages to a lower order. Many workers have sought either to prove or to disprove this conjecture and Dolev, Klawe and Rodeh disproved it by producing an algorithm needing $O(n \log n)$. We now describe this.

The principle of this algorithm – for a unidirectional ring, recall – consists in simulating a bidirectional algorithm as follows, in which we assume that the direction of flow of messages round the ring is from right to left for each process. A process P_i waits until it has received the identifiers of its two nearest neighbours on its right, say P_k, P_l where P_k is the nearer, that is between P_i and P_l. P_i then takes the point of view of P_k and looks to see if this process has the greatest identifier of the three; if so, P_i transmits this number and if not remains passive, from then on merely handing on messages that it receives. Thus initially there are n groups of three messages each; the first stage in the procedure consists of communication and comparison of identifiers as just described, at the end of which the number of active processes will have been divided by at least 2, for if P_i remains active its neighbour on the right necessarily becomes passive. Successive stages repeat this procedure, and at each stage the ring of active processes has to be respecified; the passive processes merely relay the messages. When stage m has been reached, a process that remains active in stage $m+1$ is necessarily one whose neighbour on the right has become passive at that stage.

The algorithm
To simplify the presentation

(a) we assume that all the processes decide to participate in the election; if that were not the case, a process would decide to participate on first receiving a message concerning the election;
(b) we shall not give the details of the means for disseminating the greatest identifier number when this has been found: a message of special type can transport this around the ring so that all processes are informed in a single turn.

Each process P_i carries the following declarations:

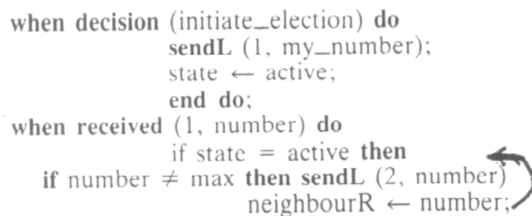
constant	my_number:	**value** i;
variables	state:	(active, passive) **initialized** passive;
	max:	**integer initialized** my_number;
	neighbourR:	**integer**;

The variable 'neighbourR' holds the identity of the first active processor on the right of P_i; 'max' holds the identity of the process whose point of view P_i has taken – recall that after obtaining the identifiers of its two nearest active neighbours P_i then behaves as the nearer one, which would have obtained the identifiers of its neighbours on the left and the right.

Messages of two types are needed to transmit the identifiers, so as to distinguish between the nearer and the further of the two active neighbours (the second being the right-hand neighbour of the first). These are of the form

$$(b, \text{number})$$

where b is either 1 or 2 according as 'number' is the identifier of the first or the second neighbour. As before, the primitive **sendL** is used to send a message to the neighbour on the left.



```
when decision (initiate_election) do
              sendL (1, my_number);
              state ← active;
              end do;
when received (1, number) do
              if state = active then
   if number ≠ max then sendL (2, number)
                    neighbourR ← number;
```

```
    else ⟨max is greatest number and is sent to all processes⟩;
     end if,
                    else sendL (1, number);
                    end if;
end do;
when recieved (2, number) do
                    if state = active then
                        if neighbourR > number and neighbourR > max then
                            max ← neighbourR;
                            sendL (1, neighbourR);
                        else state ← passive;
                        end if;
                    else sendL (2, number);
                    end if;
end do;
```

As already stated, the role of a passive process is simply to pass to its neighbour on the left any message received from its neighbour on the right. Initially all the processes are active and each local variable max holds the number of its own process. Thus the first message received by each process is of the form (1, number), which enables it to update its variable neighbourR and to send the message (2, number) to its neighbour on the left. When a process P_i has received such a message it has three identifier numbers of active processes: max, giving its own number, neighbourR giving that of its first neighbour on the right and number giving that of the second. If neighbourR is the greatest of these, P_i takes over the role of that process by means of the instruction max ← neighbourR, remains active and sends the message (1, neighbourR) to start the next stage in the procedure; otherwise P_i becomes passive.

Types of message

Each active process receives messages of type (1, number) and (2, number) alternately, thus obtaining the identifiers of its first and second active neighbours on its right. Since the order is always (1, . . .) (2, . . .) (1,. . .) (2,. . .) . . . the identifier 1/2 is redundant and only the process identifier number need be sent; thus only n distinct messages are needed. However, we have preferred to follow [DOL 82] and distinguish between the two types, in the interests of clarity.

Validity of the algorithm; number of messages

We shall give only an indication of the principle of a formal proof

of the validity of the algorithm; detailed proofs are given in [DOL 82] and in [PET 82].

At any given stage in the calculation, after receiving the two numbers, the only processes that remain active are those P_i for which neighbourR is the maximum of the set of three, when $P_{\text{neighbourR}}$ becomes passive. P_i then takes over the role of $P_{\text{neighbourR}}$ and starts the next stage by performing **sendL** (1, neighbourR). Thus the maximum value that P_i has found remains in the election.

As we have seen, the number of processes remaining at this next stage is at most half the number of the stage just concluded; so if the procedure starts with n active processes, at most ($\log_2 n + 1$) stages are needed to reduce this number to 1, when the algorithm terminates, leaving a single process that knows the greatest identifier.

At each stage $2n$ messages are exchanged by the processes: there are n processes (active and passive together) and each issues one message of type (1, number) and one (2, number). At the final stage only one process is active so only n messages are exchanged. Thus the total is $2n \log n + n$, i.e. $O(n \log n)$.

5.5 Other algorithms

5.5.1 Franklin's algorithm [FRA 82]

This is an elective algorithm for a bidirectional ring. The principle is similar to that of [DOL 82] but advantage is taken of the bidirectional nature of the ring and an active process tests to see if its identifier is greater than those of its neighbours on both left and right; if it is not then the process becomes passive and, as before, simply relays any message recieved. Again, the number of processes left active is divided by at least 2 at each stage; and as $2n$ messages are exchanged at each stage the total number is $2n\ (1 + \log n)$, or $O(n \log n)$ again.

5.5.2 Peterson's algorithm [PET 82]

As we have seen, Hirschberg and Sinclair's conjecture (Section 5.4) was shown to be false by Dolev, Klawe and Rodeh by their demonstration of an $O(n \log n)$ algorithm; the same algorithm was given independently by Peterson, so Section 5.4 could equally well have been headed 'Peterson's algorithm'. Competition between

these two groups of researchers has resulted in optimization of the algorithm: Dolev *et al.* reduced the number of messages to $1.5\ n \log n + O(n)$, then Peterson to $1.44n \log n + O(n)$, then Dolev *et al.* to $1.33n \log n + O(n)$ . . . Burns has shown that the lower bound for the multiplier is 0.15, reached when n is a power of 2, and that this holds whether the ring is unidirectional or bidirectional.

5.5.3 The algorithm of Korach, Moran and Zaks [KOR 84]

These authors have investigated the upper and lower bounds for the numbers of messages required by various algorithms. They themselves have suggested an elective algorithm — 'the king and the subjects' in which all the processes are kings at the start and at the end there is only one king, all the others having become subjects. The algorithm is important on two counts: the topology of the interprocess connections is no longer a ring but a complete graph, and the number of messages is again $O(n \log n)$.

REFERENCES

[CAR 83] Carvalho, O., and Roucairol, G., On Mutual Exclusion in Computer Networks, *Comm. ACM*, **26**(2) (Feb. 1983), pp. 146–147.

[CHA 79] Change, E. G., and Roberts, R., An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processors, *Comm. ACM*, **22**(5) (May 1979), pp. 281–283.

[COR 81] Cornafion (group name), *Systèmes Informatiques Répartis*, Dunod (1981), 367 p.

[DIJ 65] Dijkstra, E. W., Cooperating Sequential Processes. In F. Genuys (ed.) *Programming Languages*, Academic Press, New York (1965), pp. 43–112.

[DIJ 74] Dijkstra, E. W., Self-Stabilizing Systems in Spite of Distributed Control, *Comm. ACM*, **17**(11) (Nov. 1974), pp. 643–644.

[DOL 82] Dolev, D., Klawe, M., and Rodeh, M., An $O(n \log n)$ Unidirectional Distributed Algorithm for Extrema Finding in a Circle, *Journal of Algorithms*, **3** (1982), pp. 245–260.

[FRA 82] Franklin, W. R., On an Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processors, *Comm. ACM*, **25**(5) (May 1982), pp. 336–337.

[HIR 80] HIRSCHBERG, D. S. and SINCLAIR, J. B., Decentralized Extrema Finding in Circular Configurations of Processors, *Comm. ACM,* **23**(11) (Nov. 1980), pp. 627–628.

[KOR 84] KORACH, E., MORAN, S., and ZAKS, S., Tight Lower and Upper Bounds for Some Distributed Algorithms for a Complete Networks of Processors, *TAP, ACM Conference* (1984), pp. 199–205.

[LAM 74] LAMPORT, L., A New Solution of Dijkstra's Concurrent Programming problem, *Comm. ACM,* **17** (8) (Aug. 1974), pp. 453–455.

[LAM 78] LAMPORT, L., Time, Clocks and the Ordering of Events in a Distributed System, *Comm. ACM*, **21**(7) (July 1978), pp. 558–565.

[LEL 77] LELANN, G., Distributed Systems: Towards a formal Approach, *IFIP Congress, Toronto* (Aug. **1977**), pp. 155–160.

[MIS 83] MISRA, J., Detecting Termination of Distributed Computations Using Markers, *Proc. of the 2nd ACM Conf. on Principles of Distributed Computing, Montreal* (Aug. **1983**), pp. 290–294.

[PET 82] PETERSON, G. L., An O(n log n) Unidirectional Algorithm for the Circular Extrema Problem, *ACM Toplas,* **4**(4), (Oct. 1982), pp. 758–762.

[PET 83] PETERSON, G. L., A New Solution to Lamport's Concurrent Programming Problem Using Small Shared Variables, *ACM, Toplas,* **5**(1) (Jan. 1983), pp. 56–65.

[PES 83] PETERSON, J. L., and SILBERSCHATZ, A., Operating Systmes Concepts, Addison Wesley (1983), 548 p.

[PLO 84] PLOUZEAU, N., and RAYNAL, M., Spécifier les Algorithmes Distribués en ADA: Uné approche Cirtique, *Actes des Journées AFCET-ADA, Paris*, **42** (Dec. 1984), pp. 148–62.

[RAY 86] RAYNAL, M., *Algorithms for Mutual Exclusion*, MIT Press (1986), 120 p.

[RIC 81] RICART G., and AGRAWALA, A. K., An Optimal Algorithm for Mutual Exclusion in Computer Networks, *Comm. ACM,* **24**(1)(Jan. 1981), pp. 9–17.

[RIC 83] RICART, G., and AGRAWALA, A. K., Author's response to 'On Mutual Exclusion in Computer Networks' by Carvalho and Roucairol, *Comm. ACM,* **26**(2) (Feb. 1983), pp. 147–148.

[SUZ 82] SUZUKI, I., and KASAMI, T., An Optimality Theory for Mutual Exclusion Algorithms in Computer Networks, *Proc. of the 3rd Int. Conf. on Distributed Computing Systems, Miami* (Oct. **1982**), pp. 365–370.