Note

Two distributed problems involving byzantine processes*

Joffroy Beauquier

LRI-CNRS, Université Paris Sud, Bâtiment 490, F91405 Orsay Cedex, France

Communicated by M. Nivat
Received June 1990

Abstract


We present solutions to two classical problems concerning distributed systems in which some sites or processes can possibly have byzantine faulty behavior. We first study the naming problem (how to give each site of a network a unique identifier). We are naturally led to make some supplementary assumptions about the synchrony of message passing, the connectivity of the underlying graph and the existence of a special site, provided with a digital signature, for initiating the protocol. The solution that we present uses three waves of messages between the initiator and any other site. Then, we solve the mutual exclusion problem with particular assumptions about the behavior of byzantine processes. The solution implements each critical section as a separate segment, whose address (necessary to access it) is "hidden". A process must reconstruct this address before entering its critical section, involving the cooperation of a number of other processes. For the two problems, protocols are given and their complexity is estimated.

1. Introduction

In this paper we will consider two classical problems concerning distributed systems, under the assumption that some sites or some processes can possibly have a faulty behavior. The first one is the naming problem and the second one the mutual exclusion problem. We will briefly present them and discuss about the assumptions that we make.

* This research has been supported by the PRC C.1.
Almost all distributed algorithms assume as a precondition that each site has a unique identifier. Under usual hypotheses, giving each site an identifier is straightforward, even if each site does not know about the entire network. For example, a circulating token containing an integer can perform a traversal of the network, starting with the value 1, and increasing it by 1, each time it leaves a site for the first time [6]. Each site chooses as its unique identifier the integer in the token at its first arrival. This method, like some other methods (cf. [4, 11]), depends on the fact that each site transmits the token and increases its value correctly. If a given site decreases the value of the token instead of increasing it, two (correct) sites would receive the same identifier. And what about a site keeping the token forever?

In this paper, we will be interested in the (non-trivial) problem of giving each site a unique identifier, assuming that some sites can have byzantine behavior [7]. That means that, roughly speaking, the bad sites can perform anything: sending false messages or not sending messages at all, having a correct behavior, stopping and then restarting, etc. A site that permanently follows its protocol is called correct.

Without some supplementary—rather strong—assumptions, the naming problem has no solution. We will not present a formal treatment yielding results of indecidability, but simply describe the wrong situations that can possibly occur.

Firstly, let us consider the network in Fig. 1.

If the site s can only communicate with the rest of the network through the byzantine site byz, it cannot necessarily distinguish between the situation above and the situation in Fig. 2.

Thus, if a deterministic algorithm for solving the naming problem would exist, it would have to give s an identifier independent of those given to the n other sites. Then the uniqueness of the identifiers would not be ensured.

![Fig. 1.](image1)

![Fig. 2.](image2)
Such a paradox also appears when a site is "surrounded" by byzantine sites or when two or more correct sites cannot communicate without avoiding a byzantine one.

So, we are naturally led to an assumption about the connectivity degree of the network.

Let $k$ be a strict upper bound to the number of possible byzantine processes. We will assume that the network is $k$-connected. That is there are at least $k+1$ disjoint paths connecting two correct sites. Among these $k+1$ disjoint paths, at least one goes through only correct processes.

The most general and difficult hypothesis about message passing is asynchrony. When the network is not known, asynchrony does not allow to detect the termination of an algorithm. We will assume here that each site knows an upper bound to the diameter of the network.

The third assumption concerns asymmetry of the sites. The naming problem has no solution even if all sites are correct, if the sites are perfectly symmetrical and, therefore, indistinguishable.

In order to prove that ab absurdo, consider the network of Fig. 3, in which the two sites are identical.

![Fig. 3.](image)

Since the two sites follow the same protocol at the same time, they receive the same information from the other and then choose the same identifier. So we will assume that there exists a particular site, the initiator, that starts the naming algorithm. Each site knows whether it is the initiator or not, and there is a unique initiator. We assume that the initiator is always correct and that the messages it sends are authenticated. This requirement can be achieved, for instance, by a cryptographic digital signature [10].

The problem of the mutual exclusion is another classical problem concerning the synchronization of concurrent or distributed processes. Each process is given with a special part of its code, called the critical section and a protocol must be designed, to ensure that, at any time, at most one process is executing inside its critical section. Moreover, the protocol must yield some fairness, absence of starvation and deadlock requirements. This problem appears, in particular, when an exclusive access to a resource, like a printer, must be imposed.

In a large network, with a number of host computers, the correctness of all the processes running in the network cannot be guaranteed at the same level. For instance, processes running on micro-computers or small computers are supposed to be less reliable than those on large mainframes, where a control system does exist. But the mutual exclusion condition for some access must be guaranteed,
whatever the processes do. Some solutions to the problem of mutual exclusion have been presented in the case of crash faults [9, 3]). But, in contrast with crash faults, a faulty byzantine process cannot generally be detected. The difficulty is thus to design a protocol that deals with byzantine faults.

It is not hard to see that such a protocol does not exist for the mutual exclusion problem and that supplementary assumptions must be made, in order to obtain partial solutions. The main difficulty that we are faced to is: when a correct process is executing in its critical section, how can we avoid that a faulty process enters its own critical section? The idea for solving it is to use a mechanism analogous to a capability. If the critical section is implemented as a separate segment and if the access to this segment requires the knowledge of its address, this address should be given to any process only under distributed control. In our solution, a process wanting to enter its critical section must reconstruct the required address from pieces of information sent by the other processes, after they have agreed on him. Schematically, the protocol for a process is:

- ask the other processes for their pieces of information for reconstructing the address of the critical section segment,
- wait until this address can be reconstructed,
- enter the critical section, then leave it,
- send acknowledgments to all other processes (the exit section).

In this paper, we will consider processes that can exhibit byzantine behavior everywhere in their code, except during the execution of some special parts of the protocol, namely the critical and the exit sections. We will call such processes, as in [2], locally byzantine processes. Note that this hypothesis is absolutely necessary for avoiding a byzantine process to remain forever in its critical section or not to notify its exiting to the other processes.

2. The naming problem with faulty processes

2.1. Preliminaries

Let $G_i$, $1 < i < n$ be processes. Assume that every $G_i$ can directly exchange messages with some other $G_i$ (its neighbors), that message passing is synchronous and that no message is modified or lost. $G_i$ only knows about its neighbors and nothing else about the network topology. Assume topology is fixed.

The naming problem is to design a protocol $P$ that $G_i$ may use to obtain a unique identifier. The protocol $P_i$ involves exchanging messages with its neighbors. As long as $G_i$ computes according to $P_i$, it is called correct. Once a process $G_i$ deviates from $P_i$, it becomes faulty (or byzantine) and is considered to remain faulty, even if, later on, it reverts back to following $P_i$.

Initially, each site knows its adjacent communication lines and knows them by a local name (a number).
We also assume that a particular process has the role of initiating the protocols. So each process initially knows whether it is the initiator or not. The initiator is assumed to be correct.

At the end, we suppose that there is a public key cryptographic system, allowing the initiator to use a digital signature [10]. Let $\sigma$ be the digital signature of the initiator and $\sigma(m)$ a signed message.

2.2. The protocol

We will first sketch how the protocol works. At the beginning, the initiator numbers its neighbors and for all numbers $i$, sends to $i$ the (signed) message $\sigma(\text{"initialization"}, i)$. Each neighbor, receiving such a message, checks for the digital signature of the initiator (in case of failure, the message is rejected), numbers its own neighbors, for each of them concatenates the number to the message, then sends it to the related neighbor. Such a message has the form $(\sigma(\text{"initialization"}, i), j)$, where $j$ is the neighbor number. In a general manner, we call valid any message endorsed by the digital signature of the initiator. During the whole protocol, correct sites only accept valid messages.

When a correct site $s$ receives a valid message, it knows how its neighbor has numbered the communication line between them. If this number is $j$ and if $s$ has itself numbered the communication line by $i$, $s$ registers that $iC' = j$. If, after this first numbering, $s$ receives from the same neighbor conflicting information about the number of the line, it simply does not accept any other message from this neighbor.

The first time a correct site $s$ receives a valid message, it starts a watchdog timer, initialized to the value $3 \times D \times 6$, where $D$ is the network diameter.

At this point, it should be noticed that byzantine sites can possibly relay messages endorsed by the initiator, after having modified some unencoded information.

In the sequel, each site receiving a valid initialization message relays it to its numbered neighbors, after having appended its number to the message. Moreover, each site manages a list, containing the valid messages already received. Each time a valid message is accepted, the list is searched to check that the message is not a looping message (more precisely, it is searched whether a left prefix of the message is already in the list, as a complete message). Looping messages are rejected. So correct sites progressively delete valid messages.

It should be noticed that a correct site can possibly accept and relay a message that contains false information. Figure 4 shows such a situation.

The first byzantine site receives $\sigma(\text{"initialization"}, 3)$ from the initiator. It relays it to the other byzantine site without change. This one appends 1 and sends the message $(\sigma(\text{"initialization"}, 3), 1)$ to the correct site. So, $s$ can think that the network is as in Fig. 5.
Now, we introduce the notion of *confirmation*. Each time a site accepts a valid message, it returns to the initiator a return message, by the reverse path. For that it uses the inverse of the numbers of its outgoing communications lines as registered previously, cf. Fig. 6.

As a matter of fact, the actual reverse path is followed only if the message does not go through a byzantine site deviating it. A return message is identical to an
Distributed problems involving byzantine processes

initialization message, except that it has a first field “return” (unsigned). A return message is relayed by the correct sites toward the initiator and a pointer shows at what step the message has arrived. For instance,

\( \text{("return", } \sigma(\text{"initialization"}, 3), 1, 3, 3, 2) \)

is a return message that has followed back a line numbered \( 2^{-1} \) and a line numbered \( 3^{-1} \).

Each time the initiator receives a return message, it erases the fields “return” and “initialization”, signs the rest of the message, adds the message in clear with a pointer on the first number and a field “confirmation”. For instance, such a confirmation message is:

\( \text{("confirmation", } \sigma(3, 1, 3, 3, 2), (3, 1, 3, 3, 2)) \).

Then the confirmation message is transmitted by the correct sites, according to the included path, until arriving at the destination. At each step, the digital signature of the initiator is checked. Moreover, the point is checked and updated, indicating at what step the message has arrived. For instance, this is a circulating confirmation message:

\( \text{("confirmation"}, \sigma(3, 1, 3, 3, 2), (3, 1, 3, 3, 2)) \),

indicating that, from the initiator, the message has followed the lines 3, 1, 3 and 3 and that the destination will be reached by following line 2. When a confirmation message arrives at its destination, the site checks the digital signature of the initiator, checks whether or not the message is already in its list of valid messages and, if so, stores the value of the path in a table. After the delay \( 3 \ast D \ast \delta \), a correct site stops and concatenates in a string all the values of paths that are stored in the table (using a separator). Then it chooses the result of the concatenation as an identifier.

Now, we present a more precise formulation of the protocols. Some data structures and primitive are presented in Appendix A. The symbol “!“ denotes the marked catenation of strings (catenation with a marker insertion).

**protocol initiator;**

begin

\( \langle \text{numbering of the neighbors} \rangle \) 
\( \text{timer} \ (3 \ast D \ast \delta) \);

for all number \( i \) send \( \sigma(\text{initialization}, i) \) to (neighbor \( i \));

upon reception of (return, \( i!m \)) send 
(\( \sigma(\text{confirmation}, i!m), i!m \)) to (neighbor \( i \));

upon reception of 
(\( \text{initialization or confirmation messages} \))

do nil;

upon timer expiration do stop

end;
protocol site;
  begin
    identifier := empty string;
    (numbering of the neighbors)
    upon reception of (α(initialization, i), i!m) do
      if (first reception of an initialization message) then
        timer (3*D*δ);
      if i!m not in prefix (valid) then
        send (s(init, i), i!m) to (neighbor j);
        valid := valid + i!m;
      send (return, i!m]j, i!m) to (neighbor j^{-1});
    endif
    endo
    upon reception of (return, i!m']j]k]m", i!m']j]k]m") do
      send (return, i!m']j]k]m", i!m']j]k]m") to (neighbor k^{-1});
    endo
    upon reception of (α(confirmation, m']j]k]m"), m']j]k]m") do
      begin
        send (α(confirmation, m']j]k]m"), m']j]k]m") to (neighbor i);
        if m']j is in valid then ident := ident|m]j|
      end
    endo
    upon timer expiration do stop
  endo
end;

We now prove that the protocols are correct. All protocols do terminate, since
the clock interrupts have a finite delay.

Lemma 1. Any correct process receives at least one initialization message and one
confirmation message with the same path information that has only passed through
correct processes.

Proof. The graph is k + 1 connected. Then there exist k + 1 disjoint paths leading
from the initiator to any correct process. Among them, at least one does not contain
any faulty process (recall that there are at most k faulty processes). □

We call normal for P, any message that has gone only through correct processes
between the initiator and P. Lemma 1 states that any correct process receives normal
initialization and confirmation messages. Moreover, any message normal for P, is
valid for P.
Lemma 2. Let $P_1$ and $P_2$ be two distinct correct processes and let $m_1$ and $m_2$ be two normal messages, respectively for $P_1$ and $P_2$. Then $m_1$ and $m_2$ are different.

Proof. If $m_1$ and $m_2$ are equal, that means that the path from the initiator to $P_1$ is the same as the path from the initiator to $P_2$, since $m_1$ and $m_2$ are relayed only by correct processes. So $P_1$ and $P_2$ cannot be distinct. □

Lemma 3. Let $P_1$ and $P_2$ be two distinct correct processes and let $m$ be a normal message for $P_1$. Then $m$ cannot be a valid message of $P_2$.

Proof. By Lemma 2, $m$ cannot be a normal message for $P_2$. Thus, if $m$ is a valid message for $P_2$, it has been relayed by a faulty process. But, since $m$ is a normal message for $P_1$ it only transits through correct processes. □

Lemma 4. Two distinct correct processes accept (at least) two distinct valid messages.

Proof. By Lemma 1 each of them accepts at least one normal message and by Lemma 3, such a message is not valid for the other. □

Theorem 5. At the end of the protocols, two distinct correct processes have distinct identifiers.

Proof. Since the identifier consists of the concatenation of valid messages separated by a marker, they differ, by Lemma 4, by at least one message. □

2.3. Complexity

Let $m$ be the maximum degree of the graph (maximum number of outgoing lines from a site). Since each correct process stops after the delay $3 \times D \times \delta$ ($D$ is the diameter of the graph) and sends at most $m$ messages per $\delta$ time units, for each accepted message, an upper bound of the total number of exchanged messages is $3 \times D \times m^D$.

The time complexity is obviously $3 \times D \times \delta$.

3. The mutual exclusion problem with faulty processes

3.1. Preliminaries

Let $G_i$, $1 < i < n$ be processes. Assume that every $G_i$ can directly exchange messages with every other $G_j$, that communications are asynchronous and that, whenever a process $G_i$ sends a message to $G_j$, the message eventually reaches $G_j$.

Each $G_i$ executes a program, that has a section of code, called its critical section. When one process is executing in its critical section, no other process is allowed to
execute in its own critical section. The critical section problem is to design a protocol $P_i$, which $G_i$ may use to cooperate. The protocol $P_i$ involves exchanging messages with other $G_i$'s, deciding when to enter the critical section and when to wait for entering it. As long as $G_i$ computes according to $P_i$, it is called correct. Once a process $G_i$ deviates from $P_i$, it becomes faulty and is considered to remain faulty, even if, later on, it reverts back to following $P_i$. $P_i$ also includes the code that $G_i$ executes to request permission to enter its critical section. The section of code implementing this request is called the entry section. The critical section is followed by an exit section. The remaining code is the remainder section, that can be divided into two parts.

A solution to the mutual exclusion problem must satisfy three requirements:

(i) **Mutual exclusion.** If process $G_i$ is executing in its critical section, no other process can be executing in its own critical section.

(ii) **Progress.** If no process is executing in its critical section and there exists some process that wishes to enter its critical section, it cannot be delayed by processes executing forever in their remainder sections.

(iii) **Bounded waiting.** Once all processes are aware of the intention of a process $G$ to enter its critical section, there must exist a bound on the number of times that other processes are allowed to enter their critical section before $G$'s request is granted.

We make the important assumption that the entry and exit sections are always correctly executed, and that a process never crashes in its critical section. Even after becoming faulty, it correctly executes its critical and exit sections. Such a faulty process is called a locally faulty process.

The underlying idea to practically implement this assumption is that each $G_i$ must be a process executing under control of the operating system of an host computer. The critical section is implemented as a separate segment, in its own memory locations. For executing this segment, a process must necessarily know its address (for instance own a capability for the segment). In the sequel, we will drop these implementation details and keep in mind that the three sections are always correctly executed.

We will also suppose that the distributed system has a public key cryptographic system, yielding digital signatures [10]. We note $\sigma_i$ the digital signature of the process $G_i$ and $\sigma_i(m)$ a signed message. All messages will be implicitly signed by the sender, so that we will always omit $\sigma_i$ and simply write $m$ for a signed message.

3.2. The protocol

The solution that we will present uses some data structures and primitives, that will be part of each protocol (thus, variables are local to a site, but some are also global to procedures on a same site). They are given in Appendix B. The solution consists in a main program and two interrupt handlers, that are the same for all $G_i$'s.

The main program is made of three procedures that are concurrently executed.
Program Mutual_exclusion;  \{for $G_i$\}
var s: address;
in: boolean;
begin
allocate(critical_section, s);  \{loads the procedure segment in main memory\}
in := false;  \{in is local to $G_i$\}
cobegin
   Listening;
   Byz_cons;
   Local_prog
coend
end.

Local_prog is any specific program that $G_i$ is supposed to execute. The only thing that it is assumed about this program is its general structure.

Program Local_prog;
procedure critical_section;
begin
repeat
   (remainder section 1);
   svc(ask_for_critical_section);
   (remainder section 2)
forever
end.

A FIFO message buffering system is assumed to exist on each site. The procedure Listening receives the arriving messages (from the buffering system), implicitly checks that they are correctly signed and manages two information tables, ask and acq of type control.

The record ask[j] (of type note) is used to memorize the last message “ask for entering” (setting flag to the value true) with its data (the value of counter) and the corresponding value $I_i(s)$ (coded as an integer). The record acq[j] is used to memorize the date of the last message “exit”. The records ask[j] and acq[j] are both local to $G_i$. The fields flag are initialized to false and date and info to 0.

It must be noticed that only the last messages “ask for entering” and “exit”, with their dates, are saved. We consider only solutions to the problem that use finite storage. Thus, since communication is asynchronous, we cannot store all untreated messages, because their number is unbounded.

The procedure Listening treats an inbound message atomically. The text of Listening is quite straightforward and we will omit it.

The procedure Byz_cons is used to obtain a consensus among the $G_i$’s, on the particular $G_i$ that should be allowed to enter its critical section. It uses as a primitive the procedure byzantine_agreement of Rabin [8], that gives the agreement with an
expected number of four rounds. As in [8] a correct “dealer” is assumed to provide to each $G_i$ an infinite sequence of shadows (in the sense of threshold schemes, cf. [5] or [12]) of random bits. The reconstruction of the secret bit, indexed by the value of the variable $cons\_counter$, is supposed to be part of byzantine agreement.

An important point of Rabin’s technique is that, when a process stops the agreement protocol, it has the proof that all correct processes will stop it in the same round or one round later.

The parts of the procedure represented underlined are executed atomically, at the local level.

**Procedure** byz_cons; {for $G_i$}

var $cons\_counter$: time_stamp;
$i, j$: site;
$ask, acq$: control;
$m, result$: consensus;
$f$: message;

begin

repeat

$cons\_counter := 1; \{ cons\_counter is local to G_i \}$

$j := 1; \{ j is local to G_i \}$

repeat

if $ask[j].flag = false$ then

$m.value := no\_demand$

else begin \{ $G_i$ is wanting to enter \}

$m.value := j;\$

$m.counter := ask[j].dute;$

end;

end;

byzantine_agreement ($cons\_counter, m, result$);

$cons\_counter := cons\_counter + 1;$

$j := j + 1 \mod n; \{ try for another process \}$

until $result.value <> no\_demand and result.value <> default\_value;$

{repeat until a wanting process is chosen}

if ($result, ask[result].date) = m and

ask[result].date > acq[result].date

{no new demand and no exit during execution of byzantine_agreement}

then

begin

send($f, shadow, i, result, ask[result].counter), I(result) (s));

{giving the shadow of the address to the wanting process}

while $ask[result].date > acq[result].date$ do nil; \{ waiting for exit \}

end

forever

end;
At the end, we present the two interrupt handlers, executed under the control of the local operating system. Recall that execution of a \textit{svc} causes the processor status word to be put at the top of the stack.

**Interrupt-handler for** \textit{svc(ask\_for\_critical\_section)};

\textbf{var} \hspace{1em} \textit{j}: site; \\
\hspace{2em} \textit{f}: message; \\
\hspace{2em} \textit{s}: address; \\
\hspace{2em} new\_psw: psw; \\
\textbf{begin} \\
\hspace{2em} \textbf{while} \textit{in} = \textit{true} \textbf{do nil}; \hspace{1em} \{G, already made an svc for entering without exiting\} \\
\hspace{2em} threshold\textit{(s, n, t + 1)}; \hspace{1em} \{each \textit{I}_j(s) is signed by the local system\} \\
\hspace{2em} \textit{counter} := \textit{counter} + 1; \\
\hspace{2em} \textbf{for} \textit{j} := 1 \textbf{to} \textit{n} \textbf{do} \\
\hspace{4em} \textbf{begin} \\
\hspace{6em} \textit{send}(\textit{f, ask\_for\_entering, i, j, counter, empty field}); \hspace{1em} \{fields are signed by the system\} \\
\hspace{6em} \textit{send}(\textit{f, shadow, i, j, counter, I}_j(s)); \hspace{1em} \{fields are signed by the system\} \\
\hspace{4em} \textbf{end}; \\
\hspace{2em} \textbf{wait until} arriving of \textit{t + 1} answers \textit{(shadow, j, i, counter, I}_j(s)) from \textit{t + 1} \textit{G}_j's; \hspace{1em} \{the signatures over \textit{I}_j(s) and of \textit{G}_j over \textit{(j, i, I}_j(s), \textit{counter}) are checked\} \\
\hspace{2em} \textit{construct} \textit{s} from the \textit{I}_j(s); \hspace{1em} \{using Lagrange's interpolation polynomials\} \\
\hspace{2em} \textit{in} := \textit{true}; \\
\hspace{2em} \textit{jump\_to(s)}; \hspace{1em} \{enter the critical section\} \\
\textbf{end}; \\

**Interrupt-handler for** \textit{svc(exit\_critical)};

\textbf{var} \hspace{1em} \textit{i, j}: site; \\
\hspace{2em} \textit{f}: message; \\
\hspace{2em} \textit{s}: address; \\
\textbf{begin} \\
\hspace{2em} \textbf{while} \textit{in} = \textit{false} \textbf{do nil}; \hspace{1em} \{G, did not make an SVC for entering\} \\
\hspace{2em} allocate\textit{(critical\_section, s)}; \hspace{1em} \{system primitive\} \\
\hspace{2em} \textbf{for} \textit{j} := 1 \textbf{to} \textit{n} \textbf{do} \\
\hspace{4em} \textbf{begin} \\
\hspace{6em} \textit{send}(\textit{f, exit, i, j, counter, empty field}); \hspace{1em} \{just an acknowledgment\} \\
\hspace{4em} \textbf{end}; \\
\hspace{2em} \textit{continue}; \hspace{1em} \{continues with \textit{G}_j\} \\
\textbf{end};
Recall that it is assumed that the two interrupt handlers are always correctly executed, by any process.

**Theorem 6.** If the number of locally byzantine processes does not exceed \( n/3 \), then the protocol above solves the mutual exclusion problem.

**Proof.** The proof is based upon the three following properties of the protocol.

(a) **Mutual exclusion.** Assume that two processes \( G_i \) and \( G_j \) are simultaneously (for a hypothetical global clock) in their own critical section. Each of them has necessarily entered its critical section by an svc, in some fixed order, for instance \( G_i \) first and \( G_j \) second. In order to build the address of the segment containing its critical section, \( G_i \) has received at least \( t + 1 \) shadows \( I_i(s) \). Among those \( t + 1 \) responding processes, at least one is a correct one, \( G_k \). Since \( G_k \) sent \( I_i(s) \) and \( G_k \) is correct, \( G_k \) obtained, together with all correct processes, an agreement on \( G_i \). Thus, each correct process is executing:

```plaintext
while ask[i].date > acq[i].date do nil;
```

since \( G_i \) is still in its critical section.

Consequently, no correct process sent \( I_j(s) \) to \( G_j \). So \( G_j \) received at most \( t \) shadows (from incorrect processes) and, contradictory to the hypothesis, is unable to know the segment address of its critical section.

(b) **Progress.** Assume that \( G_i \) executes a primitive \( svc(ask\_for\_critical\_section) \) and let \( G_j \) be the first process to enter its critical section after this system call.

Since the only way for a process to enter its critical section is through the Interrupt handler for \( svc(ask\_for\_critical\_section) \), \( G_j \) requested to enter its critical section. So, \( G_i \) cannot be delayed by processes executing forever in their remainder sections.

(c) **Bounded waiting.** Assume that the process \( G_i \) executes the primitive \( svc(ask\_for\_critical\_section) \). Since there is no loss of messages, the messages (ask for entering, \( i, j, \) counter, empty field) will eventually reach their destinations. When the last of these messages arrives, each correct process \( G_j \) has: \( ask[i].flag := true \). With a maximum number of \( n - 1 \) executions of the inner loop of \( byz\_cons \), allowing at most \( n - 1 \) processes to enter their critical sections, the agreement is reached on \( (i, ask[i].date) \). So \( G_i \) can enter its critical section after at most \( n - 1 \) other distinct processes enter their critical sections.

### 3.3. Complexity

The number of exchanged messages necessary for a process that requests to enter its critical section has the same order of magnitude than the number of exchanged messages necessary for reaching a byzantine agreement and is in \( O(n^2) \).
4. Conclusion

We have first presented a method for assigning to each site of a network an unique identifier. This identifier is a rather long character string and can seem not to be of a great practical interest. Nevertheless, there do exist renaming algorithms, that allow a reduction in the size of the initial name space, even if some processes are faulty [1] and a sequel to this work should be to merge our solution with such an algorithm.

Then, we have given an authenticated solution to the mutual exclusion problem. In [13], a methodology is given for deriving non-authenticated algorithms from algorithms using digital signatures. This methodology could possibly be applied here, giving a non-authenticated algorithm that solves the mutual exclusion problem for locally byzantine processes.

Appendix A

\begin{verbatim}

type message: string of char,
site: integer;
const \( D = \ldots \); \{diameter of the network\}
\( \delta = \ldots \); \{maximum transmission delay\}
var identifier: string of char;
valid: set of message;

procedure timer (delay: integer);
begin \{initializes the watchdog timer to the value \( delay \); at the expiration of the
delay an interrupt aborts the calling procedure\} end;

procedure send (m: message) to (s: site);
begin \{send the message \( m \) onto the communication line numbered \( s \)\} end;

Appendix B

\begin{verbatim}

const \( maxlength = \ldots \); \{maximum length of a message\}
\( memory\_max = \ldots \); \{range of the primary memory of the host computer\}

type bit = 0 .. 1;
site = 1 .. n; \{n is the number of sites\}
time\_stamp: integer;
data = packed array [0, maxlength] of bit;
addr = 1 .. memory\_max;
svc\_type = (ask\_for\_critical\_section, exit\_critical\_section);
\{types of system call\}
\end{verbatim}

\end{verbatim}
consensus_value = (1, 2, \ldots, n, \text{no-demand}, \text{default_value});

consensus = \text{record}
  \begin{align*}
  \text{value:} & \quad \text{consensus_value}; \\
  \text{counter:} & \quad \text{time\_stamp};
  \end{align*}
\text{end};

message_kind = (\text{ask\_for\_entering}, \text{exit}, \text{shadow});

message = \text{record}
  \begin{align*}
  \text{kind:} & \quad \text{message\_kind}; \\
  \text{sender:} & \quad \text{site}; \\
  \text{destination:} & \quad \text{site}; \\
  \text{counter:} & \quad \text{time\_stamp}; \\
  \text{info:} & \quad \text{data}
  \end{align*}
\text{end};

note = \text{record} \quad \{\text{for bookkeeping asks and exitings of critical} \\
\quad \text{section}\}
  \begin{align*}
  \text{flag:} & \quad \text{boolean}; \\
  \text{date:} & \quad \text{time\_stamp}; \\
  \text{info:} & \quad \text{data}
  \end{align*}
\text{end};

control_array = \text{array} [1..\text{site}] \text{ of note};

\textbf{procedure} \text{svc}(\text{var} t: \text{svc\_type});
\textbf{begin} \{\text{generates an interrupt and a context-switching, causing the process status} \\
\text{word register to be saved on the stack and an interrupt handler to be executed under} \\
\text{control of the local operating system}\} \textbf{end};

\textbf{procedure} \text{continue};
\textbf{begin} \{\text{loads the top value of the stack in the processor status word register, including} \\
\text{the program counter, of the host computer}\} \textbf{end};

\textbf{procedure} \text{allocate} (\text{var} \text{proc}: \text{procedure}, \text{s:} \text{addr});
\textbf{begin} \{\text{allocates a new memory segment to the procedure proc and returns the base} \\
\text{address in the variable s}\} \textbf{end};

\textbf{procedure} \text{jump\_to} (\text{var} \text{s:} \text{addr});
\textbf{begin} \{\text{settle the value s in the program counter}\} \textbf{end};

\textbf{procedure} \text{send} (f: \text{message});
\textbf{begin} \{\text{transmits the message f to the site f.destination}\} \textbf{end};

\textbf{procedure} \text{threshold} (\text{var} \text{s:} \text{address}, \text{t:} \text{integer}, \text{n:} \text{site});
\textbf{begin} \{\text{starting from an address s and two integers n and t, t < n, this primitive} \\
\text{produces n pieces of information, I_1(s), I_2(s), \ldots, I_n(s), called the shadows of s,} \\
\text{such that the knowledge of any set of at least t pieces and of no set of strictly less} \\
\text{than t pieces allows to built s. The n shadows of s are signed by the system executing} \\
\text{this primitive}\} \textbf{end};

\text{Shamir} [12] \text{ and Denning} [5] \text{ give several methods for building this primitive.}
Distributed problems involving byzantine processes

procedure byzantine agreement (var counter: time_stamp, initial_value, final_value: consensus);
begin {reaches a byzantine agreement, by only considering messages that are time-
stamped by the value counter, with initial_value as initial message. The result of the
agreement protocol is in final_value. The default value chosen for this primitive will
be the string: “default value”) end;
Since communications are assumed to be asynchronous, this primitive can be imple-
mented by the probabilistic solution of Rabin [8] or its improved version Toueg [14]. These solutions guarantee exact agreement within a finite expected number of
rounds, provided the number of incorrect processes does not exceed n/3 (cf. [14]).

References

[4] T. Cheung, Graph traversal techniques and the maximal flow problem in distributed computation,
(IEEE, 1988) 82-86.
(1980) 228-234.
403-409.
Distributed Computing, Vancouver, Canada, August 1984.