Abstract:

In this paper we describe the design of a reliable Remote Procedure Call mechanism intended for use in Local area networks. Starting from the hardware level that provides primitive facilities for data transmission, we describe how such a mechanism can be constructed. We discuss various design issues involved, including the choice of a message passing system over which the remote call mechanism is to be constructed and the treatment of various abnormal situations such as lost messages and node crashes. We also investigate what the reliability requirements of the Remote Procedure Call mechanism should be with respect to both the application programs using it and the message passing system on which it itself is based.

The Design of a Reliable Remote Procedure Call Mechanism

By

S.K. Shrivastava and F. Pansieri

TECHNICAL REPORT SERIES

Editor: Mr. M.J. Elphick

Number 171
October, 1981

© 1981 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne,
Computing Laboratory, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.
In this paper we describe the design of a reliable Remote Procedure Call mechanism intended for use in local area networks. Starting from the hardware level that provides primitive facilities for data transmission, we describe how such a mechanism can be constructed. We discuss various design issues involved, including the choice of a message passing system over which the remote call mechanism is to be constructed and the treatment of various abnormal situations such as lost messages and node crashes. We also investigate what the reliability requirements of the Remote Procedure Call mechanism should be with respect to both the application programs using it and the message passing system on which it itself is based.
SHRIVASTAVA, Santosh Kumar
The design of a reliable Remote Procedure Call mechanism.
Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1981.
(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 171.)

Added entries
PANZIERI, Fabio
UNIVERSITY OF NEWCASTLE UPON TYNE.

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404
U.D.C. 519.687

Suggested keywords

ATOMIC ACTIONS
DATA COMMUNICATIONS
DISTRIBUTED SYSTEMS

FAULT TOLERANCE
LOCAL AREA NETWORKS
RELIABILITY

Abstract
In this paper we describe the design of a reliable Remote Procedure Call mechanism intended for use in local area networks. Starting from the hardware level that provides primitive facilities for data transmission, we describe how such a mechanism can be constructed. We discuss various design issues involved, including the choice of a message passing system over which the remote call mechanism is to be constructed and the treatment of various abnormal situations such as lost messages and node crashes. We also investigate what the reliability requirements of the Remote Procedure Call mechanism should be with respect to both the application programs using it and the message passing system on which it itself is based.

About the author
Dr. Shrivastava joined the Computing Laboratory of the University of Newcastle upon Tyne in August 1975, where he is currently a Lecturer.

Mr. Panzieri joined the Computing Laboratory of the University of Newcastle upon Tyne in June 1979, where he is currently a Research Associate.
1. Introduction

In this paper we describe the design of a reliable Remote Procedure Call (RPC) mechanism which we have been investigating within the context of programming reliable distributed applications. In the following we consider a distributed system as composed of a number of interacting "client" and "server" processes running on possibly distinct nodes of the system; the interactions between a client and a server are made possible by the suitable use of the RPC mechanism. Essentially in this scheme, a client's remote call is transformed into an appropriate message to the named server who performs the requested work and sends the result back to the client and so terminating the call. The RPC mechanism is thus implemented on top of a message passing interface. Some of the interesting problems that need to be faced are: i) the selection of appropriate semantics and reliability features of the RPC mechanism, ii) the design of an appropriate message passing interface over which the RPC is to be implemented, and iii) the treatment of abnormal situations such as node crashes. These problems and their solutions are discussed in this paper. We shall concentrate primarily on the relevant reliability issues involved, so other directly or indirectly related issues such as type checking, authentication and naming will not be addressed here.

The RPC mechanism described in the following has been designed for a local area network composed of a number of PDP 11/45 and LSI 11/23 computers (nodes) interconnected by the Cambridge Ring[1]; each node runs the Unix (V7) operating system. However, most of the ideas presented in this paper are, we believe, sufficiently general to be applicable to any other local area network system.
2. An Overview of Reliability Issues in Distributed Programming

In this section we briefly review the main reliability problems in distributed programming and discuss which of these problems need closer attention during the design of an RPC mechanism.

In this discussion we shall concentrate upon a distributed system consisting of a number of autonomous nodes connected by a local area network. A node in such a network will typically contain one or more processes providing services (e.g. data retention) that can be used by local or remote processes. We shall refer to such processes as "servers" and "clients" respectively. So an execution of a typical application program will give rise to a computation consisting of a client making various service requests to servers. These service requests take the form of procedure calls - if a server is remote then the calls to it will be remote procedure calls. In the rest of the paper we will assume the general and more difficult case of remote calls (note however that it is possible to hide the "remoteness" of servers by providing a uniform interface for all service calls). It can be seen that we have adopted a "procedure based" model of computation rather than a "message based" model. It has been pointed out that these two models appear to be duals of each other[2]. Bearing this in mind, we have chosen to support the first model because this allow us to directly apply the existing knowledge on the design and development of programs to distributed systems.

Let us ignore, for the time being, any reliability problems in the mechanisation of a suitable RPC facility and concentrate upon the reliability problems at the application program level. The most vexing problem is to do with guaranteeing a clean termination of a program despite breakdowns (crashes) of nodes and communication subsystems. It is now well known that this can be achieved by structuring a program as an atomic action with the following "all or nothing" property: either all of the client's requested services are performed or none are[3,4,5]. Thus a program terminates either producing the intended results or none at all. In a distributed system the implementation of atomic actions require the provision of a special protocol, such as the two phase commit protocol [3,4], to co-ordinate the activities of clients and servers. In addition, some recovery capability is also needed at each node to "undo" any results produced at that node by an ongoing atomic action that is subsequently to be terminated with null results. We shall not discuss here the details of how the various facilities needed for the provision of atomic actions can be constructed - they are well documented in the already cited references - but draw the reader's attention to figure 1 which shows a typical hierarchy of software interfaces.
Figure 1. Hierarchy of software interfaces.

The point to note is that the atomic action software that maintains L3 contains major reliability measures for application programs (undo capability, two phase commit). This has important consequences on the design of RPC—in particular in choosing its semantics and reliability capability.

Figure 2, which shows the bare essentials of an RPC mechanism, will be used to illustrate the reliability problems.

```
CLIENT
send(...);

--->
SERVER

receive(...);

"work"

```

Figure 2. A simple RPC algorithm.

The send and receive primitives provide a message handling facility (the precise semantics of which are not relevant in the following discussion). Suppose that the message handling facility is such that messages occasionally get lost. Then a client would be well justified in resending a message when it "suspects" a loss. This could sometimes result in more than one execution at the server. To take another case, suppose that the client's node crashes immediately after the server starts to perform the requested work. Suppose now that the client's node "comes up" again and the client re-issues the remote call: this again gives rise to the possibility of repeated executions at the server (the above situation can occur even if messages never get lost). If a client is not aware of the fact that repeated executions have taken place, then many of a server's executions will be in vain, with no client to receive
the sent responses. Such executions have been termed orphans by B.
Lampson who has also devised some ingenious schemes for detecting and
treating orphans[3][6]. The above mentioned problems have led B. Nelson
to classify the semantics of remote procedure calls as follows[7]:

(a) "Exactly once" semantics: If a client's call succeeds (i.e. the
call does not return abnormally) then this implies that exactly one
execution has taken place at the server; this is of course the
meaning associated with conventional procedure calls.

(b) "At least once" semantics: If a client's call succeeds then this
implies that at least one execution has taken place at the server.
Further subclassification is also possible (e.g. first one or last
one) indicating which execution is responsible for the termination
of a call.

To start with, it should be clear that out of the two, (a) has the
more desirable semantics but is also the more difficult of the two to
achieve. An approach that has been widely used (see for example [3]) is
to adopt - for the sake of simplicity at the RPC level - the "at least
once" semantics and to make all of the services of servers idempotent
(that is, repeated executions are equivalent to a single execution).
Thus the problems of repeated executions and orphans can be more or less
ignored. The major shortcoming of this approach is that it is no longer
possible to provide servers with arbitrary services (e.g. a server can
not provide services that include increment operations); for this reason
we have rejected this option in our RPC design and have chosen instead
the "exactly once" semantics. This has been achieved by introducing
sufficient measures at the RPC level to enable processes to reject
unwanted messages arising from a call. This capability is not enough to
cope with orphans though, since as stated before, a client's crash can
result in more than one remote call directed at a server when only one
call was intended. We treat orphans at the next level (in the software
supporting L3, see figure 1) by insisting that all programs that run
over L3 be atomic actions with the "all or nothing" property. In partic-
lar this means that any executions at servers be atomic as well.
This atomicity criteria implies that repeated executions at a server are
performed in a logically serial order with orphan actions terminating
without producing any results. This is the basis of the work presented
in [5] where the techniques needed for the construction of level L3,
given the existence of L2, are described (a broadly similar approach
has, we understand, been independently developed by B. Liskov's group at
MIT[8]).

To sum up this section: (i) we have chosen the exactly once seman-
tics for our RPC, (ii) the main reliability feature needed at the RPC
software is that necessary to discard any unwanted messages, and (iii)
any other reliability features necessary for ensuring proper executions
of application programs are added not at the RPC level but at the next
level concerned with the provision of atomic actions. In the design of
the RPC mechanism to be presented we have followed the rule of keeping
each level as simple as possible; this has been achieved by making reli-
ability mechanisms application specific rather than general as argued in[9].
3. Communications Support for RPC: Datagram versus Transport Service

The implementation of RPC requires that the underlying level supports some kind of Interprocess Communication (IPC) facility. On the one hand this facility could be quite sophisticated with features such as guaranteed, undamaged and unduplicated delivery of a message, flow control and end to end acknowledgement. An interface supporting such features is usually said to provide a transport service for messages. On the other hand, the IPC facility could be rather primitive, lacking most of the above desirable properties. An interface supporting such an IPC mechanism is said to provide a datagram service[10].

Transport services are designed in order to provide fully reliable communication between processes exchanging data (messages) over unreliable media – they are particularly suitable for wide area packet switching networks which are liable to damage, lose or duplicate packets. The implementation of a ‘transport layer’ tends to be quite expensive in terms of resources needed since a significant amount of state information needs to be maintained about any data transfer in progress. The initialisation and maintenance of this state information is required to support the abstraction of a ‘connection’ between processes. To establish, maintain and terminate a connection reliably is rather complex and a significant number of messages are needed just for connection purposes[11].

On the other hand, a datagram service provides the facility of the transmission of a finite size block of data (a message known as a datagram) from an origin address to a destination address. In its simplest form, the datagram service does not provide any means for flow control or end to end acknowledgements; the datagram is simply delivered on a ‘best effort’ basis. If any of the features of the transport service are required, then the user must implement them specifically using the datagram service.

At a superficial level, it would seem that a good way to construct a reliable RPC would be to start with a reliable message service – i.e. a transport service. However we reject this viewpoint and adopt the datagram service as the more desirable alternative. The argument for this decision is as follows. To start with, it must be noted that in the distributed system previously mentioned, the users are not given the abstraction of sending or receiving messages; rather only a very specific piece of software – that needed to implement RPC – is the sole user of messages. As such the full generality of the transport service is not needed. The provision of the transport service entails a considerable reduction of the available communication bandwidth (this is because of the overheads of connection management and the need for end to end acknowledgement). We may be able to utilise this bandwidth more effectively by reducing the need for connection management and acknowledgements as much as possible. This is indeed feasible in typical local area networks since the underlying hardware – the Cambridge Ring in our case – provides a reliable means of data transmission. So, a fairly reliable datagram service (whereby every datagram is delivered with a high probability to its destination address) can certainly be built on top of the hardware interface. Any additional facilities
needed are then specifically implemented making the implementation of the RPC a bit more complex but highly efficient. Hence we conclude that it is appropriate to give the software of the RPC mechanism the responsibility of coping with any unreliabilities of a datagram service. In the next section we will describe the specific datagram service to be implemented over the Cambridge Ring hardware in order to support our RPC mechanism.

4. The Hardware and the Datagram Service

The Cambridge Ring hardware provides its users with the ability to transmit and receive packets of a fixed size between nodes connected to the Ring - each transmitted packet is individually acknowledged. At the Ring level each node is identified by a unique station address. The following two primitive operations are available[12]:

(i) transmit-packet (destination:...; pkt:...; var status:...)
where the acknowledgement is encoded as

status = (OK, unselected, busy, ignored, transmission-error)

The meaning of `status` is as follows:

status = OK: The destination station has received the packet.

status = unselected: The packet was not accepted by the destination station because that station was `listening` to some other source station.

status = busy: The packet was not accepted by the destination station because that station was `deaf` (not listening to anyone). Note that either of the above two status conditions implies that the destination station is most likely to become available shortly.

status = ignored: The packet was not accepted because the destination station was not on-line. This indication can be taken to mean that there is little chance of packets being accepted by that station for a while.

status = transmission-error: The packet got corrupted somewhere during its passage through the Ring. This is the only case where the response of the destination station is not known.

It is worth mentioning here that the transmit primitive does not have a time-out response associated with it. As a consequence, the execution of this primitive will not return if the packet is not acknowledged due to a fault in the Ring hardware.

(ii) receive-packet ( var source:...; var pkt:...)

The receive primitive allows for the reception of packets either from any source station on the Ring or from a specific source (a special operation is provided by the Ring for setting up a station in either of the modes). In either case, `source` will contain the identity of the
sender with "pkt" containing the received packet. A curious aspect of
the Ring is that each node has a parity error detection logic but nei-
ther the sender nor the receiver of a packet get any indication when a
parity error is detected in a packet (this does not matter all that much
in reality as the probability of a packet getting corrupted has been
shown to be very low).

We shall assume that all of the hardware components (e.g. Ring,
processors) either perform exactly as specified or a component simply
does not work (so for example, for the Ring, a "send" or "receive"
operation will not terminate). If this assumption were realistic then
the design to follow has some very nice reliability properties. However
unpredictable behaviour of the hardware interface (i.e. a behaviour that
does not meet the specification) is likely to result in the same at the
RPC/user interface to the extent that guaranteed behaviour can not be
promised.

The proposed datagram service will provide its user (a process)
with the ability of (i) sending a block of data to a named destination
process; and (ii) receiving a block of data from a specific or any pro-
cess. We shall ignore here the fine details of how this may be imple-
mented using the Ring operations described earlier; only the properties
of the datagram service primitives will be described:

(i) send msg (destination:...; message:...; var status:...)
    where,
    status = (OK, absent, not-done, unable)

The message is broken into packets and transmitted to the home sta-
tion of the destination process. If all these packets are accepted by
the station then "status=OK" will hold. Note that this only means that
the message has reached the station, and not that it has been accepted
by the destination process. If a packet is not accepted (possibly even
after a few retries) by the station (packet level response is
"unselected" or "busy") then "status=not-done" will hold. A packet
level response of "ignored" is translated as "status=absent" indicating
that the destination process is just not available. The last two
responses indicate that the message was not delivered. A time-out
mechanism will be needed to cope with Ring malfunctions during the
transmission of a message. The "unable" status holds either if the time
out expires or if a packet level "transmission-error" response is
obtained. The "unable" response indicates inability of the datagram
layer to deliver a message properly (the message may or may not have
reached the destination station).

(ii) receive msg ( source:...; var msg:...)

The above primitive is to receive a message from a specified
"source" process. This primitive is implemented by repeatedly making use
of the receive-packet(...) primitive. A time-out mechanism will be
needed to detect an incomplete message transmission and Ring failures.
Any corruption of the sent message can be detected if the sender
includes a checksum in the message and appropriate computation is per-
formed at the receiver; corrupted messages are simply discarded. So,
the receive_msg(...) primitive only delivers a "good" message (if any).

The receive_msg(...) primitive can also be used for receiving messages from any source by simply specifying "source" parameter as "any".

The datagram service described above is based on the Basic Block Protocol designed at Cambridge[13].

5. RPC Mechanism

A client invokes the following primitive to obtain a service from a server (where the "time-out" parameter specifies how long the client is willing to wait for a response to his request):

remote_call(server:...;service:...;var result:...;var status:...;time-out:...)

where
status = (OK, not-done, absent, unable)
and parameters and results are passed by value.

The meaning of the call under various responses is given below:

status = OK: The service specified has been performed (exactly once) by the server and the answers are encoded in "result".

status = not-done: The server has not performed the service because it is currently busy (so the client can certainly reissue the call in the hope of getting an "OK" response).

status = absent: The server is not available (so it is pointless for the client to retry).

status = unable: The parameter "result" does not contain the answers; whether the server performed the service is not known. The action of the client under this situation will depend typically on the property of the requested service. If the service required has the idempotency property then the client can retry without any harm, otherwise backward recovery should be invoked to maintain consistency. How this is achieved is not relevant here; it is sufficient to observe, as noted in section 2, that the consistency and recovery problems could be handled within the framework of the two phase commit protocol and atomic actions.

We believe that these responses are meaningful, simply understood and quite adequate for robust programming. We shall show next that it is possible to design RPC with the above properties based on our datagram service despite numerous fault manifestations in the distributed system (including node crashes). A skeleton program showing only the essential details of the RPC implementation is shown in figure 3; it should be self-explanatory. The following two assumptions will be made in the ensuing discussion: (i) some means exists for a receiver process to reject unwanted (i.e. spurious, duplicated) messages; the next section contains a proposal for achieving this goal; (ii) node crashes amount to that station being not on line. We now consider the treatment
of various responses obtained during message handling.

<table>
<thead>
<tr>
<th>CLIENT</th>
<th>SERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>remote-call(...) corresponds to the following code:</td>
<td>cycle</td>
</tr>
<tr>
<td>___</td>
<td>___</td>
</tr>
<tr>
<td>send_msg();</td>
<td>--&gt;</td>
</tr>
<tr>
<td>&quot;send service request&quot;</td>
<td></td>
</tr>
<tr>
<td>___</td>
<td>___</td>
</tr>
<tr>
<td>set(time-out)</td>
<td>___</td>
</tr>
<tr>
<td>repeat</td>
<td>___</td>
</tr>
<tr>
<td>receive_msg();</td>
<td>___</td>
</tr>
<tr>
<td>until msg=valid;</td>
<td>___</td>
</tr>
<tr>
<td>___</td>
<td>___</td>
</tr>
</tbody>
</table>

**Figure 3. The RPC Algorithm.**

(a) The client sends a service request: Recall that a send_msg(...) can return the response "OK", "absent", "not done" or "unable". If the response is "OK", the control goes to the "set(time-out)" statement. If the response is "absent" then the execution of remote_call(...) terminates with "status=absent". If the response is "not-done" then the message is sent again. If after a few retries the same response is obtained then the execution of remote_call(...) terminates with "status=not-done". A few retries can also be made if the send_msg(...) results in an "unable" response. If this response still holds after retries then the execution of remote_call(...) terminates with "status=unable". Note that it is all right to send a message repeatedly. The server is in a position to discard any duplicates.

(b) The client waits for a message: The client prepares to wait for a response from the server. A time-out is set to stop the client from waiting for ever; the maximum duration of the waiting is as specified in the last parameter of the remote_call(...). All the unwanted messages are discarded. A client may get such messages for example as a result of the actions performed by that node before it crashed and came up again. If a valid message is received then the execution of remote_call(...) terminates with "status=OK" and "result" containing the answer. If the time-out expires then the execution of the remote_call(...) terminates with "status=unable". Note that "unable" response can be obtained for several reasons: server did not receive the message, server node crashed, server's message not received because of a Ring fault, etc.
(c) Server waits for a service request: Any spurious, in particular duplicated, messages are rejected. This guarantees that despite the possibility of repeated requests being sent by a client, only one service execution will take place.

(d) Server sends the reply: If the execution of send\_msg(...) results in an "OK" response then the server is ready for the next request it goes to the beginning of its cycle. Note that it is not guaranteed that the client will receive the reply, rather it implies that most probably the reply has reached the client. If the "send" operation gives rise to the "absent" response then "unable" is signaled to the server. If the "send" operation gives rise to either a "not-done" or an "unable" response then the message can be re-sent a few times before accepting defeat by signaling "unable" to the server. The reason for mapping all the three abnormal responses of send\_msg(...) on to a single "unable" response is based on the belief that it is of little interest to a server as to why he was not able to deliver the result satisfactorily (this response means that most probably the client did not receive the reply). As before, any recovery actions of the server will be handled within the framework of atomic actions and the two phase commit protocol.

We conclude that the level concerned with RPC implementation provides three operations: (i) remote\_call(...) - this is the client half of the program with the semantics discussed earlier; (ii) get\_work(...) - this corresponds to the repeat loop code of the server; and (iii) send\_result(..., status) - this corresponds to the code concerned with sending of the results, with status=(OK, unable), where the "absent", "not-done and "unable" responses of send\_msg(...) are all mapped onto "unable" response of send\_result.

It should be noted that if fault manifestations are rare and messages are delivered with a high probability then almost always, only two messages are required for RPC (there are no hidden overheads). This is not possible if the transport service is used for message passing.

Finally, the treatment of various normal and abnormal responses is summarised in a pictorial form in figure 4.

6. Generation of Sequence Numbers

In the previous section it was assumed that a receiver is always in a position to reject unwanted messages; this can be arranged by appropriately assigning sequence numbers (SNs) to messages. The problem of sequence numbering of messages is fairly complex if tolerance to node crashes is required. In essence, it is necessary for a process of a node, that has "come up" after a crash, to be able to distinguish those incoming messages that have originated as a result of any actions performed before the crash. This typically requires maintaining relevant state information on a "crash proof" storage. This is a complicated and expensive process, so a scheme that has minimum crash proof storage requirements is to be preferred. A transport level is designed to cope with sequence numbering problems and users are not concerned with them;
Figure 4. The treatment of responses at various levels in a node. However in our case they need to be generated explicitly within the RPC level. There can be three approaches to the generation and assignment of SNs:

(a) SNs are unique over a given client-server interaction: this would be the approach implicitly taken by a transport level supported RPC. This is a fairly complex approach requiring the maintenance of a relatively large amount of state information that has to survive crashes [11].

(b) SNs are unique over node to node interactions: rather than maintaining state information on a process to process basis, it is possible to maintain information on a node to node basis only. Clearly, it is less demanding than (a) above, in its requirements for crash proof storage.
(c) SNs are unique over the entire system: if SNs are made unique over the entire network then a very simple scheme suggests itself. A server need only maintain "the last largest SN received" in a crash proof storage (and as we shall see, even this requirement can be dispensed with). Further, all the retry messages are sent by a sender with the same SN as the original message. If a server accepts only those messages whose SN is greater than the current value of "last largest SN" then it is easy to see now that we have the server property assumed in the previous section (that of rejecting unwanted messages). A similar approach is necessary at the client's end.

We have chosen to incorporate the third method in our design because, as indicated above, coping with node crashes is comparatively easier in such a technique. Two of the best known techniques for the generation of network wide unique sequence numbers are based on i) the circulating token method of Le Lann[14] and ii) the loosely synchronised clock approach of Lamport[15]. In the former all of the nodes are logically connected in a ring configuration and an integer valued "token" circulates round the ring in a fixed direction. A node that wants to send a message waits for the token to arrive, then it copies its value, increments the value of the token and passes it on to the next node. The copied value can be used for sequence numbering. In the latter method each node is equipped with a clock and each node is also assigned a unique "node number". A sequence number at any node is the current clock value concatenated with the node number. For "acceptable behaviour" (see later) it is necessary that the clock values at various nodes be approximately the same at any given time. This is achieved as follows: whenever a process at say node $n_i$ receives a message, it checks the SN of that message with the current SN of $n_i$; if SN (received) is greater or equal to SN($n_i$) then the clock of $n_i$ is advanced by enough ticks to make SN($n_i$) greater than SN(received).

Out of the above two methods, we have adopted the second in our system for the following two reasons: i) because of the kind of message facility we are using, it will not be easy for a node to find out whether its sent token has been received by the next node or not; as a result the detection of the lost token is not a straightforward process; and ii) the algorithm for the reinsertion of the token - which must ensure that only one token gets inserted - is rather complex. In comparison, as we shall see, Lamport's technique can be made to tolerate lost messages and node crashes in a straightforward manner. We shall next describe how we have incorporated Lamport's technique into our design.

The SN at a node at any time is constructed out of the time of day and calendar clock of the node and the Ring station number:

\[
\text{SN} = \begin{array}{c|c|c}
\text{time and date} & \text{station number} \\
\end{array}
\]
The SN of a node is maintained by a monitor[16] that provides the following two procedures:

```plaintext
get_SN(var s_number; ...);
```

This procedure returns the SN:

```plaintext
update_SN(s_number; ...);
```

The SN at the monitor is compared with passed sequence number and the clock of the node adjusted as described earlier. The SN's are used in the RPC algorithm as depicted in figure 5.

```plaintext
CLIENT
begin

---

llsn := get_SN(...);
"llsn = last largest seq. no."

---

I := get_SN(...);

send_msg(...);
"message includes i"

---

set(time-out)

---

repeat

receive_msg(...)

until msg.SN = I;

---

send_msg(...);

---

end

 cycle

 ---

repeat "get work"

receive_msg(any, ...)

until msg.SN > llsn;

llsn := msg.SN;

---

"perform work"

---

end

---

Figure 5. Using sequence numbers.
```

Strictly speaking, in our system there is no logical requirement that the various clocks be "approximately the same". However, in the absence of such a situation, a client with a slower clock will have difficulty in obtaining services since his requests will stand a higher chance of rejection by servers. Hence it is necessary that each node regularly receives messages from other nodes so that it can keep its clock value nearer to those of others. For this purpose, we maintain two processes at each node (see figure 6):
The "broadcaster" process of a node regularly (say once every few minutes) sends its sequence number to all of the remote clock-synchroniser processes. A few retries can be made if a send operation returns a "not-done" or an "unable" response. If these responses persist or an "absent" response is obtained, then no further attempt is made to send the message to that clock-synchroniser in that cycle (at this level, a crashed node in no way affects the non-crashed nodes).

We shall now discuss how our sequence numbering scheme can be made to tolerate node crashes economically. We can avoid the need for any crash proof storage for our scheme by being careful during the start-up phase of a node after a crash. In a centralised system, when the computer system is started up, the operator inputs the time and date to the system clock. This is not desirable in our system since careless clock updates can introduce problems. For example, entering "future time and date" will eventually affect the rest of the system in that all clocks will become "inaccurate" in the sense that they will not represent physical time (logically this is irrelevant). Also, entering a "past time and date" can result in the acceptance of wrong messages. We simply insist that the "clock-synchroniser" process be the only process (with one exception, see below) that can update the clock. So, when a node comes up, eventually (within a few minutes) it will be able to get an appropriate clock value. An important requirement is that a node, when it comes up, should have its clock initialised to zero. The only drawback of the above scheme is that if all the other nodes are down (presumably a rare event) then our node will never get a clock value. This problem can be solved by giving some privileged user the authority for clock updates.

We conclude this section by summarising the net effect of our
The SN of a node is maintained by a monitor[16] that provides the following two procedures:

get_SN(var s_number: ...);
this procedure returns the SN;
update_SN(s_number: ...);

The SN at the monitor is compared with passed sequence number and the clock of the node adjusted as described earlier. The SN's are used in the RPC algorithm as depicted in figure 5.

CLIENT

begin

---

l1sn := get_SN(...);
"l1sn = last largest seq. no."

---

send_msg(...);
"message includes i"

---

set(time-out)

repeat
receive_msg(...)
until msg.SN = i;
"msg sent with SN = l1sn"

---

end

SERVER

begin

---

l1sn := get_SN(...);

---

cycle

repeat "get work"
receive_msg(any, ...)
until msg.SN > l1sn;
l1sn := msg.SN;
"perform work"

---

end

---

Figure 5. Using sequence numbers.

Strictly speaking, in our system there is no logical requirement that the various clocks be "approximately the same". However, in the absence of such a situation, a client with a slower clock will have difficulty in obtaining services since his requests will stand a higher chance of rejection by servers. Hence it is necessary that each node regularly receives messages from other nodes so that it can keep its clock value nearer to those of others. For this purpose, we maintain two processes at each node (see figure 6):
Figure 6. Loose synchronisation of clocks.

The "broadcaster" process of a node regularly (say once every few minutes) sends its sequence number to all of the remote clock-synchroniser processes. A few retries can be made if a send operation returns a "not-done" or an "unable" response. If these responses persist or an "absent" response is obtained, then no further attempt is made to send the message to that clock-synchroniser in that cycle (at this level, a crashed node in no way affects the non-crashed nodes).

We shall now discuss how our sequence numbering scheme can be made to tolerate node crashes economically. We can avoid the need for any crash proof storage for our scheme by being careful during the start up phase of a node after a crash. In a centralised system, when the computer system is started up, the operator inputs the time and date to the system clock. This is not desirable in our system since careless clock updates can introduce problems. For example, entering "future time and date" will eventually affect the rest of the system in that all clocks will become "inaccurate" in the sense that they will not represent physical time (logically this is irrelevant). Also, entering a "past time and date" can result in the acceptance of wrong messages. We simply insist that the "clock-synchroniser" process be the only process (with one exception, see below) that can update the clock. So, when a node comes up, eventually (within a few minutes) it will be able to get an appropriate clock value. An important requirement is that a node, when it comes up, should have its clock initialised to zero. The only drawback of the above scheme is that if all the other nodes are down (presumably a rare event) then our node will never get a clock value. This problem can be solved by giving some privileged user the authority for clock updates.

We conclude this section by summarising the net effect of our
sequence number assignment and generation scheme on the fault tolerant behaviour of our RPC mechanism: (i) since none of the state information of a call is maintained on a crash proof storage, a call does not survive a crash; (ii) the clock management scheme ensures that any messages belonging to a "crashed call" are ignored.

7. Implementation notes

The design presented here is currently being implemented on our Unix systems using the C language. We present enough details here to convince the reader that the implementation requires straightforward Unix system programming (the details of message formats and the like are not relevant and will be ignored). One of the main problems requiring attention is that of buffer management for message queues. Currently we envisage the maximum length of a message to be equal to that necessary to accommodate a disc block (512 bytes) thus enabling file transfers on a page by page basis. The process structures we are using in our implementation are shown in figure 7.

![Diagram of process structures](image)

(a)  (b)

pipe:

system call:  

**Figure 7.** Process structures.

The datagram service is implemented as a Unix device driver process (a version of this software already exists, programmed by W. Sharpe of SRC Rutherford Laboratories, UK). The datagram process (i.e. the driver process) maintains a few buffers for messages not yet taken up by the receivers (it is practical to keep only a few buffers since they take up the limited Unix kernel space). We have tackled the problem of coping with limited buffer space as follows. A client expects at the most one
message and almost always will be in a position to receive it, so no message queues are likely for clients. The clock synchroniser process is also in a position to service the incoming messages since each message results in only a very small amount of work (possible clock update); so we do not expect any long message queues for this process. If the datagram process runs out of free buffers then it rejects incoming messages till one or more buffers become available (this behaviour is consistent with the already stated semantics of the datagram service). Only the server processes of a node pose any problem since they can receive arbitrary number of service requests; hence we need a mechanism to quickly remove messages from the datagram process. This has been achieved by a server maintaining a child process whose task is to execute repeatedly the `get work()` operation and buffer the request in the pipe[17] between the child and the server. Unix pipes as typically used can accommodate up to 4096 bytes of data using buffered disc storage where necessary. Thus the likelihood of messages being rejected by the datagram process due to lack of buffers has been made quite small. The price we have paid is the overall reduction in the speed of execution of a remote call, because it involves local inter-process communication activity via pipes, which in turn may occasionally involve disc accesses.

8. Conclusions

At a superficial level it would seem that to design a program that provides a remote procedure call abstraction would be a straightforward exercise. Surprisingly this is not so. We have found the problem of the design of the RPC to be rather intricate. To the best of our ability we have checked that all of the possible normal and abnormal situations properly map onto the responses of the "remote_call(...)", "get_work(...)" and "send_result(...)". Clearly, a formal validation exercise and experience with the completed implementation should expose any inadequacies in our design. We intend using the RPC mechanism for the construction of a reliable distributed filing system. Major parts of this system have been already designed and implemented[18]. Our next goal is the investigation of reliability problems at the application level as discussed in [19].

9. Acknowledgements

The work reported here was supported jointly by Science Research Council (UK) and Royal Signals and Radar Establishment (UK). Our understanding of the subject matter reported here has been improved as a result of discussions with our colleagues at Newcastle; in addition we have also benefited from informal contacts with other groups, most notably those at Cambridge, MIT and Xerox.
References


6. B. Lampson (XEROX), Private communication.

7. B. Nelson (XEROX), Private communication.

8. B. Liskov (MIT), Private communication.


