This approach to fault tolerance in digital systems is quite different form the usual methods. Normally, there is an error detection procedure which is monitored, and when detection occurs a set of recovery procedures are called. With the containment set approach there is neither a detection mechanism nor a recovery procedure. The fault tolerance results entirely from the structure of the system itself. It is assumed that the system will experience a period of errors when subjected to I/T faults; this is accepted, but the system is constructed so that it will return within finite time to its original function. There is no error masking—the tolerance is an inherent part of the design.

In general, because of system complexity, creating a single element containment set may be a task which is not possible or too difficult to achieve. However, there is a large class of digital systems for which this approach is indeed viable: namely, microprocessor controllers.

Removing all invalid elements from the containment set guarantees that eventually the system will return to the design function, but says nothing about how much time can elapse before task resumption. An analysis of specific systems can be done to minimize this recovery time and to obtain an upper limit for it. This can be done by examination of each system state which can cause a delay. Modifications of these states can lower the recovery times.

IV. Conclusions

The principle requirement for practical system application is finding a set of operational states which are complete, mutually exclusive, and finite, to allow use as a containment set. This allows digital system fault phenomena to be viewed through a high level upset perspective. This integrated approach to the fault/system/program complex can then yield provable, yet useful, design and validation techniques.

Upset theory has been applied to microprocessor controllers, and methods to obtain containment sets for 8085-based systems developed [9]. A testbed system was used to verify the practicality for the implementation of crash-proof controllers based on upset theory. This new method has been found to be far superior to the conventional watchdog timer method for applications where the need for fault tolerance is not critical enough to warrant an expensive approach such as TMR. The cost increase for fault tolerance is approximately the same as that of the watchdog timer method.

Microprocessor controllers are small enough that it is possible to reduce the containment set to valid states by removing all erroneous states. Larger systems most likely will not offer this luxury. The containment set would then categorize the various erroneous states which are not removable, and the fullest use of transition matrices would be made. The application of upset theory to the design of transient fault-tolerant microprocessor controllers is just one of its uses in the field of transient fault analysis.

References

[1] C. P. Disparte, "A self-checking VLSI microprocessor for electronic engine control," in *Proc. FTCS-11*, June 1981, p. 253.
[2] B. Courtois, "Some results about the efficiency of simple mechanisms for the detection of microcomputer malfunctions," in *Proc. FTCS-9*, June 1979, pp. 71–74.
[3] ——, "A methodology for on-line testing of microprocessors," in *Proc. FTCS-11*, June 1981, pp. 272–274.
[4] D. R. Ballard, "Designing fail-safe microprocessor systems," *Electronics*, vol. 52, pp. 139–143, Jan. 4, 1979.
[5] D. G. Platteter, "Transparent protection of untestable LSI microprocessors," in *Proc. FTCS-10*, 1980, pp. 345–347.
[6] J. F. Wakerly, "Microcomputer reliability improvement using triple-modular redundancy," *Proc. IEEE*, vol. 64, pp. 889–895, June 1976.
[7] A. Avizienis, "Architecture of fault-tolerant computing systems," in *Proc. FTCS-5*, June 1975, pp. 3–16.
[8] ——, "Fault-tolerance: The survival attribute of digital systems," *Proc. IEEE*, vol. 66, pp. 1109–1125, Oct. 1978.
[9] R. E. Glaser and G. M. Masson, "The containment set approach to crash-proof microprocessor controller design," in *Proc. FTCS-12*, June 1982.

The Design of a Reliable Remote Procedure Call Mechanism

S. K. Shrivastava and F. Panzieri

***Abstract*—In this correspondence 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.**

***Index Terms*—Atomic actions, data communication, distributed systems, fault tolerance, local area networks.**

I. Introduction

In this correspondence 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: 1) the selection of appropriate semantics and reliability features of the RPC mechanism, 2) the design of an appropriate message passing interface over which the RPC is to be implemented, and 3) the treatment of abnormal situations such as node crashes. These problems and their solutions are discussed in this correspondence. 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[1] (V7) operating system. However, most of the ideas presented in this correspondence are, we believe, sufficiently general to be applicable to any other local area network system.

Manuscript received October 2, 1981; revised January 7, 1982. This work was supported by the Science and Engineering Research Council of the United Kingdom and the Royal Signals and Radar Establishment of the United Kingdom.

The authors are with the Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, England.

[1] UNIX is a trademark of Bell Laboratories.

0018-9340/82/0700-0692$00.75

II. 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 correspondence 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 allows 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 mechanization of a suitable RPC facility and concentrate upon the reliability problems at the application program level. The most vexing problems 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]–[5]. Thus, a program terminates either producing the intended results or none at all. In a distributed system the implementation of atomic actions requires the provision of a special protocol, such as the *two phase commit* protocol [3], [4] to coordinate 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 Fig. 1 which shows a typical hierarchy of software interfaces. The point to note is that the atomic action software that supports $L3$ 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.

The algorithm below, which shows the bare essentials of an RPC mechanism, will be used to illustrate the reliability problems.

Client		Server
		cycle
send (· · ·);	→	*receive* (· · ·);
- - -		- -
		"work"
		- -
receive (· · ·);	←	*send* (· · ·);
		end

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 reissues the remote call:

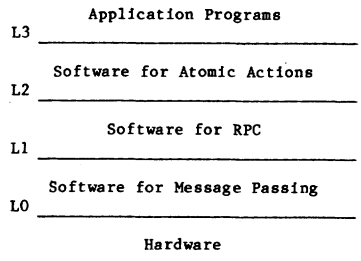

Fig. 1. Hierarchy of software interfaces.

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 Lampson and many ingenious schemes have been deviced for detecting and treating orphans [6], [7]. The above mentioned problems have led Nelson to classify the semantics of remote procedure calls as follows [7].

1) "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.

2) "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, 1) 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 relatively difficult to provide servers with arbitrary services (e.g., a server cannot easily 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 during a call. This capability is not enough to cope with orphans, however, 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 Fig. 1) by insisting that *all* programs that run over $L3$ be atomic actions with the "all or nothing" property. In particular, 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 Liskov's group at the Massachusetts Institute of Technology [8]).

To sum up this section: 1) we have chosen the exactly once semantics for our RPC, 2) the main reliability feature needed at the RPC level is that necessary to discard any unwanted messages, and 3) 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 reliability mechanisms application specific rather than general as argued in [9].

III. Communications Support for RPC: Datagram Versus Transport Service

The implementation of RPC requires that the underlying level support 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 acknowledgment. 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 initialization 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 acknowledgments; 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 acknowledgment). We may be able to utilize this bandwidth more effectively by reducing the need for connection management and acknowledgments 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.

IV. The Hardware and the Datagram Service

The Cambridge Ring hardware [12] 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.

1) *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.

2) *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 neither 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, clocks) 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 behavior of the hardware interface (i.e., a behavior that does not meet the specification) is likely to result in the same at the RPC/user interface to the extent that guaranteed behaviour cannot be promised.

The proposed datagram service will provide its users (processes) with the ability of: 1) sending a block of data to a named destination process, and 2) receiving a block of data from a specific or any process. We shall ignore here the fine details of how this may be implemented using the Ring operations described earlier; only the properties of the datagram service primitives will be described.

1) *send_ msg* (*destination:* ···; *message:* ···; **var** *status:* ···); where *status* = (*OK*, *absent*, *not-done*, *unable*).

The message is broken into packets and transmitted to the home station 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).

2) *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 performed 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].

V. 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 the section on reliability issues, 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 depicted below which should be self-explanatory. The following two assumptions will be made in the ensuing discussion: 1) 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; 2) node crashes amount to that station being not on line. We now consider the treatment of various responses obtained during message handling.

Client		Server
remote-call (···) *corresponds to the following code:*		**cycle**
- - -		- - -
send_ msg ( );	→	**repeat** *"get work"*
"send service request"		*receive_ msg* (*any*, ···);
		until *msg* = *valid;*
		- - -
		"perform work"
set(*time-out*);		- - -
repeat		*"send result"*
receive_ msg ( );	←	*send_ msg* ( );
until *msg* = *valid;*		- - -
- - -		**end;**

1) 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.

2) 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 forever; 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, or server's message not received because of a Ring fault.

3) 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.

4) 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 resent a few times before accepting defeat by signaling "unable" to the server. The reason for mapping all the three abnormal responses of send_ msg (···) onto 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: 1) remote_ call (···)—this is the client half of the program with the semantics discussed earlier; 2) get_ work (···)—this corresponds to the repeat loop code of the server; and 3) 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. This is not possible if the transport service is used for message passing.

VI. 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 (SN's) to messages. The problem of sequence numbering of messages is fairly complex if tolerance to node crashes is required. For example, 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; 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 SN's.

1) SN's 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].

2) SN's 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 1) above, in its requirements for crash proof storage.

3) SN's are unique over the entire system: if SN's 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: 1) the circulating token method of Le Lann [14], and 2) the loosely synchronized 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 behavior" (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 reason: 1) 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 2) 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.

SN = | time and data | station number |

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 below.

```
Client                                   Server
                                         begin
                                           llsn := get_SN (···);
  - - -                                  "llsn = last largest seq. no."
                                           cycle
i := get_SN (···);                           - - -
                                           repeat "get work"
  send_msg (···);              →             receive_msg (any, ···)
"message includes i"                       until msg · SN > llsn;
                                           llsn := msg · SN;
                                             - - -
                                             "perform work"
  set (time-out);                            - - -
repeat
  receive_msg (···);           ←           send_msg (···);
until msg · SN = i;                      "msg sent with SN = llsn"
  - - -                                      - - -
                                           end;
                                         end;
```

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 below).

```
Broadcaster                              Clock Synchronizer
cycle                                    cycle
  delay(t);                                receive_msg (···);
  i := get_SN(···);                        update_SN (msg · SN);
  for all remote nodes                   end;
  do
    send_msg (···);
  "send message with SN = i
   to a clock synchronizer"
end;
```

The "broadcaster" process of a node regularly (say once every few minutes) sends its sequence number to all of the remote clock-synchronizer 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-synchronizer in that cycle (at this level, a crashed node in no way affects the noncrashed 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 centralized 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-synchronizer" 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 initialized 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 summarizing the net effect of our sequence number assignment and generation scheme on the fault-tolerant behavior of our RPC mechanism: 1) since none of the state information of a call is maintained on a crash proof storage, a call does not survive a crash; 2) the clock management scheme ensures that any messages belonging to a "crashed call" are ignored.

VII. Concluding Remarks

The design presented here is currently being implemented on our UNIX systems. 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 $(\cdots)$," "get_ work $(\cdots)$," and "send_ result $(\cdots)$." Clearly, a formal validation exercise and experience with the completed implementation should expose any inadequacies in our design.

Note Added in Proof: The RPC is now operational; its implementation is described in a report available from the authors.

Acknowledgment

The authors' understanding of the subject matter reported here has been improved as a result of discussions with their colleagues at Newcastle; in addition, they have also benefited from informal contacts with other groups, most notably those at Cambridge, MIT, and Xerox.

References

[1] M. V. Wilkes and D. J. Wheeler, "The Cambridge communication ring," in *Proc. Local Area Network Symp.*, Boston, MA, Nat. Bureau of Standards, May 1979.

[2] H. C. Lauer and R. M. Needham, "On the duality of operating system structures," in *Proc. 2nd Int. Symp. on Operating Syst.*, IRIA, Oct. 1978; also in *Oper. Syst. Rev.*, vol. 13, pp. 3–19, Apr. 1979.

[3] B. Lampson and H. Sturgis, "Atomic transactions," in *Lecture Notes in Computer Science, Vol. 105.* New York: Springer-Verlag, 1981, pp. 246–265.

[4] J. N. Gray, "Notes on data base operating systems," in *Lecture Notes in Computer Science, Vol. 60.* New York: Springer-Verlag, 1978, pp. 398–481.

[5] S. K. Shrivastava, "Structuring distributed systems for recoverability and crash resistance," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 436–447, July 1981.

[6] B. Lampson, "Remote procedure calls," in *Lecture Notes in Computer Science, Vol. 105.* New York: Springer-Verlag, 1981, pp. 365–370.

[7] B. J. Nelson, "Remote procedure call," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, CMU-CS-81-119, 1981.

[8] B. Liskov, "On linguistic support for distributed programs," in *Proc. Symp. Reliable Distributed Software and Database Syst.*, Pittsburgh, PA, July 1981, pp. 53–60.

[9] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End to end argument in system design," in *Proc. 2nd Int. Conf. on Distributed Syst.*, Paris, France, Apr. 1981, pp. 509–512.

[10] R. W. Watson, "Hierarchy," in *Lecture Notes in Computer Science, Vol. 105.* New York: Springer-Verlag, 1981, pp. 109–118.

[11] C. A. Sunshine, and Y. K. Dalal, "Connection management in transport protocols," in *Computer Networks*, vol. 2. Amsterdam, The Netherlands: North-Holland, 1978, pp. 454–473.

[12] Science and Engineering Research Council (U.K.), "Cambridge data ring," Tech. Note, Sept. 1980.

[13] R. M. Needham, "System aspects of the Cambridge ring," in *Proc. 7th Oper. Syst. Symp.*, Dec. 1979, pp. 82–85.

[14] G. Le Lann, "Distributed systems: Towards a formal approach," in *Information Processing 77.* Amsterdam, The Netherlands: North-Holland, 1977, pp. 155–160.

[15] L. Lamport, "Time, clocks and the ordering of events in a distributed system," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 558–565, July 1978.

[16] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Commun. Ass. Comput. Mach.*, vol. 17, Oct. 1974.

A Statistical Failure/Load Relationship: Results of a Multicomputer Study

RAVISHANKAR K. IYER, STEVEN E. BUTNER, AND EDWARD J. McCLUSKEY

***Abstract*—In this correspondence we present a statistical model which relates mean computer failure rates to level of system activity. Our analysis reveals a strong statistical dependency of both hardware and software component failure rates on several common measures of utilization (specifically CPU utilization, I/O initiation, paging, and job-step initiation rates). We establish that this effect is not dominated by a specific component type, but exists across the board in the two systems studied. Our data covers three years of normal operation (including significant upgrades and reconfigurations) for two large Stanford University computer complexes. The complexes, which are composed of IBM mainframe equipment of differing models and vintage, run similar operating systems and provide the same interface and capability to their users. The empirical data comes from identically structured and maintained failure logs at the two sites along with IBM OS/VS2 operating system performance/load records.**

***Index Terms*—Computer failure data, performance-reliability models, statistical failure models.**

I. Introduction

The last decade has seen an explosion in the application of computers in many key commercial, military, and industrial environments. Our society has become increasingly reliant upon the error-free performance of such systems. As a consequence, enhanced reliability, high performance, and overall availability have all assumed increasing importance in computer design.

It is well known that as a system approaches high levels of utilization, degradation in performance occurs. A more interesting question, however, is whether an increased level of utilization results in a real or apparent degradation in system *reliability*, i.e., is there an inherent load versus failure relationship? This question has strategic significance, particularly since many security and defense systems are required to have maximum reliability at precisely the time of their peak load or stress. Significant as this question is, there are few studies available in this area [2], [4], [6]. It is a widespread belief that since these systems are electronic in nature no such effect exists on a significant scale.

Analytic failure modeling and simulation appear unattractive at this stage due to a lack of understanding of the processes involved. One alternative is statistical analysis. Accordingly, we embarked on

Manuscript received September 28, 1981; revised January 12, 1982. This work was supported in part by the Air Force Office of Scientific Research under Contract F49620-79-C-0069, the National Science Foundation under Grant MCS-7904864, and the CSIRO, Australia.

R. K. Iyer and E. J. McCluskey are with the Center for Reliable Computing, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305.

S. E. Butner was with the Center for Reliable Computing, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305. He is now with the Department of Electrical Engineering, University of California, Santa Barbara, CA 93106.

0018-9340/82/0700-0697$00.75