Rajdoot: A Remote Procedure Call Mechanism Supporting Orphan Detection and Killing

FABIO PANZIERI AND SANTOSH K. SHRIVASTAVA

Abstract—**Rajdoot is an RPC mechanism with a number of fault tolerance capabilities. The paper first discusses the reliability related issues and then describes how these issues have been dealt with in the RPC design. Rajdoot supports exactly once semantics with call nesting capability and incorporates effective measures for orphan detection and killing. Previously reported RPC mechanisms have not paid adequate attention to orphan treatment issues. Performance figures show that the reliability measures of Rajdoot impose little overhead.**

Index Terms—**Distributed systems, fault tolerance, interprocess communications, network protocols, remote procedure calls.**

I. Introduction

THIS paper describes Rajdoot, a remote procedure call (RPC) mechanism intended for distributed programming. (Rajdoot, derived from *Sanskrit*, means a royal messenger.) Rajdoot has been designed to provide a convenient set of primitives that can be used by arbitrary clients and servers. Language and programming environment specific issues to do with stub generation, binding, and naming, although important, have not been addressed here; rather, the paper concentrates on novel orphan handling aspects of Rajdoot. It is our claim that existing RPC mechanisms have paid little attention to the orphan problem and this represents their major shortcoming. The design of Rajdoot was completed during the Winter of 1984. A version running on PDP11's connected by a Cambridge Ring was soon completed. Since then it has been ported on 4.2BSD UNIX® systems connected by an Ethernet.

The paper is structured as follows. In the next section we discuss those RPC related reliability issues that we have regarded as important and then describe in Section III the design choices made for Rajdoot where we mention the specific reliability mechanisms of the RPC. The remaining sections describe the design and implementation of those mechanisms.

We conclude this introduction by summarizing the main features of our RPC. Rajdoot supports: 1) *exactly once* semantics; 2) arbitrary nesting of RPC's; 3) client timeouts and repeated retries of the call; and 4) orphan detection and killing. As we shall see, Rajdoot differs from other RPC mechanisms reported in the literature mainly because of the manner in which a number of fault tolerance features (e.g., orphan killing) have been integrated into it.

Manuscript received July 31, 1985; revised April 30, 1986. This work was supported in part by the Royal Signals and Radar Establishment of the UK Ministry of Defence and the UK Science and Engineering Research Council.

F. Panzieri is with the Department of Computer Science, University of Pisa, 56100 Pisa, Italy.

S. K. Shrivastava is with the Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne NE1 7RU, England.

IEEE Log Number 8718294.

®UNIX is a registered trademark of AT&T Bell Laboratories.

II. Reliability Related Design Issues

Failures in a distributed system, such as lost messages and node crashes, can create reliability problems not normally encountered in a centralized (one node) system. Thus, treatment of failures is one of the main issues that requires close attention in an RPC design. In this section we discuss the reliability issues, pointing out the problems posed by orphans.

A. Fault Models, RPC Semantics, and a Correctness Criterion

We will model a distributed system as a collection of nodes connected by a communication subsystem. Faults in the communication subsystem are responsible for the following types of failures: 1) a message transmitted from a node does not reach its intended destination (termed a *communication failure*); 2) messages are not received in the same order as they were sent; 3) a message gets corrupted during its transmission; and 4) a message gets replicated during its transmission.

There are well known mechanisms (based on checksums and sequence numbers) that enable a receiver to treat messages that arrive out of order, corrupted or are copies of previously received messages, so we need only concern ourselves with the treatment of communication failures. The fault model for *node failures* is as follows: either a node works according to its specifications or that node stops working (*crashes*). After a crash, a node is repaired within a finite amount of time and made active again. Most published works on RPC's have implicitly assumed the fault models we have described here explicitly (e.g., [1], [2]).

Given that we wish to design an RPC mechanism for a system prone to the faults just described, we can envisage a range of fault tolerance measures. The following is one such classification (which indicates RPC's with increasing degrees of fault tolerances). We will assume that the reception of a reply message from the called server constitutes a *normal termination* of a call. Then the classification given below indicates conditions under which a normal termination is possible.

1) No communication and/or node failures occur during the call. 2) The RPC mechanism copes with a fixed finite number of communication failures. 3) The RPC mechanism copes with a fixed finite number of communication failures and server node crashes (server crashes, for short). 4) Same as 3), but in addition, tolerance to fixed finite number of client node crashes (client crashes, for short) is also present.

Next, we present a classification for the semantics of remote calls [1], [3]:

1) At least once semantics: a normal termination implies one or more executions at the called server.

2) Exactly once semantics: a normal termination implies exactly one execution at the called server.

Both of the above semantics say nothing about what happens if a call does not terminate normally and it is assumed that zero, partial, one, or more executions [for type 1)] or zero, partial or one execution [for type 2)] are a possibility. A "stronger" semantics is specified by the third type given below [4]:

3) At most once semantics: same as exactly once, but in addition, calls that do not terminate normally do not produce any side effects.

Choosing appropriate fault tolerance capabilities and semantics is indeed one of the most important decisions to be taken in an RPC design. We next present a simple and intuitively appealing *correctness criterion* for an RPC implementation.

Let C_i denote a call made by a client and W_i represent the corresponding computation invoked at the called server. Let C_i and C_j be any two calls made by a client such that: 1) C_j happens *after* C_i (denoted by C_i *then* C_j); and 2) computations W_i and W_j share some data such that W_i and/or W_j modify the shared data. Then we say that an RPC implementation must meet the following *correctness criterion*, in the presence of specified types of failures [5]:

$$CR: \quad C_i \text{ then } C_j \text{ implies } W_i \text{ then } W_j$$

The criterion *CR* states that a sequence of calls at a client should give rise to computations invoked in the same sequence (obviously, if W_i and W_j are disjoint—do not share any data—then strictly speaking, no ordering is necessary). In the absence of any failures, the synchronous nature of calls guarantees that *CR* will be satisfied. However, failures can create orphans (see the next subsection) that do require special measures in order to meet *CR*. Note that the correctness criterion must be met irrespective of the RPC semantics chosen.

In any large distributed system, communication and node failures can be relatively frequently occurring events, so any well engineered RPC mechanism must strive to meet *CR*. In this respect, most existing RPC mechanisms (e.g., [2], [11], [12]) are inadequate. In contrast, Rajdoot meets *CR* in a very efficient manner.

B. Orphans

Orphans are unwanted executions that can occur due to communication and node failures. We will say that a call terminates *abnormally* if the termination occurs because no reply message is received from the called server. Network protocols typically employ *timeouts* to prevent a process waiting for a message from being held up indefinitely. Assume that a client process waiting for results from the called server has a timer set (or equivalently, some other protocol dependent mechanism that signals the client if no reply is received after some duration). If the call terminates abnormally (the timeout expires) then there are four mutually exclusive possibilities to consider: 1) the server did not receive the call message; 2) the reply message did not reach the client; 3) the server crashed during the call execution and either has remained crashed or is not resuming the execution after crash recovery; and 4) the server is still executing the call, in which case the execution could interfere with subsequent activities of the client, as depicted in Fig. 1.

Client K at node A issues a call to server X at node B which executes the requested work ("work 1" in Fig. 1), and the call terminates abnormally before X completes the work. The client then issues another call to some server Y at node B ("work 2" in Fig. 1). If the computation by X is still in progress, and "work 1" and "work 2" have data in common, then these computations can interfere with each other. Note that the concurrency depicted in Fig. 1 must be regarded as undesirable, since it is expected that the execution of a sequential program should give rise to a sequential computation characterized by a single flow of control. Concurrency control techniques (e.g., locking) are normally intended to prevent interferences between different programs under the assumption that each program will invoke a sequential computation.

The interference depicted in Fig. 1 might also occur in the case of a crash of the client node A. If the client resumes execution after recovery by reissuing the call, or by making a new call to the same node, then we have a similar situation as before. We will refer to unwanted computations (e.g., "work 1") as *orphans*. As a further example, consider the case where a server's work is some arbitrary computation, including calls to other servers, such that a crash of a server can leave orphans on other nodes. The scenario depicted in Fig. 2 is thus possible. Note that this type of interference in a nested call can also occur in the absence of a server crash, as illustrated in Fig. 3.

It is needless to say that the examples given here do not constitute an exhaustive list of possible interferences. They are intended to show that there are a variety of ways interferences can occur (and not just because of crashes as is often assumed).

How should orphans be treated? The correctness criterion *CR* stated before essentially states that executions such as work 1, work 2, Fig. 1, should be serialized to take place in the same order as the invoking calls. One way of meeting this requirement dynamically is to make sure that a server receiving a call request obeys the following two rules [5]: 1) a call request belonging to a 'past' call is not accepted for execution; and 2) once a call re-

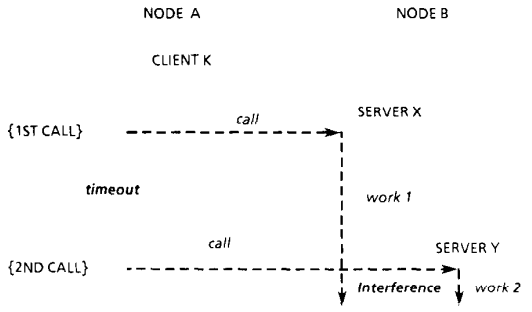


Fig. 1. Example of interference caused by a timeout.

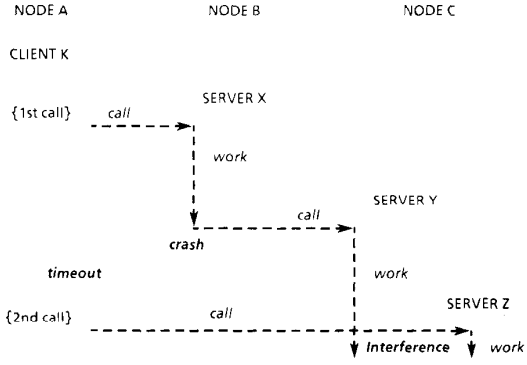


Fig. 2. Possible interference in a nested call (crash case).

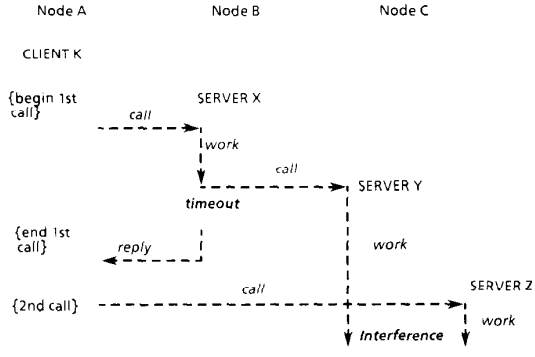


Fig. 3. Possible interference in a nested call (no crash case).

quest is accepted, the corresponding execution is started only after any ongoing executions belonging to "past" calls on the node have been aborted. It is easy to see that this will ensure that orphan executions will neither interfere nor overtake the "current" execution.

A number of orphan detection and abortion techniques have been discussed in [3], [5]. They tend to be expensive and difficult to implement (since these techniques themselves must be robust against failures). As we shall see, Rajdoot has three mechanisms built in to cope with orphans—it is by breaking the orphan problem into three subproblems that Rajdoot achieves its efficiency. We note that the semantics of the RPC can also impose some additional requirements not captured by *CR*. In particular, for *at most once* calls, not only should orphans be aborted as stated above, the abortion must also include undoing of any side-effects that may have been produced.

Note that some RPC mechanisms (e.g., [2]) do not employ timeouts to prevent indefinite waiting by a client, but make use of special *probe messages* to detect if the called server is still running. So, a call is terminated abnormally only when a crash of the called node is suspected. We show here that this mechanism is not enough to prevent interferences. Consider the situation depicted in Fig. 2—which could still occur if the probe mechanism is used for terminating the first call. The situation depicted in Fig. 1 is also possible if a communication failure lasting a sufficiently long time occurs which causes the first call to be terminated abnormally. We thus see that orphan creations due to abnormally terminated calls is a fundamental problem.

Finally we would like to remark that, in addition to causing consistency problems, orphans also consume (possibly) scarce resources such as message buffers, so their speedy abortion is desirable anyway.

III. Choosing Rajdoot's Fault Tolerance Capabilities

This section describes the design choices made in our RPC. We wanted the RPC mechanism to be general purpose, rather than for simply invoking idempotent operations, which limits the choice to either *at most once* or *excatly once*. Out of the two, we have opted for *exactly once* for the following reason. At most once calls require sophisticated backward error recovery support of atomic transactions [4], [6]. We wanted our RPC mechanism to be sufficiently "neutral" to support applications that do and do not make use of atomic transactions, which suggests that *exactly once* semantics is more appropriate.

Out of the four options for normal termination presented in the previous section, the fourth one—permitting a call to terminate normally in the presence of both client and server crashes—was discounted straightaway on the grounds that it provides too much functionality and is far too complex to implement. It is better for a client to fix its own crash resistance strategy, rather than to fix it at the RPC level. The choice is then between 2) and 3), and we have opted for 2). That is, a call can complete normally in the presence of a fixed finite number of communication failures, but not if the server node crashes (in which case the call is guaranteed to terminate abnormally). Allowing a call to terminate normally in the presence of server crashes would have required backward error recovery facilities—which in our view is better employed at higher levels than at the level of RPC.

The correctness criterion *CR* is met by Rajdoot by employing three orphan handling mechanisms:

M_1: If a client call terminates abnormally, then it is guaranteed that any computations the call may have generated have also terminated.

M_2: Consider a node that crashes and after recovery makes a remote call to some node *C*. Then, if *C* has any

orphans because of the caller's crash, then they will be aborted before the execution of the call starts at C.

M_3: What if the node remains crashed or after recovery never makes calls to C? In this case it is guaranteed that any orphans on C will nevertheless be detected and killed within a finite amount of time.

These mechanisms are discussed at length in the rest of the paper, but we give some hints here to present their essential mode of operation.

M_1: Every call contains a *deadline*, indicating to the server the maximum time available for execution. If the deadline expires, then the server aborts the execution and the call terminates abnormally. We thus see that if there are no node crashes in the system, then M_1 will be enough to cope with orphans. The remaining two mechanisms cope with crashes.

M_2: Every node maintains a stable (crash proof) counter—called a *crashcount*—that is incremented immediately after a node recovers from a crash. A node also maintains a table of crashcount values for clients that have made calls to it. A call request contains the client's crashcount value—if this value is greater than the one in the table at the called server, then there could be orphans at the server which are first aborted before proceeding with the call.

M_3: Every node has a *terminator* process that occasionally checks the crashcount values of other nodes—by sending messages to them and receiving replies—and aborts any orphans when it detects any crashes.

Mechanisms M_1 and M_2 can be designed to provide a remarkably powerful method of orphan handling with hardly any performance overheads in that no extra messages are needed for orphan detection and killing. Mechanism M_3 does impose some overheads but as it turns out, they need not be excessive since a terminator need only perform its checks once every few minutes.

For the sake of completeness we present here the specification of the RPC primitive available to a client (where parameters and results are passed by values):

```
rpc(server: . . . ; call: . . . ; timeout: . . . ; retry: . . . ;
  var reply: . . . ; var rpc__status: . . .);
```

The rpc__status variable can assume one of the following values:

```
rpc__status = (OK, NOTDONE, UNABLE);
```

The second parameter contains the name together with the relevant parameters of the operation to be performed by the server whose address is in the first parameter. The *retry* parameter indicates the number of times the call is to be retried (default value being zero). Let, for some call, n be the value of the retry parameter and t be the timeout value. Then, if after issuing the call, no reply is received within duration t, the call will be reissued; this process is repeated a maximum of n times. So, the worst case normal completion time for a call will be at most $(n + 1)^*$ t units of time. The semantics of the call under status OK, NOTDONE, and UNABLE is given below:

rpc__status = OK: The specified call has been executed once by the server; the result is available in "reply." This represents a normal termination of a call, with the call taking at most $(n + 1)^*$ t units of time to complete.

rpc__status = NOTDONE: The call has not been executed. This response is obtained when some communication failure prevents the call message from being transmitted to the server; the response is obtained in less than t units of time.

rpc__status = UNABLE: At most one execution may have taken place at the called server; "reply" does not contain any results. This case represents an abnormal termination, with the call taking at most $(n + 1)^*$ t units of time to complete. It is guaranteed that any computation the call may have generated has also terminated.

So, referring back to the previous examples of interferences, when the first call from the client K terminates (either abnormally as in Figs. 1 and 2, or normally as in Fig. 3) there will be no ongoing computations for that client at nodes B and C.

We claim that Rajdoot closely approximates the behavior of local calls. For a program, when a local call terminates either normally or abnormally (an exceptional return is obtained), we do not expect any ongoing activities at the called procedure. The same behavior is modeled by our RPC. In a single node system, a crash destroys all the ongoing computations. This behavior is approximated by Rajdoot as follows: a crash of a node does not "instantly" stop all the remote calls initiated from the node, rather, when post crash calls are made, any orphans on the called node are first aborted.

IV. Reliability Mechanisms

This section contains the details of the three reliability mechanisms mentioned before. In order to present these details, it is important to know the execution model employed by the RPC mechanism. This model, together with some protocol related details are presented in the next subsection.

A. The Execution Model and the RPC Protocol

The execution model adopted has been influenced by the use of RPC's in the Newcastle distributed system [7], and is described here with the help of a diagram (see Fig. 4). Each node runs a *manager* process that operates at some well known address. The primary task of a manager is that of creating server processes that execute clients' remote calls; in particular, each server executes only the calls of the client for which it was created. At the same time, once a server has been created at a node, a client directs all its remote calls intended for that node to the created server. Servers themselves may invoke remote calls as part of their "work," thus giving rise to nested calls.

The process of creating a server is handled transparently to the client program by the RPC mechanism. In essence, the first remote call issued by a client to a node

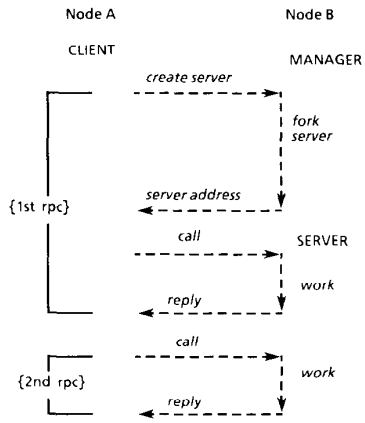


Fig. 4. RPC execution model.

is converted by the RPC mechanism into a request for the manager at that node to create a server, and is transmitted to that manager (the *create server* message of Fig. 4). The manager spawns a server and goes back to receive requests from the network. The spawned server acquires a (logical) address, which remains private to that server until it terminates, and replies to the client by sending it this address (the *server address* message of Fig. 4). This message contains the same sequence number as the corresponding "create server" request, so the client is in a position to accept the right message. Lack of "server address" message within a predefined timeout period causes a retransmission of the "create server" request. The RPC mechanism incorporates measures for dealing with the possibility of multiple servers being created for the same client by possible retransmissions of this request (see the next subsection).

Once a server has been activated and its address received by the client, the client's call is transmitted with a new sequence number to that server. Any exceptions during transmission of the "call" message are dealt with by retransmitting that message with the same sequence number. The server receives the call, discarding further calls with the same sequence number, performs the work and sends the result as a "reply" message containing the same sequence number as the corresponding "call" message. If the client does not receive the reply within the specified timeout period, and "retry" value is nonzero, the call message is sent again (with the same sequence number as before), and "retry" value is decremented by one. A server always maintains the results of the most recently executed call, so that it can effectively cope with retry requests arising out of lost replies. The manager process of a node has been designed to be "stateless": after servicing a create server request, the manager simply "forgets" about this request. This greatly simplifies its design and implementation.

B. Reliability Mechanisms

We employ local (stable, crash proof) clocks for obtaining monotonically increasing sequence numbers for messages. A node crash will destroy all the servers on that node which are not recreated, thus ensuring that respective client calls will terminate abnormally. A server maintains the sequence number of the most recently executed call and will only accept new requests with higher numbers. This is a well known method of ensuring that delayed messages, representing "past" calls do not cause any executions.

It was stated earlier that the manager process of a node does not maintain any state. This means that client retries for creating a server can result in more than one server being created. A newly created server starts an idle timeout and waits for a call request; the duration of the timeout is set to slightly more than two message round trip delays. If the timeout expires, then it can only mean that 1) the client has crashed or it cannot send a request due to some communication failure; or 2) the server has been created spuriously. In either case, the server aborts itself. This simple technique ensures that only the right number of servers survive. Once a server gets a call request, it "knows" that it is not unwanted, so it will not unilaterally destroy itself (if this server becomes an orphan due to a client crash, it will be destroyed by a different mechanism which will be discusssed shortly).

A call request to a server contains the *deadline* $(n + 1)^* t$, representing the maximum time available for executing that call. The server receiving a call starts a timer whose value is based on the deadline; if the deadline expires—the server is still executing the call—then the execution is aborted, with the server initialized to receive new calls. At about the same time the client's timeout will expire, causing the call to terminate abnormally. Let d be the maximum transmission delay for a message, and D be the deadline for a call; then the computation time T available at the called server is: $T \leq D - 2^* d$. If the server makes remote calls, then the deadlines for these calls must be calculated properly. For example, if a server is making just one remote call, the deadline $D1$ should be: $D1 \leq T - t1$, where $t1$ is the local computation time. The deadline mechanism provides a simple means of guaranteeing that an abnormally terminated call does not have ongoing computations at remote nodes. The price paid for this simplicity is the requirement for clients to estimate computation times; however, the retry parameter of a call does provide some flexibility in this direction (we will return to this subject in a subsequent section).

We will now discuss how orphans due to node crashes are detected and aborted. The basic idea as mentioned earlier is quite straightforward. Every node maintains a variable called a *crashcount* which is in fact the local (stable) clock value at the time the node was rebooted after a crash. A node also maintains crashcount values of client nodes who have made calls to it. These values are maintained in a table referred to as a C-LIST. A newly created server checks the client supplied crashcount value against the corresponding value in the C-LIST; if the former is greater, then this indicates that the caller has had a crash, in which case there could be orphans on the node. So, the

server aborts all other servers created by clients of the calling node before executing the call. If the two values are the same, then there cannot be any orphans of the calling node. Finally, if the C-LIST does not contain a crashcount entry for the caller, then an entry is made with the client supplied value.

The deadline mechanism plus crashcount based orphan detection and killing technique provides a powerful means of preventing interferences with remarkably little overheads. Given the provision of stable clocks at each node, no stable storage facility is required, neither is there any need for keeping clocks synchronized. This completes the discussion on mechanisms M_1 and M_2. A description of the mechanism M_3 follows.

After a server finishes servicing a call, it waits for the next call to come. This waiting is performed with an idle timeout (which is typically a few minutes). If the timeout expires, there could be any of the following situations possible at the client: 1) the client has crashed and not yet recovered; 2) the client has crashed, but after recovery no calls to the node have been made; 3) the client program has terminated, without informing the server; and 4) communication between the nodes is no longer possible. Out of these four possibilities, we have chosen not to deal with possibility 3), believing it to be the responsibility of clients to terminate servers (for which a *terminate* primitive has been made available to clients), and possibility 4) is treated as a client crash. After the expiration of the timeout, a server marks itself as a *potential orphan* and resumes waiting for a call. If a call is subsequently received, the server unmarks itself before executing the call.

Every node has a *terminator* process that regularly (every few minutes) constructs a list of potential orphans on its node and calls relevant client nodes to see if they are running. These messages are directed to the managers of these nodes. This then is the second function of a manager. Upon receiving such a request, the manager simply sends the current crashcount value in the reply. Since this is a read only operation, message retries at either end do not pose any problems. If a terminator does not get a reply within a reasonable amount of time (after a few retries), it is taken that the called node has crashed, in which case the relevant potential orphans are aborted. The same action is performed if the crashcount value in a reply is larger than the one in the C-LIST. The terminator based mechanism certainly imposes some overheads, but these are not deemed excessive. This is because a terminator need only activate itself infrequently and it does not generate excessive message traffic.

V. Implementation Notes

A working implementation of Rajdoot was first performed over a few PDP11's with UNIX connected by a Cambridge Ring. The implementation was later ported on to 4.2BSD UNIX systems running over Vax's, SUN's and Whitechapels. This section reports on some implementation details; a full report is available in [16].

The C-LIST of a node is an array of elements of the following type:

```
type clistelement = record
    clientnode: . . . ; "client node address"
    crashcount: . . . ; "clock value at client node"
    serverlist: array [. . .] of serverid
end
```

The "clientnode" field contains the address of the client node whose calls have been serviced by servers whose identifiers are recorded in the "serverlist." The "crashcount" field contains the last known crashcount of the client node. The "serverid" variable contains two fields: a process identifier field and a Boolean flag "PO-FLAG" indicating whether the server is a potential orphan. The C-LIST of a node is initialized to empty at the node startup time and is shared between clients, servers, and the terminator of that node.

When a client calls a remote manager for creating a server, it supplies its node address and crashcount value; this pair of values is passed on to the created server. If the caller is itself a server, then all the preceding client's pairs are also supplied, as illustrated below (the term *rpc-path* will be used to refer to the set of such pairs). In Fig. 5, server X will be supplied with the rpc-path $< A, i >$; server X itself creates a server Y at node C, so the rpc-path in the "create server" request sent to the manager at C will be $< A, i >, < B, j >$.

A newly created server checks if the crashcount of clients in the rpc-path match with corresponding entries (if any) in the C-LIST. If for some client this is not the case, then all the servers recorded in the "serverlist" entry for the client node in the C-LIST must be orphans. The server aborts these servers, clears their names from the C-LIST and inserts the new crashcount value and its own identity, with PO-FLAG set to false. Finally, if there is an entry for a given pair in the rpc-path, and crashcount values are same, then the server only logs itself in the serverlist. It should be noted that these operations are performed only at a server creation time and not for each and every call; further, the rpc-path is supplied by a client only when creating a server. Whenever a server becomes a potential orphan, it changes its flag in the C-LIST. Fig. 6 shows a tabular representation of the C-LIST for node C, Fig. 5.

A client who is also a server makes use of the C-LIST as follows. Suppose server Y, Fig. 5, wants to create a server at some node D. In this case it will scan the C-LIST to retrieve pairs $< A, i >, < B, j >$ and then send a request with rpc-path $< A, i >, < B, j >, < C, k >$.

Finally, the terminator of a node makes use of the C-LIST to construct the list of clients to whom enquiry messages are to be sent. If for example, Y at node C is marked as a potential orphan, then the terminator at C will send enquiries to nodes A and B.

Performance measures of the RPC with and without reliability mechanisms indicated that these mechanisms were responsible for adding at most 30 percent overheads

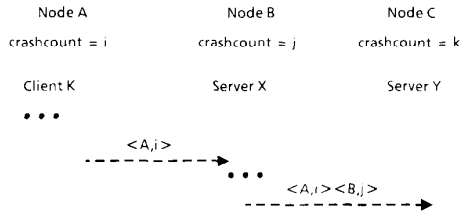


Fig. 5. RPC path in a nested call.

clientnode	crashcount	serverlist
A	i	Y.....
B	j	Y.....

Fig. 6. Tabular representation of the C-LIST at node *C*.

on null calls. Since the overheads are independent of RPC execution times, the percentage overhead figure reduces for non-null calls, indicating that Rajdoot incorporates effective reliability measures quite cheaply.

VI. A Critical Assessment of the Reliability Mechanisms

In this section we will attempt to justify our selection of Rajdoot's reliability mechanisms in preference to other mechanisms.

Certainly, the most controversial choice seems to be the employment of the deadline mechanism for aborting server executions, the main objection being *how can clients estimate the required deadlines?* Putting aside this objection for the time being, it should be clear that the deadline mechanism does represent an extremely cheap and effective means of preventing orphans from causing interferences. Consider for a moment a possible alternative—that of a server dynamically checking that there are no orphans present before commencing an execution. This would mean maintaining a shared table at a node that is accessed by all the servers of that node for each and every call request. This is likely to become a performance bottleneck. Any other solution such as requiring clients to send explicit abort messages are even more expensive and not particularly reliable. For these reasons we came to the conclusion that the deadline mechanism represents a reasonable compromise.

Returning to the problem of estimating deadlines, we offer a possible solution that can be incorporated in a stub generator. It is common to require that remote objects register their interface definitions with a name server so that clients can import these definitions during bind time [2]. If the registration information also contains an execution time estimate, then a stub generator can automatically calculate the call deadline, assuming message transmission times can be estimated. Note that the retry mechanism provides a convenient means of coping with variations in execution times (say due to overload situations).

We now examine the remaining two orphan killing mechanisms of Rajdoot. The crashcount based mechanism (M_2) has several advantages: 1) most modern computers contain a stable clock anyway; 2) the C-LIST does not require stable storage; 3) the C-LIST, which is shared between all the processes of that node, is only accessed by a server the very first time it is created—so it is not a performance bottleneck; and most importantly 4) no special crash recovery procedures are required for nodes. Contrast this with a widely known orphan killing technique based on maintaining a "hit list" on stable storage at each node (the list contains names of nodes to which calls have been made). Crash recovery of a node then involves sending abort messages to the nodes named in the hit list. This turns out to be an expensive and difficult technique to implement—particularly if several nodes are performing crash recoveries simultaneously. Lastly, the terminator based mechanism (M_3) of Rajdoot is also cheap since not only is it activated infrequently, but also it does not require stable storage.

Finally we note that Rajdoot mechanisms "scale-up" nicely—despite the fact that our implementation has been performed and tested over a system containing only a few nodes, there is every reason to believe that our design will be equally effective for systems containing thousands of nodes.

VII. Concluding Remarks

We commenced the design exercise in the Winter of 1983; the original aim was to build an orphan killing system for the existing RPC [9], [10]. Several schemes were designed but were abandoned as being too expensive for large systems and it soon became clear that a complete redesign of the RPC was required.

In our design we have tried to minimize the amount of state information processes have to maintain for proper execution of remote calls. Thus, the manager process of a node maintains no state information and a server only maintains the last sequence number value and the result of the last call for its client. The only shared data in a node is the C-LIST and the variable crashcount and this data is accessed infrequently. It is our view that an RPC mechanism that provides *exactly once* semantics but fails to provide any guarantee of freedom from interferences from computations of preceding calls is not really adequate. Also, orphans generated due to node crashes must be killed, if only to release scarce resources (such as network ports and buffers). Thus, some provision for orphan detection and killing must be available.

The execution model of Rajdoot has been optimized for multiprogramming systems where processes are not cheap. Referring to Fig. 4, it was stated that a client has a single server at a remote node to service all of its calls. This is purely an efficiency measure; logically there is no reason as to why a client cannot have more than a single server at a node—the orphan detection and killing measure will still work as expected. Taken to its extreme, one could envision a system where each call is executed by a

new process (an attractive possibility in operating systems offering cheap "lightweight" processes [2]). In such a system, mechanism M_3 of Rajdoot will not be necessary if after executing a call the process either dies or gets reallocated to serve some other call.

The first detailed study of RPC's appeared in the Ph.D. dissertation of Nelson [3], where among other things, a variety of orphan killing techniques were presented but not implemented. The subsequent Cedar implementation [2] also has not addressed the issue of orphan treatment. Cedar RPC supports *exactly once* semantics, and like Rajdoot, does not permit a call to terminate normally in the presence of server crashes. However, an abnormally terminated call does not guarantee that the computation invoked (if at all) at the callee has terminated—so no guarantee of freedom from interference for subsequent calls can be given. The same is true when a crashed node, after recovery makes remote calls. Of a few commercially available RPC's [11], [12], the SUN RPC does not specify the call semantics to be supported and has no provision for orphan treatment. Similarly, Courier RPC appears to support *exactly once* semantics, but its description is not precise about its fault tolerance capabilities and no support for orphan treatment is provided. Reference [13] reports on an interesting atomic RPC supporting what we have classified here as *at most once* semantics: as expected, the mechanism has built in facilities for concurrency control and backward error recovery.

Rajdoot is currently forming the basis for several research projects under way at Newcastle. First of all, it is being extended to include *multicasting* facilities enabling calls to groups of servers to be made. An object-based system supporting fault tolerant objects and atomic actions as discussed elsewhere [14] is being designed and constructed on top of Rajdoot. In a different experiment, Rajdoot is being integrated into a capability-based system where interesting extensions to the RPC mechanism are planned to include cheap but robust garbage collection mechanisms.

Acknowledgment

Acknowledgment is due to L. Marshall, B. Randell, and R. Stroud for their constructive criticisms. F. Hedayati was responsible for porting Rajdoot on 4.2 BSD UNIX systems.

References

[1] A. Z. Spector, "Performing remote operations efficiently on a local computer network," *Commun. ACM*, vol. 25, no. 4, pp. 246-260, Apr. 1982.

[2] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39-59, Feb. 1984.

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

[4] B. Liskov and R. Scheifler, "Guardians and actions: Linguistic support for distributed programs," *ACM Trans. Program. Lang. Syst.*, vol. 5, pp. 381-404, July 1983.

[5] S. K. Shrivastava, "On the treatment of orphans in a distributed system," in *Proc. 3rd Symp. Reliability in Distributed Software and Database Systems*, IEEE Comput. Soc., Florida, Oct. 1983, pp. 155-162.

[6] L. Svobodova, "Resilient distributed computing," *IEEE Trans. Software Eng.*, vol. SE-10, no. 3, pp. 257-268, May 1984.

[7] D. R. Brownbridge, L. F. Marshall, and B. Randell, "The Newcastle connection or Unixes of the world unite!" *Software: Practice and Experience*, vol. 12, pp. 1147-1162, 1982.

[8] F. Panzieri, "Design and development of communication protocols for local area networks," Comput. Lab., Univ. Newcastle upon Tyne, Tech. Rep. 197, Mar. 1985.

[9] S. K. Shrivastava and F. Panzieri, "The design of a reliable remote procedure call mechanism," *IEEE Trans. Comput.*, vol. C-31, no. 7, pp. 692-697, July 1982.

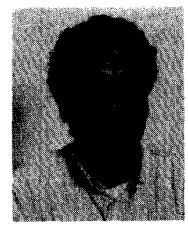
[10] F. Panzieri and S. K. Shrivastava, "Reliable remote calls for distributed Unix: An implementation study," in *Proc. 2nd Symp. Reliability in Distributed Software and Database Systems*, IEEE Computer Soc., Pittsburgh, PA, July 1982, pp. 127-133.

[11] "Remote procedure call specification," Sun Microsystems, Mountain View, CA, Jan. 1985.

[12] "Courier: The remote procedure call protocol," Xerox System Integration Standard XSIS 038112, Stamford, CT, Dec. 1981.

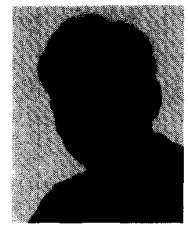
[13] K. J. Lin and J. D. Gannon, "Atomic remote procedure call," *IEEE Trans. Software Eng.*, vol. SE-11, no. 10, pp. 1126-1135, Oct. 1985.

[14] S. K. Shrivastava, "Robust distributed programs," in *Resilient Computing Systems*, T. Anderson, Ed. London: Collins, 1985, pp. 102-121.



Fabio Panzieri received the "Laurea" degree in computer science from the University of Pisa, Pisa, Italy, in 1978, and the Ph.D. degree in computer science from the University of Newcastle upon Tyne, Newcastle upon Tyne, England, in 1985.

From 1979 to 1985 he was a Research Associate at the Computing Laboratory of the University of Newcastle upon Tyne. Currently, he is an independent consultant and his activity is based in Pisa. His research interests include distributed systems architectures, primitives for distributed computing, and design of communication protocols for local and wide area networks.



Santosh K. Shrivastava received the B.E. and M.E. degrees in electronic engineering from the University of Poona, India, in 1965 and 1967, respectively, and the Ph.D. degree in computer science from the University of Cambridge, Cambridge, England, in 1975.

After working for several years in industry, he joined the computing laboratory of the University of Newcastle upon Tyne in 1975, where he is now a Professor of Computing Science. His areas of interest include reliable distributed computing, protocols, and system specifications.

Dr. Shrivastava is a member of the British Computer Society and the IEEE Computer Society.