Abstract:

Rajdoot is an RPC mechanism with a number of fault tolerance capabilities. The paper first discusses the reliability and performance 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 overheads.

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

By

F. Panzieri and S.K. Shrivastava

TECHNICAL REPORT SERIES

Editor: Mr. M.J. Elphick

Number 200

May, 1985

© 1985 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England.
bibliographical details

PANZIERI, Fabio


Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1985.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 200.)

Added entries

SHRIVASTAVA, Santosh Kumar.
UNIVERSITY OF NEWCASTLE UPON TYNE.

Suggested classmarks (primary classmark underlined)

Dewey (18th):
001.64404
U.D.C.
519.687

Suggested keywords

DISTRIBUTED SYSTEMS
FAULT TOLERANCE
INTER-PROCESS COMMUNICATIONS
NETWORK PROTOCOLS
REMOTE PROCEDURE CALLS

Abstract

Rajdoot is an RPC mechanism with a number of fault tolerance capabilities. The paper first discusses the reliability and performance 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 overheads.

About the author

Dr. F. Panzieri was with the Computing Laboratory from 1979 to 1984 as a Research Associate.
Dr. S.K. Shrivastava joined the Computing Laboratory in August 1975 where he is a lecturer.
Rajdoot: A Remote Procedure Call Mechanism
Supporting Orphan Detection and Killing

ABSTRACT

Rajdoot is an RPC mechanism with a number of fault
tolerance capabilities. The paper first discusses the reli-
ability and performance 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 overheads.

1. Introduction

This paper describes Rajdoot, an 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 atomic actions
(transactions), type checking, type conversion, binding and naming,
although important, have not been addressed here. It is our belief that
such issues should be dealt with as much as possible at other levels.

The paper is structured as follows. In the next section we discuss
those RPC related design issues that we have regarded as important and
then describe in section 3 the design of Rajdoot where we discuss how
these issues have been addressed; in addition we also describe how our design contrasts with other RPC mechanisms reported in the literature. The last section contains conclusions from this work.

We conclude this introduction by summarizing the main features of our RPC. Rajdoot supports: (i) 'exactly once' semantics; (ii) arbitrary nesting of RPCs; (iii) client timeouts and repeated retries of the call; and (iv) 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.

2. RPC 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 both the reliability issues and performance related issues for RPCs.

2.1. Reliability Issues

We will model a distributed system as a collection of nodes connected by a communication sub-system. Faults in the communication sub-system are responsible for the following types of failures: (i) a message transmitted from a node does not reach its intended destination; (ii) messages are not received in the same order as they were sent; (iii) a message gets corrupted during its transmission; and (iv) a
message can get replicated during its transmission. We will assume that messages contain sufficient redundancy (e.g. checksum) to enable a receiver to discard corrupted messages. Thus we will only concern ourselves with failures of type (i), (ii) and (iv) which will collectively be referred to as 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 RPCs 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 RPCs 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 normal termination is possible.

(i) No communication and/or node failures occur during the call. (ii) The RPC mechanism copes with a fixed finite number of communication failures. (iii) The RPC mechanism copes with a fixed finite number of communication failures and server node crashes (server crashes, for short). (iv) Same as (iii), but in addition, tolerance to a fixed finite number of client node crashes (client crashes, for short) is also present.
A closely related design issue is to do with the semantics of remote calls where the following classification is most widely accepted [1,3]:

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

(ii) **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 (i)) or zero, partial or one execution (for type (ii)) are a possibility. A 'stronger' semantics is specified by the third type given below [4]:

(iii) **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 the semantics is indeed one of the most important decisions to be taken in an RPC design. We will next consider the problem posed by orphans. Orphans are unwanted executions that occur due to failures such as node crashes. We will use the term **abnormal termination** to refer to a call that does not terminate normally. 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: (i) the
server did not receive the call message; (ii) the reply message did not reach the client; (iii) the server crashed during call execution and either has remained crashed or is not resuming the execution after crash recovery; and (iv) the server is still executing the call in which case the execution could interfere with subsequent activities of the client, as depicted in Figure 1 below.

Figure 1: Example of interference caused by a time out.

The client K at node A issues a call to server X at node B that executes the requested work ("work 1" in Figure 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 Figure 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 Figure 1 must be regarded as undesirable, since the execution of a sequential program should give rise to a sequential computation characterised by a single flow of control. Con-
currency 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 Figure 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 Figure 2 is thus possible.

```
Node A          Node B          Node C

CLIENT K
...
{1st call} ----------->+ "call" SERVER X
     | "work" |
     +----------->+
           "call" SERVER Y
           +----------->+
           "work" |
           +----------->+

**********
* Crash *
**********

**********
* Time out*
**********

{2nd call} ----------> "call" SERVER Z
                   +----------->+ "work" |
                   | Interference |
```

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

Note that this type of interference in a nested call can also occur in the absence of a server crash, as illustrated in Figure 3.
Figure 3: Possible interference in a nested call (no crash case).

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? This will depend upon the semantics of the call to be supported [5]. Let us consider 'at least once' calls first. There are two aspects of orphan treatment. (i) Recall that a normally terminated call can give rise to multiple executions at the called server - all of which must be scheduled to run without any interference. If we assume that such calls are idempotent, then no particular ordering need be imposed. (ii) When we consider a sequence of calls, then not only we should prevent interference, but also impose a particular order
of execution such that orphan executions (such as "work 1", Figure 1) do not overtake the "current" execution (such as "work 2", Figure 1). That is, referring to Figure 1, the only permissible execution sequence at node B is "work 1" followed by "work 2" (of course, if "work 1" and "work 2" computations are disjoint - have no variables in common - then strictly speaking these computations can be run in any order since they will not interfere). When we consider 'exactly once' calls, then we only need to impose the second condition stated above. One way of meeting condition (ii) is to make sure that orphans are detected and aborted before a new call is issued. A number of orphan killing techniques have been discussed in [3, 5]. They tend to be expensive and difficult to implement (since orphan killing methods themselves must be robust against communication and node failures). Finally, for 'at most once' calls, condition (ii) needs to be strengthened to include not only just abortion of orphans, but also undoing of any side effects they may have produced.

2.2. Protocol and Performance related Issues

Much has been written about the desirability of "light weight" protocols for distributed systems (e.g. [6]); and it is generally agreed that simple datagrams provide an adequate transport service for implementing RPC protocols [2,3,7,8]. A 'call' or a 'reply' message can be required to carry data objects of arbitrary size, which suggests that a protocol for fragmentation and reassembly is required. If the transport layer supports datagrams of a small fixed size (e.g. 532 bytes as in PUP [9]), then the RPC protocol must handle fragmentation and reassembly. On
the other hand, if datagrams of practically unlimited size are supported (e.g. as in [10]), then RPC protocol need not be concerned with their provision.

Experience with protocol implementations has indicated that copying of data from one data area to another, context switches and process creation are major causes of performance degradation and protocols should be implemented to avoid the above operations as much as possible. This observation influences both the functionality of the transport layer (favouring, as it turns out, large datagrams [11]) and the execution model of RPCs. In particular, the RPC execution model should strive for small number of process creations during calls.

We summarize this section by enumerating important design decisions that need be taken in an RPC design: (i) selection of semantics; (ii) selection of fault tolerance capabilities (i.e. conditions under which normal termination of a call can happen); (iii) provision of orphan treatment facilities as required by the chosen call semantics; and (iv) functionality of underlying transport service and RPC execution model.

3. RPC Design

This section describes the actual RPC design and discusses reasons for the particular design decisions that were taken. We wanted the RPC mechanism to be general purpose, rather than for simply invoking idempotent operations, which limits the choice to 'at most once' and 'exactly 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 (for undoing side effects of calls to be aborted); indeed, the two published proposals [4,12] both have integrated robust atomic actions (atomic transactions) with RPCs. We would like our RPC mechanism to be sufficiently 'neutral' in order to support applications that do and do not make use of atomic actions, which suggests that 'exactly once' semantics is more appropriate (we will return to this topic in the last section of the paper).

The following primitive has been made available to clients for invoking a remote call (where parameters and results are passed by values):

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

The "rpc_status" variable can assume one of the following values:

```plaintext
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, NOT-DONE and UNABLE is given below:
rpc-status=OK: The specified call has been executed exactly 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 Figures 1 and 2, or normally as in Figure 3) there will be no ongoing computations for that client at nodes B and C.

3.1. Fault Tolerance Capabilities

Out of the four conditions for normal termination presented in section 2.1, 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 implement its own crash resistance strategy, rather than to fix it at the RPC level. The choice then is between (ii) and (iii) and we have opted for (ii). That is, a call can complete normally in the presence of communication failures, but not if the server crashes (in which case the call is guaranteed to terminate abnormally). Why not let a call complete normally in the presence of server crashes? The problem is that - unless proper crash recovery procedures are employed - there is no guarantee that pre-crash results have not been destroyed by a crash. So, if a server after recovery restarts execution then the final outcome could be
not that of exactly one execution but one or more. The simplest solution is to prevent such situations by insisting that calls terminate abnormally whenever servers crash.

Finally, we provide the following orphan handling capability to cope with client crashes. 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, they will be aborted before execution of the call starts at C. 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.

3.2. Comparison with other RPCs

Most of the remaining part of the paper will be about how we have implemented Rajdoot to meet exactly once semantics with the fault tolerance properties stated here. So it is appropriate at this stage to compare our RPC with currently available RPC mechanisms. First of all we claim that Rajdoot closely approximates the behaviour of local calls. For a sequential program, when a local call terminates either normally or abnormally (an exceptional return is obtained), we do not expect any ongoing activities at the callee: the same behaviour is modelled by our RPC. In a single node system, a crash halts all the ongoing computations. This behaviour 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.
The first detailed study of RPCs 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 RPCs [13,14], 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.

We are not aware of any working distributed implementations of RPCs supporting 'at most once' semantics. As far as we can tell it is hard (or pointless) to implement such RPCs without the support of atomic actions, so comparisons with simple general purpose RPCs discussed earlier is not meaningful.

3.3. The Execution Model and RPC protocol

The execution model adopted has been influenced by the use of RPCs in the Newcastle distributed system [15], and is described here with the help of a diagram (see Figure 4). Each node runs a manager process that operates at some well known address; in the following we shall assume
that managers' addresses are known throughout the system (for example, using a name server allowing managers to publish their addresses). 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.

Node A

CLIENT

... + "create server" -------------> MANAGER

| + fork

| + "server address" | <-----------------+

{1st call} rpc < + "call" + "work"

| + "reply" + "work"

... + "call" + "work"

{2nd call} rpc < + "reply" + "work"

... + "reply" + "work"

Figure 4: RPC Execution Model.

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 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 Figure 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 Figure 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 non zero, 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 proc-
cess 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.

As can be seen, our protocol relies on timeouts to prevent a proc-
cess waiting forever (or for a long duration). Such use of timeouts cer-
tainly simplifies protocol design; but a disadvantage is that clients
have to specify timeout intervals for calls. We opted for this solution
rather than the use of 'probe packets' to detect node crashes as in
Cedar RPC — because timeouts also form the basis of aborting computa-
tions as discussed in the next subsection.

The protocol has been implemented on top of the 'Uniform Datagram
Service' (UDS) interface [10,11]. This is an interface providing a
datagram service for packets of practically unlimited size, together
with 'scatter and gather' facilities so that variable length data units
from a set of non-contiguous areas of memory at a source can be sent as
a single packet to a destination. This has meant that no fragmentation
and reassembly facilities are required to be implemented by the RPC pro-
tocol. The UDS interface has been designed with a view to minimize
expensive context switching and data copying operations [11]; in addi-
tion, the RPC execution model does not require extensive process cre-
tions, so the overall design has been geared towards obtaining high per-
formance.
3.4. 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 (i) the client has crashed or it can not send a request due to some communication failure; or (ii) 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 discussed 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 <= 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 <= 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 now discuss how orphans due to node crashes are detected and aborted. The basic idea is quite straightforward. Every node maintains a variable '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 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 can not 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. The orphan detection and killing scheme described above is not complete - an extra mechanism is still required to cope with the following two situations: (i) a node crashes after which it is not recovered (or is rebooted after a very long time); (ii) after recovery, a node does not call nodes with orphans. These situations are dealt with as 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: (i) the client has crashed and not yet recovered; (ii) the client has crashed, but after recovery no calls to the node have been made; (iii) the client program has terminated, without informing the server; (iv) communication between the nodes is no longer possible. Out of these four possibilities, we have chosen not to deal with possibility (iii), believing it to be the responsibility of clients to terminate servers, and possibility (iv) 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
Every node has a 'terminator' process that regularly (every 15-20 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 are not deemed excessive. This is because a terminator need only activate itself infrequently and it does not generate excessive message traffic. This completes the discussion on the reliability mechanisms of Rajdoot.

3.5. Implementation Notes and Performance

A working implementation of Rajdoot has been performed over a few PDP11's with UNIX connected by a Cambridge Ring. This subsection reports on some details of this implementation performed in the C language (complete details are presented in [16]). Plans are at hand to transfer the implementation to the recently acquired Vaxes with 4.2 BSD UNIX connected by Ethernets.