# Failure Transparency in Remote Procedure Calls

K. RAVINDRAN, MEMBER, IEEE, AND SAMUEL T. CHANSON, MEMBER, IEEE

***Abstract*—Remote procedure call (RPC) is a communication abstraction widely used in distributed programs. The general premise entwined in existing approaches to handle machine and communication failures during RPC is that the applications which interface to the RPC layer cannot tolerate the failures. The premise manifests as a top level constraint on the failure recovery algorithms used in the RPC layer in these approaches. However, our premise is that applications can *tolerate* certain types of failures under certain situations. This may, in turn, relax the top level constraint on failure recovery algorithms and allow exploiting the inherent tolerance of applications to failures in a systematic way to simplify failure recovery. Motivated by the premise, the paper presents a model of RPC. The model reflects certain generic properties of the application layer that may be exploited by the RPC layer during failure recovery. Based on the model, a new technique of adopting orphans caused by failures is described. The technique minimizes the rollback which may be required in orphan killing techniques. Algorithmic details of the adoption technique are described followed by a quantitative analysis. The model has been implemented as a prototype on a local area network. The simplicity and generality of the failure recovery renders the RPC model useful in distributed systems, particularly those that are large and heterogeneous and hence have complex failure modes.**

***Index Terms*—Client-server model, orphans, partial failures, roll back, state inconsistency.**

## I. Introduction

DISTRIBUTED systems are becoming larger and heterogeneous, with computing resources distributed extensively across hundreds of machines interconnected by one or more local area networks (LAN's) through gateways. The processes that manage resources are called *servers* (also referred to as services) and the processes that access the resources are called *clients*. Examples of services are terminals, printers, files, time information, name assignment, and mathematical library computations. A client communicates a request to a server to access a resource, and the server communicates the outcome of the request to the client by a response (*request-response* style of communication). A service may be provided by a group of server processes executing on different machines with functions replicated and distributed among the processes for reasons of availability and performance. For example, a time service may consist of a group of server processes with each one providing time information to clients. Multiple requests for time may be handled concurrently by the various processes. If a process in the group fails, the time service may continue to be provided by the other processes in the group. Thus, in a large scale distributed system, a program implementing an application[1] often consists of clients and servers residing on different machines and communicating extensively across machine boundaries. Such programs are referred to as *distributed programs*.

Remote procedure call (RPC) is widely accepted as a natural and convenient abstraction that may be used in distributed programs to map onto the client-server communications [6], [3] because the RPC encapsulates the easily understood procedure call mechanism that allows a client to access remote services in much the same way as local services. On the part of the system, a semantics of RPC close to that of a local procedure call should be provided. A key requirement is that the machine and communication failures during an RPC [9], [14] should be masked in the RPC interface to the program so that the program may function normally in the presence of the failures (failure transparency).

Machines are assumed to exhibit a fail-stop behavior [16]. Typical communication failures include: messages used for the RPC being lost or misordered in the gateways due to congestion, network partitioning due to gateway failures, and persistent message loss at the gateways and network interfaces. Frequently, the failures result in server executions continuing to exist even after termination of the RPC requests from clients. Such server executions are known as *orphans* [9].

Treating failures as a subset of RPC events, existing RPC models deal with orphans by enforcing atomicity and ordering constraints on the RPC events. In other words, an RPC event (e.g., RPC request, network failure) seen by a client should also be seen by the server and vice versa, and in the same order with respect to other causally related events. Suppose during an RPC on a server, the client terminates its request because it sees a temporary network failure. As per existing RPC models, the order of events at the server should be for the server to receive the request, then see the failure and terminate the requested operation. If the server does not see the failure (violation of atomicity), or if the server sees the failure after it has completed the requested operation (violation of ordering), the models consider the operation incorrect. Furthermore, since the orphan may interfere with normal executions subsequently requested by the client (or other clients), it it killed by using techniques such as rollback [6], [5]. Such a treatment of failures is independent of the applications.

Manuscript received September 15, 1988; revised April 1, 1989.

K. Ravindran is with Bell Northern Research, Ottawa, Ont., Canada.
S. T. Chanson is with the Department of Computer Science, University of British Columbia, Vancouver, B.C., Canada, V6T 1W5.

IEEE Log Number 8928526.

[1] Applications are programs that are written by system programmers who implement the resource-dependent component of the servers (e.g., terminal, file) or system users who implement their own specific needs (e.g., numerical program, database access program).

In this paper, we view the implications of failures from an application perspective as outlined below [1].

### A. Inherent Failure Tolerance of Applications

Many applications have an *inherent* ability to tolerate certain types of failures that may occur during RPC's. This is partly due to the evolution of a wide range of *idempotent* applications that do not change the state of the server such as query to time servers, remote computations on math library servers, access to name servers, and so on in large scale distributed systems. Typically in these applications, a server may process the RPC requests from multiple clients in any order since the requests are usually unrelated to one another. Also, the failure of a client need not be seen by the server since the failure usually does not affect the server. Thus, the servers in these applications need not enforce the atomicity and ordering constraints on the RPC events. This absolves the servers from maintaining state information which may otherwise be required if the constraints are to be enforced. This is in contrast to applications such as operations on file and database servers which usually enforce the atomicity and ordering constraints on the RPC events. Even so, a server need not enforce the constraints for a sequence of idempotent operations (e.g., reading a file).

The following examples further illustrate how applications exhibit some level of failure tolerance.

*Examples:* Consider an RPC by a client to search for a file or to get time information from a group of server processes. In both cases, the RPC event need not be seen by every server in the group. For the file search, it suffices if the client gets a response from the particular server that manages the file. For the time request, response from any of the servers will do. Thus, a communication failure which results in nondelivery of the RPC event to every server in the group does not affect the successful completion of the RPC. As another example, consider the multiple executions of a server caused by retransmissions of an RPC request message to the server from a client, say due to message loss. The orphaned server executions [8], [9] may not be harmful when they are idempotent. Consider the earlier example of an RPC on a server where the client terminates its request because it observes a temporary network failure. If the server execution is idempotent, then it does not matter whether the server observes the failure before or after completing the execution, and in some cases if the failure is observed at all by the server.

Since many such applications can tolerate certain types of failures, we suggest that the ordering and atomicity constraints on the RPC events need not be subsumed in the RPC layer but may be specified by the application layer above it. In other words, the ordering constraints on a given sequence of RPC events depend on the application. This premise allows relaxation of the constraints in the RPC layer using application layer information which may in turn significantly simplify the recovery algorithms.

Thus, failure transparency in RPC requires specifying the *failure semantics* of RPC (i.e., the implications of failures during RPC) and the treatment of orphans caused by failures. Existing RPC models typically do not make use of the application layer properties for failure recovery, and are either formulated primarily for nonidempotent applications or do not address failure transparency significantly. This paper presents a different model of RPC from an application perspective. The model makes new types of failure recovery techniques useful, particularly in large distributed systems.

The paper is organized as follows: Section II describes a model of RPC which systematically incorporates certain application layer properties and allows them to be exploited during failure recovery. Section III discusses the failure semantics of RPC. Based on the RPC model and semantics, Section IV introduces a new technique of *adopting* orphans caused by failures. The technique minimizes rollbacks that may be required for recovery and avoids wastage of useful work already completed. Section V describes the essential details of the technique. Section VI presents a quantitative analysis of the recovery technique. Section VII provides details of a prototype implementation of the model and includes performance indications. Section VIII discusses the model in relation to existing work.

## II. Model of Remote Procedure Call

As described earlier, server processes implement resources and respond to requests from client processes to access the resources. A server exports an abstract view of the resource (e.g., files) it manages with a set of operations on it. A client communicates an RPC request to the server for operations on the resource, and the server communicates the outcome of the operations to the client by an RPC response (or return). In providing the resource for its clients, the server often needs to communicate as a client with another server because the resource may be implemented on top of another resource. For example, files are implemented on top of disk storage; so a file server needs to communicate as a client with a disk server to implement the files. Thus, the role of a process as client or server is dynamic.

Additionally, a service may be provided by a group of server processes organized into a process group [13], referred to as a *server group*, to manage the resource. The member processes of a server group share one or more abstract resources and contend among themselves to access the resources. Examples of the resources are the name binding information maintained by a name server group, the leadership within a server group, and distributed load information. The *contention style* intraserver communication may take place by one-to-many (group) communications among the members of the server group. The intraserver group communication initiated by a server is usually triggered by an RPC request on the server from a client. Thus, a distributed program may be structured as a sequence of client-server communications interspersed with intraserver group communications. The latter may span across program boundaries because a shared resource managed by a server group may be accessed from more than one program.

### A. RPC Types

RPC's from a client on a server may be of two types—*connection-oriented* and *connection-less* [8]—as described below:[2]

An RPC is connection-oriented if in a sequence of such calls, the server should maintain a certain ordering relationship among them. The call (or interaction) may cause changes to the resource the server exports to the client. The server maintains state for the interaction which consists of i) information about the client, and ii) resource-dependent information which is anchored onto i). (Item i) constitutes the permanent variable for the connection.) The state is maintained in the server across calls throughout the duration of the connection. Among other things, the state information is used by the server to maintain the required ordering relationship among the calls, and to protect the resource against inconsistencies caused by client failures. An example of a connection-oriented call is a client operating on a file maintained by a file server; part of the state maintained by the server for the call is the seek pointer.

An RPC is connection-less if in a sequence of such calls, the server need not maintain *any* ordering relationship among them. This implicitly assumes that the call should not cause any changes to the resource the server exports to the client. Thus, the failure of the client is of no concern to the server. For the above reasons, the server need not maintain any state information for a connection-less call. Examples of connectionless calls are a client requesting time information from a time server, and a numerical computation from a math library server.

Because no ordering constraints are imposed, the connection-less calls are lightweight and the algorithms to implement the calls may be simpler and more efficient as compared to connection-oriented calls. The failure recovery component of the algorithms may also be simpler (Section IV-D).

We now discuss how state transitions occur in servers to formalize the application layer properties that may be used in the RPC.

### B. State Transitions in Servers

An RPC $TR$ from a client on a server is denoted by

$$(C_{\text{bef}}, S_{\text{bef}}) \xrightarrow{TR} (C_{\text{aft}}, S_{\text{aft}}) \tag{1}$$

where $C_{\text{bef}}$ and $C_{\text{aft}}$ are the states of the client before and after the execution of $TR$, and $S_{\text{bef}}$ and $S_{\text{aft}}$ are the corresponding states maintained by the server for $TR$. The $TR$ causes the server in state $S_{\text{bef}}$ to emit a value $p\_val$ and change state to $S_{\text{aft}}$, and the client in state $C_{\text{bef}}$ to accept $p\_val$ and change state to $C_{\text{aft}}$. The state transition from $S_{\text{bef}}$ to $S_{\text{aft}}$ in the server may take place by its interactions as a client with other servers and by its local executions operating on its internal permanent variables. Thus, $C_{\text{aft}}$ depends on $(C_{\text{bef}}, p\_val)$ and $S_{\text{aft}}$ depends on $(S_{\text{bef}}, TR)$. If $TR$ is connection-less, it is simply denoted by

$$(C_{\text{bef}}) \xrightarrow{TR} (C_{\text{aft}})$$

since the server does not maintain any state information for $TR$.

[2] The meanings of these terms differ somewhat from those used in communication protocols.

The $p\_val$ may be abstracted as a set of (attribute, value) pairs. An attribute, used by the client, specifies an operation on the server which may return one of many possible values for the attribute. As an example, suppose $TR$ is a query to a print server to get the status of a printer. The attribute *STATUS* may be specified in $TR$. Let the possible return values for the attribute be $\{ACTIVE, DOWN\}$. Then one possible outcome of $TR$ is $p\_val = \{(STATUS, DOWN)\}$. Such a characterization of $p\_val$ is in general useful to transmit abstract values in messages [12]. In particular, it is used to represent the return value in RPC (Section IV-B).

Based on state transitions in the server, we now describe the idempotency property of client–server interactions. It is an application layer property used in RPC for failure recovery.

### C. Idempotency

Consider a client–server call $TR$ as given by the relation (1)

$$(C_{\text{bef}}, S_{\text{bef}}) \xrightarrow{TR} (C_{\text{aft}}, S_{\text{aft}}).$$

The idempotency property of $TR$ [9] relates to the effect of $TR$ on the state maintained by the server for the calls from the client, and it specifies the ordering relationship of $TR$ with respect to a sequence of calls. $TR$ is an idempotent call if the state of the server remains unchanged after the execution of $TR$, i.e., $S_{\text{aft}} = S_{\text{bef}}$; however, $C_{\text{aft}}$ need not be the same as $C_{\text{bef}}$ since the client may change state due to the $p\_val$ returned from the server. Examples of idempotent calls are a read (without seek) operation on a file and a status query operation on a printer. If $TR$ is nonidempotent, then $S_{\text{aft}}$ may be different from $S_{\text{bef}}$. Examples of nonidempotent calls are relative seeks on a file and opening a file.

To expose additional properties of $TR$ that may be useful in the recovery algorithms, we introduce two concepts—*reenactment* of $TR$ and *reexecution* of $TR$.

*1) Reenactment:* In a reenactment of $TR$, the states of both the client and the server are first restored to those when $TR$ was issued and a new call $TR'$ which has the same properties as $TR$ is made. If $TR$ is given by the relation (1), then $TR'$ is defined as

$$(C_{\text{bef}}, S_{\text{aft}}) \xrightarrow{TR'} (C_{\text{aft}'}, S_{\text{aft}'}),$$

where $C_{\text{aft}'}$ depends on $(C_{\text{bef}}, p\_val')$ and $S_{\text{aft}'}$ depends on $(S_{\text{bef}}, TR')$. The concept of call reenactment is useful in backward recovery schemes in which the server rolls back the effect of the call, and subsequently the client reissues the call (Sections V-D and III-A). The idea is to be able to reproduce the effect of the call (i.e., $S_{\text{aft}'} = S_{\text{aft}}$ and $C_{\text{aft}'} = C_{\text{aft}}$). In order to accomplish this, the server state transition and the call $TR$ should be *deterministic*, i.e., repeated call on the server at a given state should cause the server to make the same state transition and emit the same $p\_val$. The former condition ensures $S_{\text{aft}'} = S_{\text{aft}}$ while the latter ensures $C_{\text{aft}'} = C_{\text{aft}}$. Consider, as an example, a "read" operation provided by a file server that returns the data value read from a file. It is deterministic since a reenactment of the operation returns the same value as the original operation. Suppose the "read"

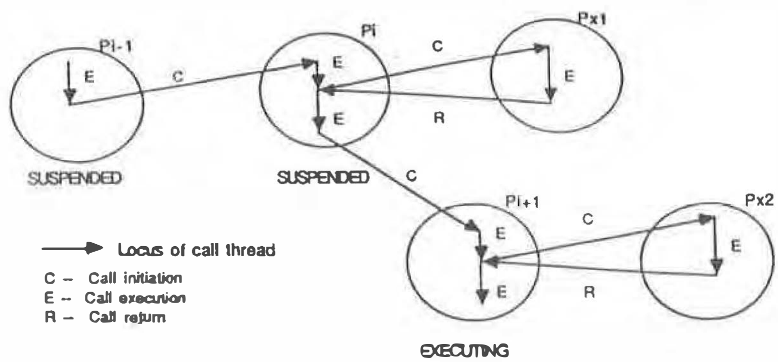


Fig. 1. Locus of the remote procedure call thread.

operation also returns a time stamp, then it is nondeterministic since every reenactment of the operation may return a different time stamp.

We observe that the change in the server state caused by $TR$ depends only on the server state prior to the execution of $TR$, but not on the *p_val* returned by the server. On the other hand, the change in the client state depends only on the client state prior to the execution of $TR$ and on the *p_val* returned by the server, but not on the server state. Thus, the idempotency and the determinism properties of $TR$ do not interfere with one another. Hence, any techniques to deal with the nondeterministic behavior of program executions need not interfere with those provided to tackle the idempotency issues. Thus, for simplicity and without loss of generality, we confine our discussion to deterministic programs.

### D. Reexecution

In a reexecution of $TR$, only the client state is restored to that when $TR$ was first initiated. In that state, the client generates a new call $TR''$ such that $TR''$ has the same properties as $TR$. If $TR$ is given by the relation (1), then $TR''$ is defined as

$$(C_{\text{bef}}, S_{\text{aft}}) \xrightarrow{TR''} (C_{\text{aft}''}, S_{\text{aft}''}).$$

The concept of call reexecution is useful in the forward recovery scheme described in Section IV and also in dealing with orphans caused by message duplicates (Section V-B1).

In order for a reexecution to be useful, $TR$ should be idempotent. It follows from the definition of idempotent calls (Section II-C) that if $TR$ (and therefore $TR''$) is idempotent, then $S_{\text{aft}''} = S_{\text{aft}} = S_{\text{bef}}$. In other words, the server state does not change under reexecutions of an idempotent call. Also, since $TR$ is deterministic, $C_{\text{aft}''} = C_{\text{aft}}$.

Based on the above concept of reexecution, the call $TR$ may further be classified as 1-idempotent if the server changes state only for the first execution of $TR$ but not under reexecutions of $TR$. An example is an absolute seek operation on a file.

Having cast the RPC model with application layer properties, we now discuss the failure semantics of RPC.

## III. Failure Semantics of RPC

Refer to Fig. 1. The $P_i$'s are the processes in the program. Suppose $P_{i-1}$ calls $P_i$ which in turn calls $P_{i+1}$, then $P_{i-1}$ is the client (or caller) of $P_i$ and $P_i$ is the server (or callee) of $P_{i-1}$. Similarly, $P_i$ is the caller of $P_{i+1}$ and $P_{i+1}$ is the callee of $P_i$. The $P_i$'s ($i = 1, 2, \cdots i, i+1$) contain portions of the call thread with the tip of the thread currently residing in $P_{i+1}$. When a caller makes a call on a callee, the caller is suspended and the tip of the call thread extends from the caller to the callee which then begins to execute. When the callee returns, the call thread retracts from the callee to the caller and the latter resumes execution.

As the call thread executes $P_i$, it may visit various servers $P_{i+1}$, $P_{x1}$, $P_{x2}$, $\cdots$ through a series of calls causing the servers to change states (c.f. Section II-B). We refer to the state of all such servers as the state of the environment as seen from $P_{i-1}$. The thread may resume execution in $P_{i-1}$ when it returns from $P_i$ either normally after completion of $TR$ by $P_i$ (i.e., $TR$ succeeds), or abnormally when $P_i$ fails or when there are communication failures between $P_{i-1}$ and $P_i$ (i.e., $TR$ fails).

Suppose $X$ is the state of the environment when the call $TR$ is initiated, a desired failure semantics of $TR$ is as follows. If $TR$ succeeds, $P_{i-1}$ should see the final state of the environment $Y$, otherwise, $P_{i-1}$ should see the initial state $X$. These two outcomes are represented as $CALL\_SUCC(TR, X, Y)$ and $CALL\_FAIL(TR, X, X)$, respectively, where $(TR, X, Y)$ indicates a state transition from $X$ to $Y$ for $TR$. The semantics underscores the *all-or-nothing* effect of the call, a requirement for the call to be atomic [5].

### A. Rollback and CALL_FAIL

Suppose that during the execution of $TR$, $P_i$ initiates a call on $P_{i+1}$ and then fails. The portion of the thread at $P_{i+1}$ down the call chain is an orphan. Let $X'$ be the state of the environment when $P_i$ failed. The failure of $P_i$ can be masked from its communicants $P_{i-1}$ and $P_{i+1}$ if the failure can be recovered and $P_{i-1}$ sees the outcome CALL_SUCC. A necessary condition for such a failure transparency is that there exists another process, identical to $P_i$ in the service provided, whose state is the same as that of $P_i$ when the latter failed and which can continue the execution of $TR$ (from the failure point), causing the state of the environment to change from $X'$ to $Y$. If the failure cannot be masked, then the failure semantics requires that $P_{i-1}$ sees the outcome CALL_FAIL. The latter is provided by killing the orphan [5], [3], [9] which manifests in rolling back the state of the environment from $X'$

to $X$. The semantics is also applicable if $TR$ is connection-less, but the rollback is not required because $P_i$ does not maintain any state.

### B. Unrecoverable Calls

Rollbacks initiated in the RPC layer may affect the application layer, particularly i/o operations that affect the external environment. In some applications such as a print operation, rollback may not be meaningful. In other applications, rollback may not be possible. Consider, for example, operations in certain real-time industrial plants; undoing the effects (on the environment) of an operation such as opening a valve or firing a motor is neither meaningful nor feasible. A call that so affects the external environment is unrecoverable when a failure occurs. The outcome of the unrecoverable call is referred to as *CALL_INCONSISTENT*$(TR, X, X')$ indicating (to $P_{i-1}$) that the state of the environment may be inconsistent.

Since the CALL_FAIL and CALL_INCONSISTENT outcomes of $TR$ occur due to unrecoverable failures in the RPC layer, they are delivered to the application layer as *communication exceptions*. Handlers may be provided to deal with the exceptions in an application dependent manner, say, either by aborting the program or by taking an appropriate corrective action. As an example, suppose $P_i$ is a time server and $P_{i-1}$ periodically calls $P_i$ to obtain time information and update its local time. If $TR$ fails with the CALL_FAIL exception, $P_{i-1}$ may deal with the exception by ignoring the failure, hoping to correct the time at the next call. The effects of communication exceptions in the presence of concurrent calls is dealt with in [1].

### C. Failures of Server Group Members

The implications of the failure of a member of a server group during an RPC are application-dependent. Take for example the case where a member of the group holds a lock on a shared resource. If the member fails, the lock should be released so that the resource is usable by the other members. Thus, as part of the lock acquisition, the member should also arrange for the restoration of the lock to a consistent state in case the member fails [1]. Hence, lock recovery becomes part of a rollback activity that may be initiated by the member if its client fails. The failure of a member that does not hold any lock on the resource may not introduce any inconsistency in the state of the resource.

In this section, we have described a model of distributed programs which allows relaxation of the atomicity and the ordering constraints on the RPC events (including failures) using application layer information such as idempotency, determinism, and connection-less calls. We now describe a new technique to deal with orphans based on the model.

## IV. Orphan Adoption in RPC

In this technique, one of the replicas of the server executes a client call while the other replicas are standing by. When the executing replica fails, one of the replicas $R$ continues the server execution from the point of failure and *adopts* the orphan caused by the failure (rather than killing it which causes rollback). The adoption may occur when $R$ reexecutes from its restart point to the adoption point (typically the point where the process failed) in the same state as the failed process. During the reexecution, $R$ (re)issues the various calls embedded in the call thread from the restart to the adoption points. However, the reexecutions by the various servers due to such calls should cause no effect on the environment (c.f. Sections II-C and IV-A). The *roll forward* of $R$ minimizes rollback (and the associated rollback propagation) which are required in orphan killing techniques [4].

Refer to Fig. 1. Consider the failure scenario described earlier in Section III, namely $P_i$, during the execution of a call initiated from $P_{i-1}$, fails after initiating a call on $P_{i+1}$. Suppose a standby $P_i$ recovers and rolls forward. If the orphan $P_{i+1}$ is adopted by the recovering $P_i$, then $P_{i+1}$ can make a normal return to $P_i$. If the roll forward is not possible, then the call fails. To deliver the CALL_FAIL outcome, rollback (killing the orphan) may be required. If rollback is not possible (unrecoverable call) or if the CALL_FAIL outcome is not required, then the outcome CALL_INCONSISTENT is delivered.

Refer to Fig. 2. Roll forward is based on 1) controlled reexecution of the calls if necessary, based on their idempotency properties, and 2) replay of call completion events from an *event log* so that a recovering process becomes consistent with other processes without actually reexecuting the calls. These points are elaborated in the following sections.

### A. Reexecution of Call Sequences

Let $EV\_SEQ = [TR^1, TR^2, \cdots TR^i, \cdots TR^k]$ be the sequence of call events seen by a server when there are no failures. The ordering on $EV\_SEQ$ is denoted by $TR^1 > TR^2 > \cdots > TR^i > \cdots > TR^k$, where $TR^1 > TR^2$ means "$TR^1$ happens before $TR^2$." The call $TR^i$ is represented as

$$(C_{i-1}, S_{i-1}) \xrightarrow{TR^i} (C_i, S_i) \tag{2}$$

where $(C_{i-1}, S_{i-1})$ is the state of the client and the server, before the execution of $TR^i$ and $(C_i, S_i)$ is the state after the execution. Suppose a failure causes a reexecution of $TR^i$, represented as $TR^{i'}$, after the server has executed $TR^k$. The call sequences $EV\_SEQ$ and $[EV\_SEQ > TR^{i'}]$ are not ordered with respect to one another. Thus, call reexecutions by the server often require relaxation of ordering constraints on calls without affecting the consistency of the server state. The reexecutions of a call underscore the idempotency property associated with the call (c.f. Section II-C) as described below.

*1) Interfering Calls:* Refer to the example given above. Let $S_k$ be the state of the server after the completion of the last call $TR^k$ in $EV\_SEQ$. Assuming that the server does not maintain an event log, the reexecution $TR^{i'}$ (i.e., $TR^k > TR^{i'}$) invoked by a recovering client may interfere with the calls in $EV\_SEQ$ which the server had already completed. $TR^{i'}$ does not interfere with $TR^k$ if

$$(C'_{i-1}, S_k) \xrightarrow{TR^{i'}} (C'_i, S_k).$$

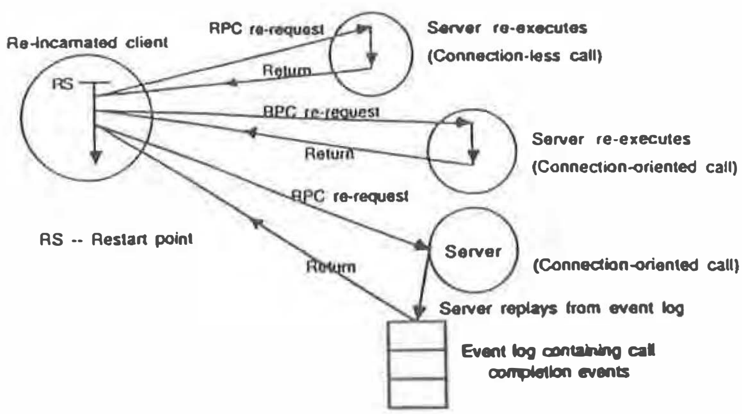


Fig. 2. Recovery of a procedure.

Thus, the necessary condition for the server to execute $TR^{i'}$ without causing state inconsistency is that $TR^{i'}$ should be idempotent. However, a sufficient condition is given by the requirements [see relation (2)] that

$$C'_{i-1} = C_{i-1}, \text{ and } C'_i = C_i.$$

Assuming the call is deterministic, the first requirement is satisfied. Thus, $TR^{i'}$ may be given by

$$(C_{i-1}, S_k) \xrightarrow{TR^{i'}} (C'_i, S_k).$$

Pattern matching this relation with (1), the second requirement, namely $C'_i = C_i$ can be satisfied only if $S_{i-1} = S_i = S_k$. This is true if the condition

$$(C_{i-1}, S_{i-1}) \xrightarrow{TR^i} (C_i, S_{i-1}) \xrightarrow{TR^{i+1}} (C_{i+1}, S_{i-1})$$

$$\cdots (C_{k-1}, S_{i-1}) \xrightarrow{TR^k} (C_k, S_{i-1})$$

is satisfied. This is possible only if $TR^i, TR^{i+1}, \cdots, TR^k$ are all idempotent calls. The condition specifies, in general, when the server may reexecute a call without causing inconsistencies. If $TR^i$ is a 1-idempotent call, then $TR^{i'}$ can be reexecuted only for $i = k$.

The above analysis supports the following commutative property of the calls seen by a server. Given that $EV\_SEQ$ and $[TR^{i'}]$ are *idempotent sequences*, i.e., contain only idempotent calls, $EV\_SEQ > [TR^{i'}]$ is an idempotent sequence (and so is $[TR^{i'}] > EV\_SEQ$). We also observe that $EV\_SEQ > EV\_SEQ' > [TR^{i'}]$ is an idempotent sequence if $EV\_SEQ'$ is an idempotent sequence. The analysis is useful in the server for 1) reexecution of calls, 2) ordering of incoming calls (e.g., generation of serializable schedules for the calls), and 3) interspersing of calls from multiple clients—even though a client may issue a sequence of idempotent calls, if there is at least one nonidempotent call from other clients interspersed in the sequence, the client perceives the effect of a nonidempotent call. We also note that connection-less calls can be interspersed in any serializable schedule.

### B. Event Logs

An event log is used to record an event so that the event can be replayed at a later time. We use the replay technique for connection-oriented calls (without reexecuting the calls) during forward recovery. When a server completes a call, it logs the call completion event (described by a data structure containing, among other things the *p_val* returned by the server to its client). The event log allows the client to perceive the effect of a call without the server actually (re)executing it. Thus, if $TR^i$ is a call represented by [c.f. relation (2)]

$$(C_{i-1}, S_{i-1}) \xrightarrow{TR^i} (C_i, S_i)$$

and $TR^j$ is the call last completed by the server, i.e., $TR^i > TR^j$, then a replay $E^i$ from the event log for $TR^i$ may be represented as

$$(C_{i-1}, S_j) \xrightarrow{E^i} (C_i, S_j)$$

where $TR^j > E^i$. Thus, a recovering client may roll forward to a consistent state with the server simply replaying the logged call completion events.

If a call from a recovering client cannot be completed either from the event log or by reexecution, the call fails with the CALL_INCONSISTENT outcome.

### C. Locks on Shared Resources

If the orphan is holding a lock on a shared resource, the suspension of the orphan during its adoption may prevent other programs from accessing the resource (e.g., a printer or name binding information) until the adoption is completed. Depending on factors such as how critical the resource is and whether the operations on the resource are recoverable, the orphan may either suspend its execution or recover the lock on the resource (c.f. Section III-C) and forces a CALL_FAIL or CALL_INCONSISTENT exception, as the case may be, to the client of the failed process.

### D. Connection-less Calls

Since connection-less calls on a server do not require any form of ordering among them, the calls are not logged by the server. So these calls, when reissued by the recovering client,

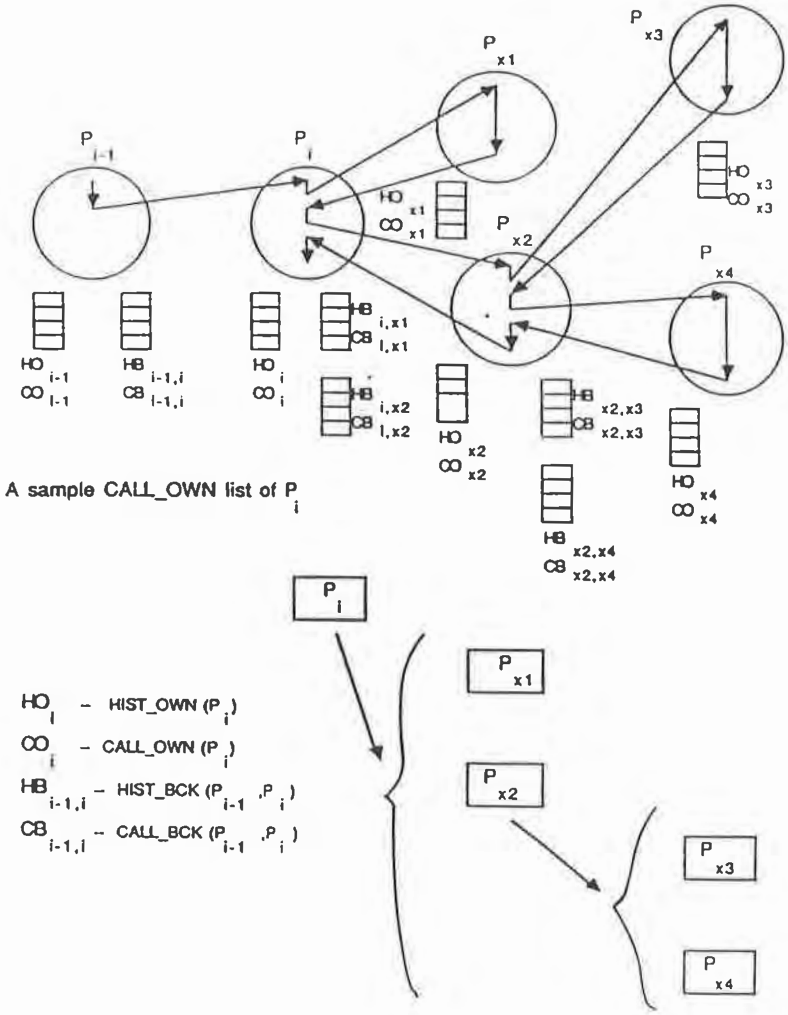


Fig. 3. Data structures used in the RPC run-time system.

are simply reexecuted by the server. Also if a server fails while executing a connection-less call, the client simply reissues the call on a replica of the server. Since our program model encapsulates connection-less calls also, such recoveries are required in the underlying algorithm. In this aspect, the algorithm is distinct from that used elsewhere [2], [7].

The orphan adoption technique is widely applicable to a variety of services since the technique uses only certain generic properties of the services but does not depend on service specifics. In the following section, we describe the algorithms and protocols used by the RPC layer to realize orphan adoption.

## V. Failure Recovery Algorithms

Refer to Fig. 3. The procedure $P_i$ may be in one of three states—EXECUTING state when the tip of a call thread is currently in $P_i$, SUSPENDED state when $P_i$ has made a call on another procedure, and IDLE state otherwise (i.e., no thread is passing through $P_i$).

Suppose $P_{i-1}$ is the client of $P_i$. We let $P_{i-1}$ assume the role of the recovery initiator for $P_i$ [referred to as RI($P_i$)] in case $P_i$ fails because $P_{i-1}$ has the name binding information for $P_i$ (c.f. Section V-B) that is needed for failure recovery. Note that $P_i$ acts as the primary among its replicas.

When $P_i$ completes a nonidempotent call, it checkpoints (i.e., saves) its state consisting of permanent variables in a buffer space provided by the run-time system at $P_{i-1}$'s site. The checkpoint is used in the recovery in the event $P_i$ fails. Our choice of $P_{i-1}$ as the checkpoint site is a design decision based on two reasons. 1) Since $P_{i-1}$ is the recovery initiator for $P_i$, the availability of the checkpoint information with $P_{i-1}$ makes failure recovery easier. 2) Since the system environment may consist of diskless machines [10], $P_i$ may not have a local disk to use as stable storage for the checkpoint. The above design decision is in contrast with that of ISIS in which a server checkpoints its state at the other replicas of the server [2].

Suppose $P_i$ fails, RI($P_i$) detects the failure [8] and selects a secondary to reconfigure as the new primary and continue the execution. In our scheme, there is no ordering relationship among the secondaries. Instead, the secondary that responds first to a message broadcast by RI($P_i$) is selected to be the new primary. RI($P_i$) then initializes the new $P_i$ to the state last checkpointed in its site, and rebinds the references to the failed $P_i$ held by its communicants to the new $P_i$. During this entire initialization (INIT) activity, RI($P_i$) sends keep-alive messages to the communicants of the failed $P_i$ to indicate to them that recovery is in progress. These messages prevent the communicants from timing out. Subsequent recovery activities depend on the state the failed $P_i$ was in at the time of failure. If $P_i$ failed when it was IDLE, no activity other than INIT is required. When the prefailure state of $P_i$ was EXECUTING or SUSPENDED, a RESTART activity whereby RI($P_i$) restarts the new $P_i$ is necessary. We describe the RESTART activity in the next section followed by the data structures required for the RESTART, and finally the recovery of $P_i$.

### A. RESTART Activity

RI($P_i$) restarts the new $P_i$ which then starts (re)issuing the calls between the last checkpoint to the point where the erstwhile $P_i$ failed (see Fig. 3). A server (such as $P_{i+1}$) handles such calls sent to it by returning the results ($p\_val$) of the calls to $P_i$. Since the server had already executed the calls previously, $p\_val$ may be obtained from the local event log or, if it is not available in the log, by reexecuting the call if this will not cause state inconsistencies [c.f. Section IV-A1)]. If the server has all the calls sent to it in its log, no reexecution of the calls is necessary. Ideally, the size of the log should be large enough to retain all the calls since the last checkpoint. However, the finite size of the log in any implementation means there is a possibility a nonidempotent call cannot be logged by the server. We consider the following options in handling this problem:

*1) Option 1: Intermediate Checkpoints:* The server (such as $P_{i+1}$) may force its client $P_i$ to take a checkpoint (at $P_{i-1}$'s site). The checkpoint may then occur even before the return of the (nonidempotent) call $P_i$ is executing. Such an intermediate checkpoint has the following implications. 1) The frequency of checkpointing may be higher than the case where checkpointing is done only at call return. This is the case if there are nonidempotent calls arriving after the log is full. 2) The state checkpointed needs to include the instruction pointer, stack pointer, and the execution stack. This may restrict the replicas of a server to run only on machines of the same hardware architecture. 3) Extra checkpoint messages are required, some of which may be piggybacked on the call return messages if the checkpointing is done during call return.

*2) Option 2: Rollback of the Unlogged Call:* The implications of the server being unable to log the nonidempotent call it returns are as follows. If the client ($P_i$ in our case) fails and recovers, the calls which are reissued from the recovering $P_i$ on the server and which are not in the server's log cannot be completed. To enable $P_i$ to roll forward by completing such calls, the effects of the unlogged nonidempotent call should be rolled back before $P_i$ can reissue the calls. If the RPC layer already maintains data structures to support rollback and provide the *CALL_FAIL* outcome, the rollback of the unlogged call does not require any additional data structures. If the rollback cannot be carried out, then since $P_i$ cannot roll forward, it may fail the call by delivering the *CALL_INCONSISTENT* outcome to $P_{i-1}$.

The RPC designer may choose one of the above options after weighing their implications in light of the application environment the system should support. We have chosen option 2 in our implementation because the data structures to support rollback are already available to provide the *CALL_FAIL* outcome.

As noted earlier in Section IV-D, the connection-less calls on a server are not logged by the server, so these calls when reissued by the new $P_i$ are invariably reexecuted by the server. Also if a server fails during a connection-less call on it, the client simply reissues the call on another replica of the server.

### B. RPC Data Structures and Protocols

The RPC layer maintains a set of variables and data structures for recovery purposes (see Fig. 3). Only those essential to describe the adoption technique are given below:

*CALL_REF*($P_i$, $P_j$): It is a name reference to a callee $P_j$ (e.g., $P_{i+1}$) held by $P_i$ in the form of a (*service_name*$_j$, *srvr_pid*$_j$) pair, where *service_name*$_j$ uniquely identifies the service provided by $P_j$ and *srvr_pid*$_j$ is the process id of $P_j$. When $P_i$ makes a call on $P_j$, this reference information is checkpointed at the caller of $P_i$ (e.g., $P_{i-1}$); when $P_j$ returns the call, the checkpointed information is deleted. If $P_j$ recovers after a failure, the process id in CALL_REF is updated to refer to the recovering process.

($G\_tid_{rqst,i,j}$, $nI\_tid_{rqst,i,j}$): $G\_tid_{rqst,i,j}$ is a global call id which is assigned the next sequence number for every new call by $P_i$ on $P_j$. $nI\_tid_{rqst,i,j}$ is a nonidempotent call id which is assigned the next sequence number for every nonidempotent or 1-idempotent call on $P_j$. The call id pair maintained by $P_i$ (as a client) pertains to its last call on $P_j$. The set of such pairs is referred to as the *thread state* of $P_i$.

($G\_tid_{last,i}$, $nI\_tid_{last,i}$): It is the call id pair maintained by $P_i$ (as a server) for the last call it has completed.

*CALL_THRD*($P_i$, $P_j$): It is the thread state of $P_j$ checkpointed at its caller $P_i$. If $P_j$ fails and recovers, it is initialized by $P_i$ to this thread state during the INIT activity.

*CALL_OWN*($P_i$): It is a recursively structured list of procedure names maintained by $P_i$ in the SUSPENDED or in the EXECUTING state. The first element of the list is the name of $P_i$ itself, and each element of the remaining list is the CALL_OWN($P_j$) returned by a callee $P_j$ when the latter completed a nonidempotent call from $P_i$. Thus, CALL_OWN($P_i$) contains at least one element, the name of $P_i$.

*CALL_BCK*($P_i$, $P_j$): It is a checkpointed version of CALL_OWN($P_j$) maintained by $P_i$ while in the SUSPENDED state for its on-going call on $P_j$.

*CALR_RL_FLAG*($P_i$): It is a Boolean variable (flag) and is meaningful only when $P_i$ is in the EXECUTING or the SUSPENDED state. A true value of the flag indicates that if the caller of $P_i$ fails, a rollback should be performed for recovery; a false value indicates otherwise.

*CALR_ENV_INTRCT*($P_i$): It is a flag meaningful only when $P_i$ is in the EXECUTING or the SUSPENDED state. A true value of the flag indicates that if the caller of $P_i$ fails and recovers, its reexecution up to the failure point will cause at least one interaction with the environment.

*HIST_OWN*($P_i$): It is a list of the values of the permanent

[3] $PV_i$ may be represented in a machine-independent form by using techniques such as external data representation and abstract syntax notation [12], [10].

variable $PV_i$ maintained by $P_i$[3] and its thread state; a value is stored when $P_i$ completes a nonidempotent call. It constitutes the history of $P_i$.

*HIST_BCK*$(P_i, P_j)$: It is a checkpointed version of the history of $P_j$ maintained by its caller $P_i$ (the recovery initiator for $P_j$). Note that the last entry contains the value $PV_j$ to which $P_j$ should be initialized (during the INIT activity) in case $P_j$ fails and recovers.

It should be noted that for connection-less calls from $P_i$ on $P_j$, only the CALL_REF$(P_i, P_j)$ is maintained; all the other data structures are maintained only for connection-oriented calls.

For details of the protocols to send and receive RPC requests and returns, see [1]. We describe below only the call validation phase of the protocols.

*1) Call Validation:* Suppose $P_i$ makes a call request, identified by $(G\_tid_{rqst,i,i+1}, nI\_tid_{rqst,i,i+1})$, on $P_{i+1}$. If $P_i$ is a recovering procedure, then $G\_tid_{rqst,i,i+1} \leq G\_tid_{last,i+1}$ and $nI\_tid_{rqst,i,i+1} \leq nI\_tid_{last,i+1}$ for the reissued calls. Thus, the following situations are possible when $P_{i+1}$ validates the request:

*Case 1)* $G\_tid_{rqst,i,i+1} = (G\_tid_{last,i+1} + 1)$: $G\_tid_{rqst,i,i+1}$ is a new call, so $P_{i+1}$ carries out the requested call and sends the completion message to $P_i$.

*Case 2)* $G\_tid_{rqst,i,i+1} < (G\_tid_{last,i+1} + 1)$: $G\_tid_{rqst,i,i+1}$ is a reissued call (e.g., a duplicate call request message). If the call completion event is available in the log, $P_{i+1}$ replays the event to the recovering $P_i$. Otherwise, if the requested call is idempotent or 1-idempotent, and $nI\_tid_{rqst,i,i+1} = nI\_tid_{last,i+1}$, then $P_{i+1}$ may reexecute the requested call and return the results to $P_i$. If the call is nonidempotent or $nI\_tid_{rqst,i,i+1} < nI\_tid_{last,i+1}$, then $P_{i+1}$ returns an error message ALRDY_OVER to $P_i$.

When $P_{i+1}$ rejects the call request with the ALRDY_OVER error message, $P_i$ may request $P_{i+1}$ to rollback. If $P_{i+1}$ rolls back, $P_i$ may reissue the call. Otherwise, the call fails with the CALL_INCONSISTENT outcome.

When $P_{i+1}$ returns the call to $P_i$, the latter uses CALL_REF$(P_i, P_{i+1})$ and $(G\_tid_{rqst,i,i+1}, nI\_tid_{rqst,i,i+1})$ to validate the return. In general, a client uses its CALL_REF to detect returns from orphaned calls. For this purpose, the process id's used in CALL_REF should be nonreusable [14].

## C. Rollback Algorithm

The structure of the list CALL_OWN$(P_i)$ [and CALL_BCK$(P_{i-1}, P_i)$] reflects the sequence in which the execution thread from $P_i$ visited the various callees. A rollback should follow a last-called-first-rolled order, i.e., only after the rollback for the last call (last entry in the CALL_OWN) is completed should the rollback for the previous call be initiated. Suppose $P_i$ is the rollback initiator (RBI). It recursively traverses its CALL_OWN (or CALL_BCK as the case may be) list in the last-in-first-out order. For each entry in the list, $P_i$ sends a message RL_BCK to the procedure identified in the entry. On receipt of this message, the concerned procedure rolls its permanent variable back to the last value contained in its HIST_OWN, and returns a RL_BCK_ACK message indicating successful completion of the rollback operation. If rollback is not possible, the procedure returns a RL_BCK_FAIL message to indicate the situation. On receipt of the RL_BCK_ACK message from all procedures listed in CALL_OWN, the RBI assumes the rollback is successfully completed. If at least one RL_BCK_FAIL message is received, the RBI considers the rollback to have failed.

A callee need not perform rollback if the calls involved in the rollback have been logged. Thus, if the log size is large enough that all calls can be logged, then rollback is not required during recovery.

We now describe below the recovery of $P_i$ when it fails in the EXECUTING or the SUSPENDED state.

## D. Recovery of $P_i$

If $P_i$ was EXECUTING when it failed, RI$(P_i)$ initiates the rollback activity (see previous section) using CALL_BCK$(P_{i-1}, P_i)$, and then executes the INIT activity. If both the activities complete successfully, $P_{i-1}$ restarts the execution of $P_i$ [c.f. section II-C1)]. If the rollback completes successfully but the INIT is unsuccessful, $P_{i-1}$ fails the call on $P_i$ with the CALL_FAIL error message. If the rollback fails (on arrival of the RL_BCK_FAIL error message from at least one of the procedures to be rolled back), $P_{i-1}$ fails the call with the CALL_INCONSISTENT error message.

If $P_i$ was SUSPENDED when it failed, the callee of $P_i$ (i.e., $P_{i+1}$) is an orphan, so the recovery should handle the orphan as described below:

*1) Adoption of* $P_{i+1}$: The orphan adoption algorithm first determines if an orphan is adoptable, i.e., if its continued existence in the system does not interfere with the recovering procedure. For this, the orphan $P_{i+1}$ executes a *brake* algorithm: if $P_{i+1}$ finds that the execution of the orphaned thread will interfere with the recovering thread (e.g., both the threads may try to acquire a lock on a shared variable), a BRAKE message is sent down the orphan chain $(P_{i+1}, P_{i+2}, \cdots)$ to suspend the tip of the orphaned thread; otherwise, the orphan continues. On successful completion of the brake algorithm, $P_i$ recovers and resorts to a *thread stitching* algorithm whereby the orphaned thread and the recovering thread are "stitched" together by sending an ADOPT message down the (erstwhile) orphan chain and resuming the suspended thread for normal execution. On the other hand, if the orphan is not adoptable even by suspending its thread, then the state of the environment is rolled back to provide the *CALL_FAIL* outcome.[4]

We now present the details of the adoption algorithm. The algorithm is recursively executed at the different $P_j$'s.

## E. Adoption Algorithm

$P_j$ maintains three Boolean variables. When true, the flags have the following meanings:

*brake_flag*$(P_j)$: A brake is set at $P_j$ whereby $P_j$ is not allowed to make a call on $P_{j+1}$ or return to $P_{j-1}$ until *brake_flag* is set to false.

[4] Such a rollback is quite infrequent. And it is not necessary if the *CALL_FAIL* outcome is not required.

*adoption_flag*$(P_j)$: Adoption is still to be completed at $P_j$. So, $P_j$ is not allowed to return to $P_{j-1}$.

*cum_clr_rl_flag*$(P_j)$: At least one of the callers $P_k(i \leq k \leq j - 1)$ up along the orphaned call chain should perform a rollback as part of the recovery.

Thus, when $P_j$ is an orphan, *brake_flag*$(P_j)$ and/or *adoption_flag*$(P_j)$ is true.

Let $P_j(j \geq i + 1)$ be a callee in the orphaned call chain. For $j = i + 1$, i.e., the first procedure in the orphaned chain, $P_{i+1}$ knows it is an orphan upon detecting the failure of $P_i$; for $j > i + 1$, $P_j$ knows it is an orphan upon receipt of the BRAKE message from its immediate caller $P_{j-1}$. In both cases, $P_j$ sets *brake_flag*$(P_j)$ and *adoption_flag*$(P_j)$ to true.

Consider $P_{i+1}$. If CALR_ENV_INTRCT$(P_{i+1})$ is false, $P_{i+1}$ continues (concurrently with the recovering $P_i$) irrespective of whether the call is idempotent or not, because the calls originating from the recovering $P_i$ between the start and the failure points will be replayed by $P_{i+1}$ from its event log, and hence $P_i$ does not interfere with $P_{i+1}$'s execution. In this case, *brake_flag* is set to false. If CALR_ENV_INTRCT$(P_{i+1})$ is true, $P_{i+1}$ sets *cum_clr_rl_flag*$(P_{i+1})$ = *CALR_RL_FLAG*$(P_{i+1})$. The rest of the algorithm applies to all the procedures in the orphaned call chain (i.e., $j \geq i + 1$).

*1) Braking Orphaned Thread:* $P_j$ piggybacks a bit given by

$$CUM\_FLAG = cum\_clr\_rl\_flag(P_j) \vee (last\_nI\_call(P_j, *X) \geq (K_s + 1))$$

on the BRAKE message to its callee $P_{j+1}$, where *last_nI_call*$(P_j, *X)$ is a function that operates on CALL_OWN$(P_j)$ and returns the global call id of the last nonidempotent call from $P_j$ on $*X$; $K_s$ is the size of the event log maintained by $*X$. On receiving the message, $P_{j+1}$ sets *cum_clr_rl_flag*$(P_{j+1})$ = *CUM_RL_FLAG*.

Consider the orphaned call on $P_j$. The call may be one of the following:

*a) Idempotent:* Suppose *cum_clr_rl_flag*$(P_j)$ is true, i.e., a rollback is required by at least one $P_k(i \leq k \leq j - 1)$ when the failed $P_i$ recovers. If $P_j$ is in the EXECUTING state, a BRAKE_ACK message is sent to $P_{j-1}$ indicating completion of the brake operation at $P_j$ and those down the call chain. If $P_j$ is in the SUSPENDED state, it sends a BRAKE message to the callee $P_{j+1}$ down the call chain. Suppose *cum_clr_rl_flag*$(P_j)$ is false. $P_j$ may continue to execute (concurrently with $P_k$) since the call is idempotent and there is no pending rollback that may interfere with the call. So, $P_j$ sets *brake_flag* to false and sends a BRAKE_ACK message to $P_{j-1}$.

*b) Nonidempotent:* If $P_j$ is EXECUTING, it sends a BRAKE_ACK message to $P_{j-1}$. If it is SUSPENDED, $P_j$ sends a BRAKE message to $P_{j+1}$.

Consider the arrival of the BRAKE_ACK message from $P_{j+1}$ at $P_j$. If the call on $P_j$ is idempotent, $P_j$ simply passes the message to $P_{j-1}$ up the call chain. If the call is nonidempotent and if CALL_OWN$(P_j)$ contains at least one returned entry, $P_j$ completes the rollback algorithm and sends the BRAKE_ACK message to $P_{j-1}$. At $P_{i+1}$, sending the BRAKE_ACK to $P_i$ completes the brake algorithm.

Upon completion of the brake algorithm, RI$(P_i)$ (i.e., $P_{i-1}$) performs the rollback algorithm using CALL_BCK$(P_{i-1}, P_i)$. If the rollback activity of either $P_{i+1}$ or $P_{i-1}$ fails as indicated by the arrival of the RL_BCK_FAIL message, $P_{i-1}$ fails the call on $P_i$ by returning the CALL_INCONSISTENT exception. If only the INIT activity fails, $P_{i-1}$ fails the call by returning the CALL_FAIL exception. If the INIT activity and both the rollback activities are successful, $P_{i-1}$ carries out the RESTART activity on $P_i$. When the (re)execution of $P_i$ falls through to the call that was orphaned, sending the call request amounts to sending an ADOPT message down the orphaned thread to "stitch" the latter with the recovering thread, as described below.

*2) Thread Stitching:* $P_j(j \geq i + 1)$, upon receipt of the ADOPT message, sets the *brake_flag*$(P_j)$ to false and resumes the orphan execution from the point where the brake was set earlier.

Consider $j = i + 1$. If CALR_ENV_INTRCT$(P_{i+1})$ is false, the AD_CON algorithm (given below) is executed to adopt the concurrently executing $P_{i+1}$. Otherwise, the following algorithm is executed. Since the algorithm applies to all procedures down the orphaned call chain, it is described for the general case, i.e., $j \geq i + 1$:

If $P_j$ is idempotent, the AD_CON algorithm is executed to adopt the concurrently executing $P_j$. Suppose $P_j$ is nonidempotent. If $P_j$ is EXECUTING, it sets *adoption_flag*$(P_j)$ to false and sends an ADOPT_ACK message up the call chain indicating completion of adoption at $P_j$; if $P_j$ is SUSPENDED, then when the (re)execution thread reaches the adoption point (i.e., where $P_j$ got suspended), an ADOPT message is sent to $P_{j+1}$.

Upon receipt of an ADOPT_ACK message, $P_j$ sets its *adoption_flag*$(P_j)$ to false and passes the message onto $P_{j-1}$ up the call chain. At $P_{i+1}$, sending the ADOPT_ACK to $P_i$ completes the adoption algorithm.

*3) AD_CON Algorithm:* As we saw earlier, the recovering caller $P_k(i \leq k \leq j - 1)$ may, under certain situations, execute concurrently with $P_j$. In such cases, *brake_flag*$(P_j)$ is false when the ADOPT message arrives and $P_j$ is allowed to make calls on $P_{j+1}$ but not return to $P_{j-1}$.

If $P_j$ completes its execution first, it awaits adoption by $P_k$ before returning the call. When $P_{j-1}$ subsequently calls $P_j$, the ADOPT message is sent to $P_j$, upon which, $P_j$ sets *adoption_flag*$(P_j)$ to false, and simply returns the already completed call piggybacking the ADOPT_ACK.[5] If $P_{j-1}$ sends the ADOPT message before $P_j$ completes execution, the ADOPT message is held until $P_j$ completes execution, upon which, the ADOPT_ACK is piggybacked on the call return and *adoption_flag* set to false.

We now provide a quantitative analysis of our failure recovery technique.

## VI. Analysis of the RPC Algorithm

We introduce two indexes to characterize the recovery activities carried out by the run-time system. The extent of

[5] $P_{j-1}$ is, however, unaware that the call has returned immediately, and has the illusion that the call went through a normal execution.

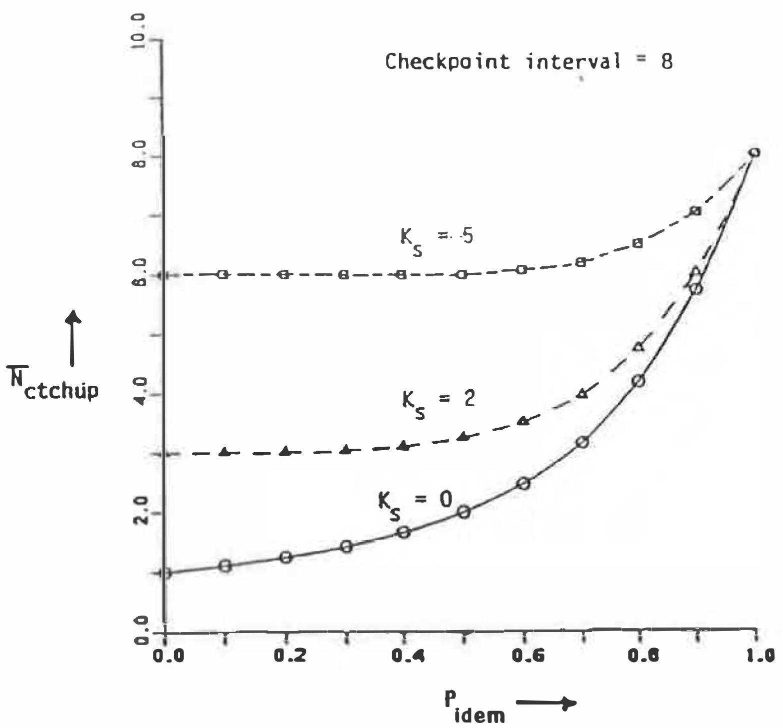


Fig. 4. Variation of catch up distance with respect to $P_{idem}$.

rollback required to recover from a failure is the criterion underscoring these indexes. The indexes guide a proper choice of the run-time parameters to minimize and/or eliminate rollback (and the associated rollback propagation).

### *A. Catch Up Distance*

A *catch up distance* is defined for a caller–callee pair. It is the maximum number of calls a caller may make to a callee such that if the caller fails and recovers, the callee need not be rolled back. The event log size $K_s$ at the callee and the application characteristics—measured in terms of $P_{idem}$, the probability that a call is idempotent—determine the size of the catch up distance for the caller–callee pair.

Let $TR^1, TR^2, \cdots, TR^i$ be a sequence of calls carried out by a caller on a callee ($TR^i$ is the last call in the sequence). Suppose the caller fails and recovers. The callee should rollback if the reexecution of $TR^1$ by the caller violates idempotency requirements. If, on the other hand, $TR^1$ can be reexecuted without rollback, then the entire sequence can be reexecuted without rollback.

Let $pR_i$ be the probability that the reexecution of $TR^1$ by the caller during recovery violates idempotency requirements. Then $pR_i$ is given by

$$pR_i = \begin{cases} 0 & \text{for } 1 \le i \le K_s \\ 1-(P_{idem})^i & \text{for } i = K_s+1 \\ (P_{idem})^{i-1} \cdot (1-P_{idem}) & \text{for } i \ge K_s+2. \end{cases}$$

The mean size of the catch up distance $\bar{N}_{ctchup}$, i.e., the mean number of calls that the caller may execute beyond which a failure will cause the callee to rollback, is given by

$$\bar{N}_{ctchup} = (K_s+1) \cdot (1-(P_{idem})^{K_s+1}) + \sum_{i=K_s+2}^{\infty} i \cdot (P_{idem})^{i-1} \cdot (1-P_{idem}).$$

$\bar{N}_{ctchup}$ is a static characterization of the program under the given run-time system. Fig. 4 shows the variation of $\bar{N}_{ctchup}$ with respect to $P_{idem}$ for a given $K_s$. This parameter lends insight into the choice of checkpoint intervals (the number of calls between two successive checkpoints) to effect recovery without rollback. Alternatively, it indicates the level of failure tolerance provided by the run-time system without a rollback, and hence may be used to determine the size of the event logs required to meet a desired level of failure tolerance. From Fig. 4, it is clear that the level of failure tolerance is higher when a server reexecutes calls (based on the idempotency properties) than when it does not.

### *B. Rollback Distance*

Rollback distance is the number of nonidempotent client calls after the last checkpoint (call return in our case) whose effects a callee should rollback when the client fails and recovers.[6] Assume $S$ calls have been completed by the client, and there is no on-going call. Suppose the client fails and then recovers. The probability that the rollback distance is $R(0 \le R \le (S-K_s))$ is given by

$$P_{rlbck,S}(R) = \binom{S-K_s}{R} \cdot (1-P_{idem})^R \cdot (P_{idem})^{S-K_s-R} \quad \text{for } S \ge (K_s+1).$$

[6] A nested rollback is considered as one rollback at the top level.

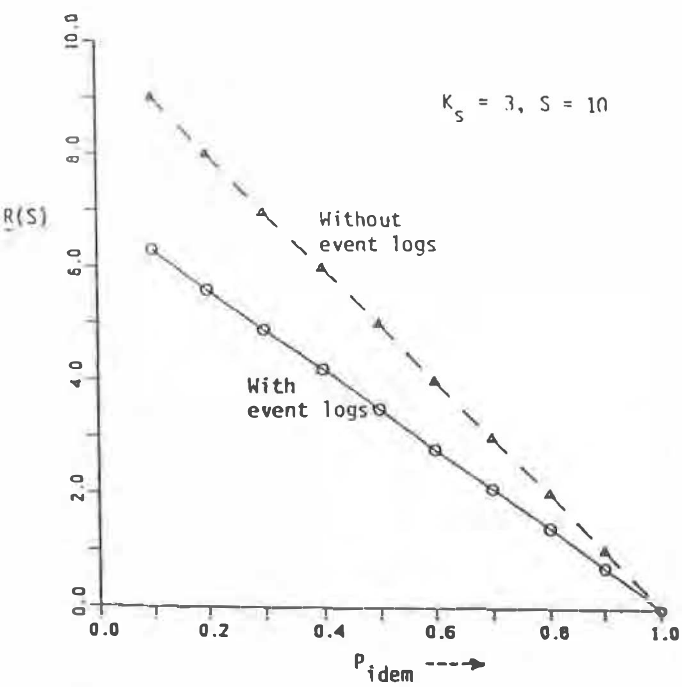

Fig. 5. Variation of rollback distance with respect to $P_{idem}$.

Note that $S$ is less than the checkpoint interval (in our case, the number of calls between call receipt and return). If $S < (K_s + 1)$, the question of rollback does not arise. The mean rollback distance is given by

$$R(S) = \sum_{R=0}^{S-K_s} R \cdot \binom{S-K_s}{R} \cdot (1-P_{\text{idem}})^R \cdot (P_{\text{idem}})^{S-K_s-R}.$$

The graphs in Fig. 5 illustrate the variation of $R(S)$ with respect to $P_{\text{idem}}$ for a given value of $S$. As can be seen, the effect of the event logs is to reduce the number of calls that have to be rolled back. A related index of interest is the probability that the callee should rollback, and is given by

$$(1 - P_{\text{rlbck},S}(0)) = \begin{cases} (1-(P_{\text{idem}})^{S-K_s}) & \text{for } S \geq (K_s+1) \\ 0 & \text{for } S \leq K_s \end{cases}.$$

When no logging is done, i.e., $K_s = 0$, the probability is $(1 - (P_{\text{idem}})^S)$. The graphs in Fig. 6 illustrate the variation of the probability of rollback with respect to $S$ for a given $P_{\text{idem}}$ and $K_s$. The effect of event logs in reducing the probability of rollback is more pronounced when $S$ is small. Thus, the farther (in terms of the number of remote calls) the failure point is from the last checkpoint, the less the advantages of event logs.

The rollback distance and the rollback probability constitute a dynamic characterization of the program since they depend also on the failure point given by $S$. These indexes lend insight into the extent of rollback required for given checkpoint intervals.

We now give some details of our prototype implementation along with indications about the performance of the orphan adoption technique.

## VII. Prototype Implementation

A prototype system based on the RPC model has been implemented on top of the V Kernel running on a network of SUN workstations interconnected by an Ethernet. The basic "send-receive-reply" style of message passing supported by the kernel is used as the message transport layer for the RPC model [11]. The system performs as expected under intentionally created machine and communication failure conditions. Two key aspects of the implementation are described here.

### *A. Information Flow Between Application and RPC Layers*

See Fig. 7. The exchange of application layer information with the RPC layer takes place through an interface consisting of a set of stub procedures. The stubs interface between a language level invocation of RPC and the underlying RPC layer [15]. A server makes static declarations about 1) the idempotency properties of the various operations it supports, and 2) the resource type (e.g., name binding information, leadership in a group). These declarations are used by a preprocessor for the language in which the server is implemented to generate the appropriate stubs. The stubs form part of the executable images of the client and the server.

At run-time, the RPC layer obtains the application layer information from the stubs and structures its internal algorithms and protocols described in the earlier sections. Communication exceptions are delivered to the stubs which then deal with the exceptions either by handlers built into the stubs or by user-supplied handlers hooked to the stubs.

### *B. Performance Indications*

Since the prototype implementation runs on top of another operating system and has not been optimized, we feel absolute timing of the various activities in RPC is not meaningful. Instead, we give an analysis of the communication overhead in terms of the number of process level messages, i.e., the number of messages exchanged by the communicating processes. The message size is usually 32 bytes long. When required to send information larger than 32 bytes in size, a segment containing up to 1024 bytes may be sent in one message.

*1) Sending Call Request and Call Return:* Refer to Fig. 3. Suppose $P_i$ makes a call on $P_{i+1}$. Sending the call request requires three messages: 1) a message from $P_i$ to $P_{i+1}$ containing the call request and the call arguments, 2) a message from $P_i$ to $P_{i-1}$ to checkpoint CALL_REF($P_i$, $P_{i+1}$) at $P_{i-1}$'s site, and 3) an acknowledgment message from $P_{i-1}$ to $P_i$. Returning the call requires three messages: 1) a message from $P_{i+1}$ to $P_i$ containing the results of the call and the thread state of $P_{i+1}$, 2) a message from $P_i$ to $P_{i-1}$ to delete the checkpointed CALL_REF($P_i$, $P_{i+1}$), and 3) an acknowledgment message from $P_{i-1}$ to $P_i$. In addition, the return of a nonidempotent call requires transfer of two types of information: CALL_OWN($P_{i+1}$) and $PV_{i+1}$. The message from $P_{i+1}$ to $P_i$ includes both CALL_OWN($P_{i+1}$) and $PV_{i+1}$. The message from $P_i$ to $P_{i-1}$ includes CALL_OWN($P_{i+1}$) (to checkpoint the list). Depending on size, various information may be transmitted in one or more segments.

For a connection-less call, one message is required for sending a call request and another for receiving the call return.

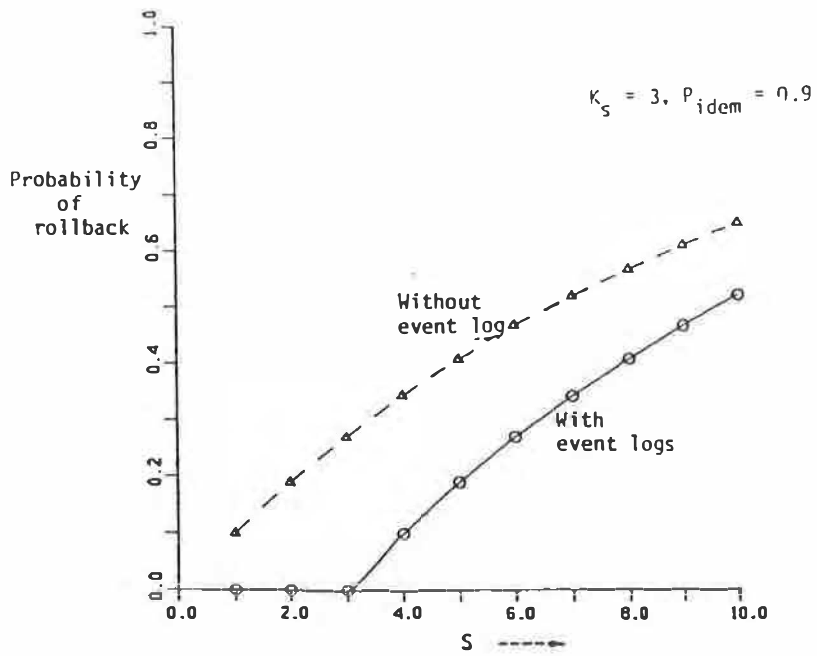


Fig. 6. Variation of probability of rollback with respect to $P_{idem}$.

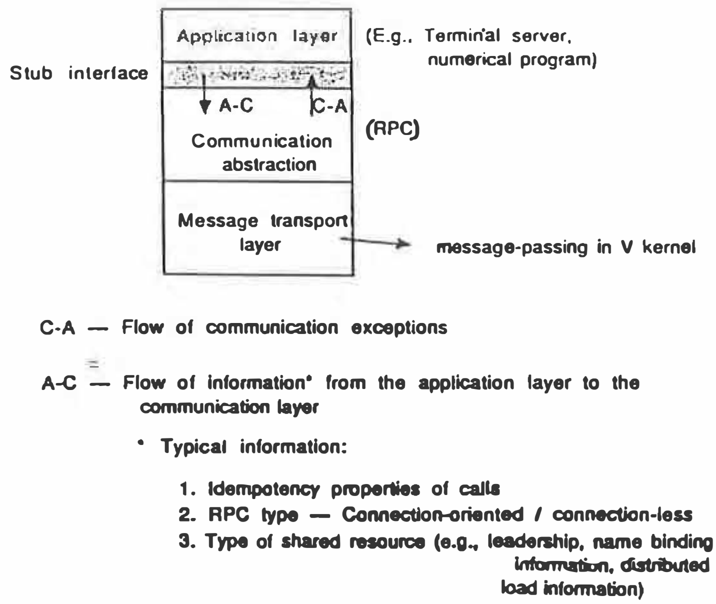


Fig. 7. Interface between application and communication layers.

In addition, a group message followed by one or more replies may be required to locate a server if the client's cache does not contain the name binding information for the server.

*2) Overhead in Failure Recovery:* Suppose $P_i$ fails. The messages required for failure recovery depend on the state of $P_i$ when it failed.

The messages required for the INIT activity are basically to locate a new server and initialize the server. Locating the server requires a group communication. The initialization requires transferring the CALL_THRD($P_{i-1}$, $P_i$) and HIST_BCK($P_{i-1}$, $P_i$) from $P_{i-1}$. The transfer requires two messages (in one or more segments). On completion of the recovery of $P_i$, two messages are required to notify the completion (one message for notification and the other for acknowledgment) to each of the procedures connected to $P_i$.

Suppose $P_i$ was IDLE when it failed, then the messages required for the INIT activity constitute the only overhead.

Suppose $P_i$ was EXECUTING when it failed. Then, in addition to the messages required for the INIT activity, the recovery requires messages for the transfer of CALL_BCK($P_{i-1}$, $P_i$) from $P_{i-1}$ and for any required rollback. For each element in CALL_OWN($P_i$), the rollback requires two messages.

Suppose $P_i$ was SUSPENDED when it failed. The brake algorithm requires two messages for each procedure in the orphan chain in addition to the messages required for any rollback initiated by the procedure. The thread stitching algorithm requires two messages for the procedure.

## VIII. Related Works

In this section, we compare our adoption technique to techniques proposed elsewhere and used in some experimental systems.

*ISIS:* In ISIS [2], one of the replicas of a server is designated to be the coordinator that executes client calls while the others act as cohorts. The coordinator periodically takes checkpoints at the cohorts, and retains the results (the *p_val's*) of all calls returned to the client since the last checkpoint. These results are used in forward failure recovery when the coordinator fails and a cohort takes over as the new coordinator and reissues the sequence of calls from the checkpoint. The technique implicitly assumes that all client–server calls are connection-oriented because only these calls may have the required connection descriptors to retain results of the calls. In other words, the descriptors (including retained results) should be maintained for every call irrespective of the operation it invokes. Our program model on the other hand is application-driven, and so encapsulates connection-less calls also. The recovery of such calls is simple in our technique—the calls are simply reexecuted. Second, it is not clear if ISIS deals with an on-going call thread that may be orphaned due to a failure. Our technique uses explicit algorithms to adopt the orphaned thread. Also, ISIS checkpoints the instruction pointer and the execution stack in addition to the application layer state. Our technique does not require these unless intermediate checkpoints are taken.

*DEMOS/MP:* In DEMOS/MP [7], checkpoints are periodically taken for every process at a central site. Also, every message received by a process since the last checkpoint is logged and the sequence number of the last message sent by the process to each of the other processes is recorded. If the process fails and recovers (from the last checkpoint), the logged messages are replayed to the process. Also, the kernel discards all the messages the process tries to (re)send up to the last message prior to failure. In effect, the process rolls forward to a consistent state without affecting the environment. The logging of messages is done at a low level (the central site monitors the broadcast network). The method requires logging of a large number of messages per process and regeneration of all low-level events when the process fails and recovers. Second, it requires a reliable broadcast bus because every message put on the bus (sent or received by a process) has to be logged by the central site. It is not clear how such a broadcast may efficiently be realized. Our technique, in contrast, is driven by application layer requirements, and works at a much higher level of abstraction.

*ARGUS:* ARGUS is a distributed programming language supporting *guardians* and *atomic actions* whereby client guardians can invoke atomic actions on server guardians [6]. The emphasis in ARGUS is to provide language level constructs to deal with failures. The RPC run-time system uses orphan killing based recovery to ensure call atomicity. Thus, the scope of our work as well as the underlying recovery technique are different from those of ARGUS.

*Lin's model of RPC:* Lin provides a model of RPC which ensures call atomicity by orphan killing and rollback [5]. Though his notion of atomic and nonatomic calls is similar to that of nonidempotent and idempotent calls, his program model does not support connection-less calls. Thus, our program model as well as the underlying recovery technique are different from those of Lin.

## IX. Conclusions

We have described a new model of RPC which systematically incorporates certain application layer properties and allows them to be exploited during failure recovery. The motivation for the model arises from our premise that many applications have an inherent ability to tolerate certain types of failures. The application layer failure tolerance capability is partly due to the evolution of many idempotent applications in large scale distributed systems. These applications do not require enforcement of ordering and atomicity constraints on the RPC events. The paper presents a wide range of examples to illustrate the effects of failures on various applications to support the premise.

Existing RPC models enforce the atomicity and ordering constraints on the events without regard to the application layer failure tolerance capability. So the algorithms used in the RPC layer to enforce the constraints are usually complex. Instead, the inherent failure tolerance capability of the application may be exploited to relax the constraints to simplify the algorithms in the RPC layer. Our RPC model provides a framework by which the application layer failure tolerance capability may be systematically exploited in failure recovery.

The model incorporates specific properties such as idempotency and connection-less calls. The properties allow a new type of failure recovery whereby orphans caused by failures during RPC are adopted rather than killed. The adoption technique minimizes the rollback which may be required in orphan killing techniques. Essential details of the technique are presented along with a quantitative analysis. A prototype of the model has been implemented on a network of SUN workstations interconnected by Ethernet.

The model is generic and simple, and is useful in distributed systems, particularly those which have complex failure modes (e.g., large and heterogeneous systems).

## References


[1] K. Ravindran, "Reliable client-server communication in distributed programs," Tech. Rep., Univ. of British Columbia, July '87.

[2] K. P. Birman *et al.*, "Implementing fault-tolerant distributed objects," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 502–508, June 1985.

[3] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, pp. 39–59, Feb. 1984.

[4] R. H. Campbell and B. Randell, "Error recovery in asynchronous systems," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 811–826, May 1986.

[5] K. J. Lin and J. D. Gannon, "Atomic remote procedure call," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 1121–1135, Oct. 1985.

[6] B. Liskov and R. Scheifler, "Guardians and actions: Linguistic support for robust distributed programs," *ACM Trans. Programming Languages Syst.*, vol. 5, pp. 381–404, July 1983.

[7] M. L. Powell and D. L. Presotto, "PUBLISHING: A reliable broadcast communication mechanism," in *Proc. 9th Symp. Oper. Syst. Principles*, ACM SIGOPS, June 1983, pp. 100–109.

[8] K. Ravindran and S. T. Chanson, "State inconsistency issues in local area network based distributed kernels," in *Proc. 5th Symp. Reliability Distrib. Software Database Syst.*, Jan. 1986, pp. 188-195.

[9] S. K. Shrivastava, "Treatment of orphans in a distributed systems," in *Proc. 3rd Symp. Reliability Distrib. Software Database Syst.*, Dec. 1983.

[10] SUN Network Services—System Administration for the SUN Workstation. Feb. 1986.

[11] D. R. Cheriton, "V-Kernel: A software base for distributed systems." *IEEE Software*, vol. 1, pp. 19–42, Apr. 1984.

[12] M. Herilihy and B. Liskov, "A value transmission method for abstract data types," *ACM Trans. Programming Languages Syst.*, vol. 4, pp. 527-551, Oct. 1982.

[13] K. Ravindran and S. T. Chanson, "Relaxed consistency: A basis for structuring interprocess communications in distributed server architectures," *IEEE Trans. Comput.*, to be published.

[14] S. T. Chanson and K. Ravindran, "Host identification in reliable distributed kernels," *Comput. Networks ISDN Syst.*, vol. 15, pp. 159-175, Aug. 1988.

[15] P. B. Gibbons, "A stub generator for multilanguage RPC in heterogeneous environments," *IEEE Trans. Software Eng.*, vol. SE-13, pp. 77-87, Jan. 1987.

[16] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Trans. Comput. Syst.*, vol. 1, pp. 222-238, Aug. 1983.

**K. Ravindran** (S'84–M'87) received the B.Eng. degree in electronics and communications engineering in 1976 and the M.Eng. degree in automation in 1978, both from the Indian Institute of Science, Bangalore.

Until 1982, he was a Systems Engineer at the ISRO Satellite Centre, Bangalore, working primarily on computer simulation techniques for real-time systems. He received the Ph.D. degree in computer science from the University of British Columbia, Vancouver, B.C., Canada in 1987 in the areas of Distributed Operating Systems. Until July 1988, he was an Assistant Professor in the Department of Computer Science and Automation at the Indian Institute of Science. Currently, he is a Research Scientist with the Bell Northern Research, Canada pursuing exploratory work on broad-band ISDN architectures and protocols. His research interests include design and modeling of distributed systems, distributed programming languages, architectures and protocols for LAN's and ISDN's, computer architectures, software engineering and real-time systems.

Dr. Ravindran is a member of the Association for Computing Machinery and the IEEE Computer Society.

**Samuel T. Chanson** (M'76) received the Ph.D. degree in electrical engineering and computer sciences from the University of California, Berkeley, in 1974.

He was an Assistant Professor at the School of Electrical Engineering, Computer Division at Purdue University for two years before joining the Department of Computer Science at the University of British Columbia where he is an Associate Professor and a founding member and director of its Distributed Systems Research Group. He has been actively doing research in distributed operating systems, computer communications, and performance analysis of distributed systems.

Dr. Chanson has served on program committees of international conferences and workshops on distributed operating systems and communication protocols. He organized and co-chaired the first International Workshop on Protocol Test Systems held in Vancouver, B.C., Canada, in October 1988.