Application-driven Failure Semantics of Interprocess Communication in Distributed Programs

K. Ravindran*
Department of Computer Science & Automation,
Indian Institute of Science,
Bangalore - 560 012, India.

Samuel T. Chanson,
Department of Computer Science,
University of British Columbia,
Vancouver B.C. V6T 1W5, Canada.

K. K. Ramakrishnan,
Distributed Systems Architecture and Performance,
Digital Equipment Corporation,
Littleton MA 01460, USA.

25 June 1988

Abstract

Distributed systems are often modelled after the client-server paradigm where resources are managed by servers, and clients communicate with servers for operations on the resources. These client-server communications fall into two categories — connection-oriented and connection-less, depending on whether the servers maintain state information about the clients or not. Additionally, each of the servers may itself be distributed, i.e., structured as a group of identical processes; these processes communicate with one another to manage shared resources (intra-server communications). Thus, the activities of a distributed program may be viewed as a sequence of client-server communications interspersed with intra-server communications. In this paper, we identify suitable interprocess communication (IPC) abstractions for such communications — remote procedure calls for client-server communications and *application-driven shared variables* (a shared memory-like abstraction) for intra-server group communications. We specify the properties of these abstractions to handle partial failures that may occur during program execution. The issues of orphans and consistency arising due to partial failures are examined, and solution techniques specified as part of the run-time system. The abstractions allow certain relevant information of the application to be incorporated in the run-time system. Examples are given to illustrate the use of these abstractions as primitives for constructing distributed programs.

Key words:

Client-server model, remote procedure call, shared variable, partial failures, orphan, group communication

---

*Work was partly performed when the author was with the Department of Computer Science, University of British Columbia. The author is currently with Bell Northern Research, Ottawa, Canada.

1 Introduction

The client-server model of interprocess communication (IPC) is a very useful paradigm for structuring communication between the various processes in a distributed system. In this model, the processes that implement and manage resources (also referred to as services) are called *servers* and the processes that access these resources are called *clients*. The clients and the servers may possibly reside on different machines. A server exports an abstract view of the resource it manages with a set of allowable operations on it. A client communicates a request to the server for operations on the resource, and the server communicates the outcome of the operations to the client by a response. This *request-response* style of communication is fundamental to client-server communications (also referred to as client-server interactions) [24,10].

In contemporary distributed systems, a service may itself be distributed, i.e., provided by a group of identical server processes executing on different machines, with functions replicated and distributed among the various processes to enhance failure tolerance and availability. Clients access the service as a unified logical entity across a well-defined interface. An example is a distributed file service shared by a cluster of workstations across a local network. The service may consist of a group of server processes each one managing a subset of the file name space. Clients access the file service with a unified set of primitives in a network transparent fashion. If a process in the group fails, only the files managed by the failed process will be unavailable to clients. To encompass such an architecture, we extend the client-server model as follows: In general, the management of a distributed service underscores some form of resource sharing among the processes of the service; the sharing manifests in the form of communication among the processes, which we refer to as *intra-server communication*. The communication exhibits a *contention style* whereby the processes contend among themselves to access the resource and coordinate the sharing.

Communication abstractions are required to map the client-server and the intra-server communications in distributed programs on such systems. The abstractions reside on top of a primitive message-passing IPC substrate and interface to the clients and servers which constitute the application layer. Usually, two types of IPC mechanisms are provided by the message-passing substrate—one-to-one and one-to-many. The client-server and intra-server communications are structured on top of this substrate. Viewing from the application side, though the basic message-passing substrate is sufficient for some classes of client-server and intra-server communications, semantically it is not powerful enough to handle the consistency issues with regard to the distributed state maintained in the client-server interface. Higher level abstractions are needed to provide a stronger form of IPC for reliable communications among the processes of the program.

The theme of this paper is to identify suitable communication abstractions for use in distributed programs,

and to specify their abstract properties to handle process, machine and communication failures that may occur during the IPCs. The paper uses a new *application-driven* approach in specifying the properties of the abstractions. The approach allows certain relevant information of the application to be incorporated in the client-server interface, thereby simplifying the underlying failure recovery algorithms.

The paper is organised as follows: Section 2 characterizes the client-server and intra-server communications in a distributed program. Section 3 identifies two high level IPC abstractions mapping the different characteristics of these communications: (1) Remote Procedure Call (RPC) for structuring client-server communications, and (2) *application-driven shared variables* (ADSV) for structuring intra-server communications. Sections 4 and 5 specify the properties of the abstractions with respect to failures. Using these IPC abstractions, the client process of a distributed program may interface to operating system services in a manner similar to that offered in single machine systems. While specifying the properties of the abstractions, we have largely refrained from specific implementation details. Sections 6 and 7 show how the abstractions may be used as primitives for reliable distributed programming.

2 Application-driven model of a distributed program

As described in section 1, server processes implement resources and client processes communicate with the servers to access the resources. Such a request-response communication is a common communication style in distributed programs. Additionally, a service may be provided by a group of server processes. We organize such server processes into a process group [11,1], referred to as a *server group*, to manage the resource. In general, the member processes (or simply members) of a server group share one or more abstract resources and contend among themselves to provide a unified interface to the clients. The intra-server communication in the service may take place by one-to-many (group) communication among the members of the server group, i.e., the sender of a group message and the recepients of the message belong to the same group. The server group is specified by the pair (*rsrc_nm*, *srvr_gid*), where *rsrc_nm* is the name of the resource and *srvr_gid* is the identifier (id) of the process group.

Examples: A file server group is a process group *file_srvr_gid* that provides file service (referred to as FILE). Thus, for example, it manages the global name space of files in the system with each member of the group implementing a subset of the name space. The server group is identified by the pair (*FILE*, *file_srvr_gid*). A spooler group is a process group *prnt_gid* that provides print spooling service, referred to as SPOOL, with each member of the group serving a subset of the clients. The spooler group is identified by the pair (*SPOOL*, *prnt_gid*).

The intra-server group communication initiated by a server is orthogonal to the communication between the server and its clients. Thus, a distributed program may be structured as a sequence of client-server communications interspersed with intra-server 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 (see Figure 1). Each of the two communication styles is further discussed in the following subsections.

2.1 Client-server interactions

Client-server interactions may be of two types — *connection-oriented* and *connection-less* — as described below:

A client-server interaction is connection-oriented if in a seqnence of such interactions, the server maintains certain ordering relationship among them. The interaction may cause permanent changes to the resource the server exports to the client. State information about the resource and the client is maintained in the server across interactions throughout the duration of the connection. The information is used by the server to maintain the required ordering relationship among the interactions, and to protect the resource against inconsistencies caused by client failures. An example of a connection-oriented interaction is a client operating on a file maintained by a file server; part of the state[1] maintained by the server is the seek pointer. As another example, consider the static variables supported in a distributed implementation of the 'C' language. A server implements a procedure and maintains the static variables, while a client implements a calling procedure and interacts with the server over a connection to operate on the variables.

A client-server interaction is connection-less if in a sequence of such interactions, the server need not maintain *any* ordering relationship among them. This implicitly assumes that the interaction 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 relating to a connection-less interaction with the client during or past the interaction. Examples of connection-less interactions are a client requesting i) time information from a time server, and ii) a numerical computation from a math library server.

Because of their inherent characteristics, connection-less interactions are *light-weight* — the algorithms to implement them may be simpler and more efficient — as compared to connection-oriented interactions. The failure recovery component of the algorithms may also be simpler (section 6).

[1] Here, 'state' refers to high level, resource-dependent information.

2.2 Intra-server group interactions

Members of a server group manage one or more shared resources and interact among themselves to provide a uniform interface to clients. Such interactions exhibit a contention style of communication whereby members contend with one another to access the resources. This style of communication is different from the request-response style in client-server interactions, and hence requires a different type of communication abstraction.

The shared resources are characterized by ***distributed state variables*** maintained by the group. Let V be such a state variable maintained by a server group S_G (see Figure 2). V may assume a set of values which are dependent on the resource abstracted by V. It may be updated during an interaction between the client and S_G, or an intra-server interaction within S_G. Examples of V are the name binding information maintained by a name server group, the lock variable maintained by a spooler group managing a shared printer, the leadership within a server group, distributed lists containing information about membership in a group, distributed load information, and the alive status of machines.

Let $v_1, v_2, \ldots, v_N$ be the instances of V maintained by the members $s_{g_1}, s_{g_2}, \ldots, s_{g_N}$ (of S_G) respectively, and let v_c be the instance of V maintained by the client. The values assumed by these instances of V may be inconsistent with one another due to partial failures and due to the asynchronous nature of the intra-server interactions. Our premise is that these inconsistencies may be tolerated to some extent depending on the resource V abstracts (i.e., it is not necessary to ensure the instances are consistent at all times, see section 7), and that the inconsistencies may be handled by the IPC abstraction that governs access operations on V.

2.3 State transitions in the program

The state of a distributed program is given by the set of states of all the processes in the program. Thus an interaction between a client and a server may cause the program state to change if the state of the client or the server changes during the interaction. A client-server interaction TR is denoted by

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

where C_{bef} and C_{aft} are the states of the client before and after the execution of TR, and S_{bef} and S_{aft} are the corresponding states of the server. S_{aft} depends on (S_{bef}, TR) and C_{aft} depends on (C_{bef}, TR, p_val) where *p_val* is a value emitted by the server in state S_{bef} in response to TR. Thus TR causes the server to emit a value *p_val* and change its state to S_{aft}, and the client to accept *p_val* and change its state to C_{aft}. Suppose TR is a connection-less interaction, then since the server does not maintain any state information,

TR is simply represented by

$$(C_{bef}) \stackrel{TR}{\rightarrow} (C_{aft}).$$

In a client-server interaction, the server may change its state in the following ways: (i) Interactions as a client with other servers; the state transition is caused by the returned values *p_val*. (ii) Intra-server group interactions when a shared resource is manipulated. We examine each in detail below:

2.3.1 Returned value

The *p_val* may be abstracted as a set of (attribute, value) pairs. An attribute is a name used by the client to specify an operation on the server, and the server may return one of many possible values for the attribute. The (attribute, value) pairs are defined for the operation by the application layer. They are specified in the request and the response messages exchanged between the client and the server to transport TR. As an example, suppose TR is a request to a file server to open a file. Two attributes FILE_LOOK_UP and ALLOCATE_RESOURCE may be specified in TR for a look up operation on the file and allocation of resources for the file respectively. Let the possible return values for the attribute FILE_LOOK_UP be { FILE_FOUND, FILE_NOT_FOUND }, and that for the attribute ALLOCATE_RESOURCE be { RESOURCE_ALLOCATED, RESOURCE_UNAVAILABLE }. Then one possible return value for TR is

$$p\_val = \left\{ \begin{array}{l} (FILE\_LOOK\_UP, FILE\_FOUND), \\ (ALLOCATE\_RESOURCE, RESOURCE\_UNAVAILABLE) \end{array} \right\}.$$

Such a characterization based on attribute names and values is useful in specifying the semantics of client-server communication in general (section 7.1).

The client level interpretation of the outcome *p_val* as a successful completion of TR or otherwise depends on the application. In the example given above, suppose the file server is unable to locate a file (FILE_NOT_FOUND) under a given name in response to a search request from a client. The fact that the search failed may be considered by the client to be successful if the search is a prelude to creating a new file under the name, or unsuccessful if the search is a prelude to opening the file. Such client level interpretations of non-success constitute ***application-level failures*** which may be present even with a fully reliable communication layer. Thus one should distinguish between partial failures and application-level failures. The latter are application-dependent and are outside the scope of the paper.

The *p_val* may also be used to specify the determinism properties of the program, namely a server generates the same *p_val* whenever the server executes the same call under identical conditions (see [1] for details). For simplicity and without loss of generality, we assume only deterministic programs in the model.

2.3.2 Operations on shared resources

Suppose the server is a member of a server group. The server may change the local instance of the state variables it maintains by its interactions with other members in the group. The server may initiate the interactions when it tries to access the resource shared among the members, or it may participate in the interactions initiated by other members.

Based on the above discussion of how a server may change state, we describe in the next subsection the *idempotency* property of client-server interactions which is useful in the failure recovery algorithms.

2.4 Idempotency

Consider a client-server interaction TR, as given by the relation (1)

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

The idempotency property of TR [23,24] relates to the effect of TR on the state maintained by the server, 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, $S_{aft} = S_{bef}$; however, C_{aft} need not be the same as C_{bef} since the client may change its state due to the *p_val* returned from the server. Examples of idempotent calls are read operations on a file which do not change the seek pointer, time requests to a time server group and file search requests to a file server group[2]. If TR is a non-idempotent call, then S_{aft} may be different from S_{bef}. Examples of non-idempotent calls are relative seeks on a file and opening a file.

To examine the usefulness of the idempotency property in the recovery algorithms, we introduce the concept of *re-execution* of TR.

2.4.1 Re-execution

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

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

The concept of call re-execution is useful in the forward recovery scheme described in section 6. It is also useful in dealing with *message orphans*, i.e., multiple executions of a server caused by re-transmissions of a

[2]The latter two examples are of the connection-less type, and hence are always idempotent because the server does not maintain any state.

call request message from a client (see section 6.1).

In order for a re-execution to be useful, TR should be idempotent. It follows from the definition of idempotent calls that if TR (and therefore TR^n) is idempotent, then $S_{aft^n} = S_{aft} = S_{bef}$. In other words, the server state does not change under re-executions of an idempotent call. Also, since TR is deterministic, $C_{aft^n} = C_{aft}$. If TR is non-idempotent, then S_{aft^n}, S_{aft} and S_{bef} may be different; also, C_{aft^n} and C_{aft} may be different.

Based on the above concept of re-execution, 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 re-executions of TR. An example is an absolute seek operation on a file.

3 Communication abstractions

Having characterized the communication styles in a distributed program from an application point of view, we now identify suitable communication abstractions which map naturally onto these styles. Failure semantics will be discussed in the context of these abstractions in the subsequent sections.

3.1 Remote procedure call

RPC is a high level communication abstraction by which a client may interact with a server on a different machine [5]. It is a widely accepted abstraction for building distributed programs because it encapsulates the procedure call mechanism that is common and easily understood in programming, and allows a programmer to access remote processes and system services much the same way as local processes.

Refer to Figure 3. 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,... i, i+1) are said to 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.

Though RPC maps well onto client-server interactions, it is not adequate for intra-server interactions because the latter exhibit a contention style of communication (for accessing shared resources) that is different from the request-response style of communication supported by RPC. Though in some cases, access to a resource by a server may be triggered by RPC from a client, the server still needs to contend with other

members to access the resource. In some other cases, contentions by the server to access a resource may exist independently of any client interactions with the server. For the above reasons, we introduce another abstraction in the next subsection which may be used either in conjunction with RPC or independently for access to shared resources.

3.2 Application-driven shared variables

A high level abstraction that maps well onto the intra-server group interactions is shared-memory because i) the interactions primarily deal with the distributed state variables shared among the server group members, and ii) IPC by shared memory is a well-understood paradigm in centralized systems. The abstraction presents a memory (i.e., a state variable) that is logically shared among the members. We refer to such a logical memory as *application-driven shared variable* (ADSV) because, as we shall see later, the consistency requirements of the variable are specified by the application. Conceptually, ADSV is similar to physical shared memory, and is an abstract container of the instances of V (see Figure 2) distributed across the members of the server group. Conventional operations for reading, writing, locking and unlocking physical memory are definable for the ADSV as well, so the members may use these operations to interact with one another to operate on a shared resource. However, the procedural realization of the operations should handle the underlying consistency issues.

3.2.1 Operations on ADSV

The operations on the ADSV may be realized by using group communication across the server group members because group IPC lends itself well for a member to interact with other members of the group conveniently and efficiently through a single message (the process group mechanism allows processes to form groups, join and leave them [11]). In such a realization, a shared variable V may be associated with the group id of the server group whose members share the variable. The details of the protocols used to implement the operations may be found in [1]. We now identify the basic operations:

status = Create_Instance(V). The operation creates an instance of the variable V for the requestor so that the latter may perform a series of operations on V. Procedurally, a server process joins the server group and acquires the state of the shared resource.

val = Read(V). The operation returns the value of the variable V in *val*. Note that this operation (and the write operation given below) and the interpretation of *val* are application-dependent. Procedurally,

the member reads the local instance of V; the member may also interact with other members of the group to correct its local instance if necessary.

Write(V, val). The operation writes the value *val* into the variable V. Procedurally, the member writes into its local instance of V, and may also communicate with other members of the group to update their instances of V.

status = Lock(V). The operation locks the variable V for exclusive access by the requestor. The operation succeeds immediately if V is readily allocatable (typically, if V is not already locked); otherwise it is queued up waiting for V to become allocatable (typically, when the current holder of V either releases V or suffers a failure). Once allocated, the requestor has exclusive possession of V until the lock is released. In the realization of this operation, the member may interact with the group to resolve any simultaneous attempts to lock V (the arbitration mechanism is unspecified).

Status = Unlock(V). The operation unlocks the variable V from the exclusive possession of the requestor. If there are queued lock requests on V, some form of arbitration mechanism is employed to allocate V to one of the waiting requestors. Procedurally, the member may send a group message advising release of the lock on V.

Status = Delete_Instance(V). The operation deletes the instance of the variable V created by the requestor. Procedurally, the member may leave the group. If it has locked V, i.e., holds any shared resource, it should send a group message advising return of the resource.

Since the arbitration mechanisms in the **Lock** and the **Unlock** operations do not guarantee any specific ordering among the operations, they are semantically weaker than the P and V operations on semaphores where a well-defined arbitration order (such as FIFO, LIFO) is usually specified. As we shall show later, the weaker semantics is sufficient at this level of abstraction for many applications. Specific ordering required by high level algorithms, say for concurrency control and serialization of access to the resource, may be structured using these operations.

We now specify the failure semantics of the RPC and the ADSV. The semantics allow design of the failure handling algorithms and protocols in the later sections.

4 Failure semantics of RPC

Refer to Figure 4. Let P_{i-1} (itself the callee of P_{i-2}) make a call (TR) on P_i. As the call thread executes P_i, it may visit the various servers $P_{i+1}, P_{x1}, P_{x2}, \ldots$ through a series of calls causing the servers to change

states. 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 or abnormally. The abnormal return may occur when P_i fails or when there are communication failures between P_i and P_{i-1}. TR is considered to have succeeded if the thread makes a normal return, failed otherwise.

A desired failure semantics of the call TR is as follows: Suppose $\underline{X}$ is the state of the environment when the call is initiated. If the call succeeds, P_{i-1} should see the final state of the environment $\underline{Y}$; otherwise, P_{i-1} should see the initial state $\underline{X}$. These two outcomes are represented as:

$$CALL\_SUCC(TR) \equiv (\underline{X}, \underline{Y}), \quad \text{and}$$
$$CALL\_FAIL(TR) \equiv (\underline{X}, \underline{X}) \tag{2}$$

where $(\underline{X}, \underline{Y})$ indicates a state transition from $\underline{X}$ to $\underline{Y}$. The RPC run-time system exposes these outcomes to the caller, abstracting (denoted by '$\equiv$') the underlying state transitions. The semantics underscores the notion of the ***recoverability*** of the call, an important property for the call to be atomic [16]. It means that the overall effect of the call should be all-or-nothing.

Suppose when P_{i-1} initiates the call TR on P_i, the state of the environment is $\underline{X}$. Suppose also that during the execution of TR, P_i initiates a call on P_{i+1} and then fails. Let $\underline{X}'$ be the state of the environment when P_i failed. The failure of P_i is considered to have been masked from its communicants P_{i-1} and P_{i+1} if the run-time system is able to recover from the failure and provide the outcome $CALL\_SUCC(TR)$ to P_{i-1}. 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 $\underline{X}'$ to $\underline{Y}$. If the run-time system is unable to mask the failure, say due to the unavailability of such an identical process, then the failure semantics requires that P_{i-1} sees the outcome $CALL\_FAIL(TR)$.

The failure semantics implies a failure recovery that allows delivery of the outcome $CALL\_SUCC(TR)$ or $CALL\_FAIL(TR)$ as the case may be to P_{i-1}. If TR is connection-less, the semantics is still applicable, but requires no recovery because P_i does not maintain any state.

4.1 Rollback and *CALL_FAIL* outcome

Consider the failure scenario described in the previous section namely that P_i, during the execution of TR initiated from P_{i-1}, fails after initiating a call on P_{i+1}. The portion of the thread at P_{i+1} down the call chain is an orphan while that at P_{i-1} up the call chain is an uprooted call.

Suppose the RPC run-time system is unable to mask the failure of P_i, then the run-time system rolls back the state of the environment from X' to X to provide the required $CALL\_FAIL(TR)$ outcome. This requires, among other things, killing the orphan P_{i+1} [15,19]. In general, if the failure of a process cannot be recovered, it may be necessary to rollback all servers to their states prior to the initiation of the orphaned thread that visited the servers. Thus, to provide the $CALL\_FAIL$ outcome, the orphan should be detected and killed [5,23]. This amounts to destroying the execution of the orphan and undoing its effects (rollback).

For connection-less calls, the requirement for such a rollback does not exist as far as the failure semantics is concerned. However, killing the orphans may still be desirable since they waste system resources.

4.2 Unrecoverable calls

Assume the RPC run-time system encapsulates algorithms and protocols to support rollback and provide the outcome $CALL\_FAIL$. Even so, rollback may not be possible in many situations, particularly i/o operations that affect the external environment (e.g., human user or a peripheral device). In some applications such as print operations, rollback may not be meaningful [18]. 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. The calls that so affect the external environment are unrecoverable when a failure occurs. The outcome of such unrecoverable calls is referred to as CALL_INCONSISTENT indicating to the caller that the state of the environment may be inconsistent.

The $CALL\_FAIL(TR)$ and $CALL\_INCONSISTENT(TR)$ outcomes of TR, when they occur, are delivered to the caller as 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 a client of a time server periodically calls the server to obtain time information and update its local time. If a call TR fails with the $CALL\_FAIL(TR)$ outcome, the client may deal with the exception by tolerating the failure and hoping to correct the time at the next call. Details of how the exceptions are communicated to the application layer and their effects in the presence of concurrent calls may be found in [1,21].

5 Failure semantics of ADSV

Recall that the concept of ADSV is used in the context of a server group, so the failure semantics of ADSV is specific to the group. The implications of the failure of a group member 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 should the member fail [1]. For certain resources, the lock recovery becomes part of a rollback activity that may be initiated by the member if its client fails (refer to section 4.1). 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.

Suppose in another case, the group maintains a ranking among its members. Each member occupies a position in the ranking and has a view about the positions occupied by other members in the ranking. This view constitutes the shared variable of the group. If it is required that all members have a consistent view of the ranking, then the failures of members should be observed in the same order by the (surviving) members of the group.

Thus the atomicity and the ordering constraints on the failure events are application-dependent.

The communication layer obtains such application layer information during run-time and structures its internal algorithms and protocols accordingly as outlined in sections 6 and 7 below.

6 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 standing by reconfigures to continue the server execution from the point of failure. Such a *re-incarnated* process *adopts* the orphan caused by the failure, i.e., the orphan is allowed a *controlled survival* rather than being killed and causing rollback of the server state. The adoption may occur when the re-incarnated process re-executes 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 re-execution, the re-incarnated process (re-)issues the various calls embedded in the call thread from the restart to the adoption points. However, the re-executions by the various servers due to such calls should cause no effect on the environment (see sections 2.4.1 and 6.1). The re-execution of the re-incarnated process allows it to *roll forward*, and may be categorized as a forward error recovery technique [6]. The technique minimizes rollback which may otherwise be required in orphan killing techniques.

Refer to Figure 4. Consider the failure scenario described earlier in section 4, namely that P_i, during the execution of TR initiated from P_{i-1}, fails after initiating a call on P_{i+1} — $\underline{X}$ and $\underline{X}'$ are the state of the environment when TR was initiated and when P_i failed respectively. Suppose P_i re-incarnates at the point where it was initially called and rolls forward. If the orphan P_{i+1} is adopted by the re-incarnated P_i, then P_{i+1}

can make a normal return of the call 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* ($\equiv (X, X')$) is delivered.

The run-time system effects a roll forward based on two mechanisms (refer to Figure 5):

1. Controlled re-execution of the calls, if necessary, based on their idempotency properties.
2. Event logs that contain call completion events allow a recovering process to get in step and become consistent with other processes without actually re-executing the calls (see section 6.2).

These are examined in the following subsections:

6.1 Re-execution of call sequences

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

$$(C_{i-1}, S_{i-1}) \stackrel{TR^i}{\rightarrow} (C_i, S_i), \tag{3}$$

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 re-execution of TR^i, represented as $TR^{i'}$, after the server has executed TR^k; the request for $TR^{i'}$ may be either a message orphan or from a recovering client. The call sequences EV_SEQ and $[EV\_SEQ \succ TR^{i'}]$ are not ordered sequences with respect to one another. Thus call re-executions by the server often requires the relaxation of ordering constraints on calls without affecting the consistency of the server state. The re-executions of a call underscore the idempotency property associated with the call (c.f. section 2.4) as described below:

6.1.1 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 re-execution $TR^{i'}$ (i.e., $TR^k \succ TR^{i'}$) invoked by a re-incarnated client may interfere with the calls in EV_SEQ which the server had already completed. $TR^{i'}$ does not interfere with TR^k if

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

Thus, the necessary condition for the server to execute $TR^{i'}$ without causing state inconsistency between the re-incarnated client and the original instance of the client is that $TR^{i'}$ should be idempotent. However, it is not a sufficient condition. The necessary and sufficient condition for such a consistency is given by the requirements (see relation (3)) that

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

Because the program is deterministic, the first requirement is satisfied. Thus the effect of $TR^{i'}$ 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 globally true if the condition

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

is satisfied. This is possible only if $TR^i, TR^{i+1}, \ldots, TR^k$ are all idempotent calls. The condition specifies, in general, when the server may re-execute a call without causing inconsistencies. If TR^i is a 1-idempotent call, then $TR^{i'}$ can be re-executed 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, then $EV\_SEQ \succ [TR^{i'}]$ is an idempotent sequence. The analysis also lends insight into other commutative properties of calls such as: (i) ordering of calls by the server (e.g., generation of serializable schedules for incoming calls), (ii) interspersing of calls from multiple clients on the server — even though a client may issue a sequence of idempotent calls, if there is at least one non-idempotent call from other clients interspersed into the sequence, the client perceives the effect of a non-idempotent call, and (iii) connection-less calls can be interspersed into any serializable schedule.

6.2 Event logs

An event log is used to record an event that happened at some point in time so that the event can be replayed at a later time. We use the replay technique for connection-oriented calls (without re-executing the calls) during forward recovery. When a server completes a call, it logs the call completion event in a linked list. The completion event is 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 actually

(re-)executing it. Thus, if TR^i is a call represented by relation (3):

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

then a replay E^i from the event log for TR^i may be represented as

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

for some TR^j such that $TR^i \succ TR^j \succ E^i$. In other words, an event log allows a re-incarnated client to roll forward to a consistent state without violating the call idempotency requirements.

If a call from a recovering client cannot be completed either from the event log or by re-execution, the call fails with the CALL_INCONSISTENT outcome. The details of the algorithms and protocols used by the run-time system to realize orphan adoption may be found in [1].

6.3 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 recovers the lock on the resource and forces a CALL_FAIL or CALL_INCONSISTENT exception, as the case may be, to the client of the failed process.

6.4 Connection-less calls

Since connection-less calls on a server do not require any form of ordering among them, the calls are neither assigned call identifiers nor logged by the server. So these calls when re-issued by the new P_i are invariably re-executed by the server. Also if a server fails during a connection-less call on it, the client simply re-issues the call on another replica of the server. Since our program model encapsulates connection-less calls, such recoveries are required in the underlying algorithm. In this aspect, the algorithm is distinct from that used elsewhere [3,20,17].

7 Relaxing the consistency requirements on ADSV

As described in the earlier sections, RPC maps onto the request-response style in client-server communications while ADSV maps onto the contention style in intra-server group communications. Partial failures during a group communication may not affect the outcome of the communication under certain situations, and may

be difficult to be distinguished from application-level failures (see section 2.3). Hence the ADSV abstraction should deal with the problem of inconsistency among the various instances of a shared variable.

One way to ensure consistency of the shared resource is to provide atomicity, order and causality properties in the group communication layer, such as the group communication primitives proposed by Birman [4] and Cristian [12]. Such primitives may be used to perform atomic and/or ordered operations on the resource. However, distributed server interfaces typically do not require absolute consistency with the attendant penalty in higher overhead. Occasional inconsistencies may be detected at higher level (which may possibly include the user) when the resource is accessed, and corrective measures taken if necessary. For example, the consistency constraints on a service name registry need not be as strong as that for many commercial data bases [13,9]. The consistency constraints on V, i.e., the acceptable level of consistency (or inconsistency) among the instances of V depend on the resource V abstracts. Relaxing the consistency constraints in turn reduces the complexity of the underlying algorithms that maintain the consistency of V.

Our approach is not to require absolute consistency but to provide simple primitives upon which applications can build additional properties if necessary. This allows usage of a minimal set of protocols for access operations, as required by the applications. Generally, this will result in better efficiency in the execution of the various operations. Inconsistencies, if any, may be detected and handled (by the protocols realizing the access operations) when the resource is accessed. Suppose, for example, the binding information for an object changes because the object has migrated or changed its name [14]. Correcting all the externally-held references to the object requires atomic delivery of notifications of the change. Instead, if a client of the object can correct its reference to the object at the time of access using a search protocol [22], a weak delivery of the notification message (i.e., the message delivery need not be atomic) may be sufficient at the time of the change, or even no notification at all (section 7.2.3). Thus, a relaxed consistency of the ADSV may be sufficient in many applications. More detailed discussion can be found in [1].

Thus our notion of consistency refers only to operational consistency, i.e., the variable V need not be totally consistent so long as the correct operation of the group is not compromised. This approach shares some ideas with Cheriton's model of problem-oriented shared memory with the 'fetch' and 'store' operations on the memory defined in an application-dependent manner [9]. Such a relaxed consistency allows the various operations on the ADSV to be realized using a weak form of group communication as described in the following sections.

7.1 Semantics of group communication

The semantics of group communication is quite complex because:

1. The outcome of the requested operation at each member of the group may be independent of one another.

2. What happens at a particular member may not influence the outcome of the group communication. This is, for example, the case when the sender considers the operation successful if at least one member of the group has carried out the operation despite failure at other members. In addition, such application-level failures may not be distinguishable from partial failures.

3. The constituency or the size of the group may be unknown to the sender, and in fact, may change dynamically.

Taking these into considerations, we introduce a parameter R in the group communication primitive [7]. R is specified by the sender of a group message, it indicates the number of members that should carry out the requested operation for the communication to be successful. R combines the (attribute, value) pairs described earlier in section 2.3 with a qualification criterion. The criterion specifies the condition for the outcome of the communication to be considered successful. Thus, the sender of a group message may specify R and a set of (attribute,value) pairs in the group communication primitive. These are used to pattern match with the return values from the group members to determine if the requested operation is successful. The operation is considered successful only if at least R of the replies meet the qualification criterion.

It is desirable for some applications to specify R independent of the size of the group. Examples of such applications are updates on replicated data, distributed election and synchronization among the members of a group. Other applications require a specific number of replies meeting the qualification criterion. An example of such applications is name solicitation such as searching a file or binding a host name to its network address. Consequently, we specify R as follows:

Case i). R = (FRACTION, r), where $0.0 < r \leq 1.0$.

The group communication layer acquires the size N of the group using a protocol such as that based on logical host groups used in the V-kernel [11]. After receipt of at least r*N replies which meet the qualification criterion or a timeout, whichever occurs first, the sender is notified the outcome of the communication using the representation **ATLEAST(s)**, where s is a fraction indicating the relative number of group members whose return values satisfy the criterion. Note that s may or may not be the

same as r. See [7] for details.

Case ii). R = (INTEGER, num), where *num* > 0.

After the specified number, *num*, of replies meeting the qualification criterion are received or a timeout occurs, whichever is earlier, the sender is notified of the actual number of replies that qualify.

Case iii). R = (FRACTION, 0.0) or (INTEGER, 0).

The communication is stateless in that the sender is notified of success immediately after the group message is sent. The receiving members may not reply to the message.

We now show by examples how the ADSV operations proposed in section 3.2.1 may be realized using the above form of group communication. As we shall see, the underlying protocols often exemplify the contention style of communication among the members of the group.

7.2 Sample illustrations

Some examples are given below illustrating the use of the ADSV model:

7.2.1 Host name allocation

The logical host name (id) space managed by a distributed kernel is a shared resource, and name allocation to newly joining machines must be arbitrated to ensure uniqueness. Non-reusability of the allocated id's is another desirable property for reliability reasons. It may be achieved by polling a certain number N of hosts to determine the last allocated id and then using the next higher id. During this activity, the machine should exclude other machines from accessing the id space (**Lock** operation). In other words, the procedural realization of the the **Write** operation on the id space should encapsulate this exclusion mechanism [8]. The variable is subject to inconsistencies which may lead to issues such as duplicate id allocation and unused id's. Relaxing the consistency constraints on the variable depends on the implications of such issues. For example, the issue of unused id's may not be serious if the id space is chosen to be large (say a 24-bit host id resulting in 2^{24} possible id's). Thus a weak form of the the underlying group communication suffices for operational consistency. Accordingly, a new machine specifies R = (INTEGER,N) in polling (**Read** operation on the id space) for the highest host id. Before adopting the selected id, the machine specifies R = (INTEGER,1) to solicit objections from other machines. If even one objection is raised, the machine drops the tentative id and repeats the procedure.

7.2.2 Replicated data

Consider a client operating on a replicated file. An operation on the file may be idempotent or non-idempotent. A sequence of idempotent operations on the file may proceed concurrently without requiring any atomicity and/or ordering constraints[3] at the server end (see section 6.1.1). In this case, the client needs only one positive reply from any member of the server group (R=(INTEGER,1)) for each of the operations (atomicity not necessary). These operations may be interspersed by each of the members in any order, say, order of request arrivals, including re-executions. Non-idempotent operations however should meet atomicity and ordering constraints to maintain the various instances of the file in a consistent state. For each of these operations, R=(FRACTION,1.0) may be specified by the client to ensure execution by all members; the members should also prevent re-executions and preserve order among the operations. The leader in the server group or the client may use R=(FRACTION,1.0) to commit or abort such operations [2].

7.2.3 Publishing name bindings

A distributed name server group consists of a set of name server processes distributed across the network. When a server process registers a service under its group id with the local name server, the latter may publish this information by sending an intragroup message to members of the name server group. This message may serve as a hint that may be cached by the remote name servers. If such a publishing is only a supplementary activity to client-driven name solicitation mechanisms, then the local name server does not require confirmed delivery of the message to any of the other name servers. Thus it may specify R=(FRACTION,0.0), and may declare the operation of publishing (i.e., Write operation on the name space) successful as soon as the message is dispatched.

7.2.4 Name resolution

When a client wishes to resolve a logical name (such as a service) to its contact address, it may send a message to the local name server requesting a **Read** operation on the name space. The latter may send an intragroup message soliciting the binding information from the name server group. It may specify R=(INTEGER,1). If there is at least one reply with the information, the name resolution may be considered successful and the binding established. The returned binding information may be inconsistent however, and its correctness is to be ascertained at a higher level (relaxed consistency).

[3]The causal ordering of the operations is a client-level requirement, and may be met at the client end.

7.2.5 Shared printer

A printer spooler is a resource shared by clients. The spooler may realise the **Lock** operation on the printer as follows: It may first solicit any objection from other spoolers to a proposed access to the printer specifying R=(INTEGER,1) in the group communication primitive. If at least one objection is raised, the attempt is aborted and retried later. If there is no objection, it may commit a lock on the printer by sending a group message with R=(FRACTION,0.0). When **Unlocking** the printer, the spooler may specify R=(FRACTION,1.0) advising release of the lock.

8 Conclusions

This paper has specified the semantic requirements of IPC abstractions for structuring reliable client-server and intra-server group interactions in a distributed program. Our program model is distinct from those used in other related works on failure handling in that the model is application-driven, i.e., the concepts underlying the model reflect application characteristics. These concepts — the idempotency properties, the notion of connection-less interactions and intra-server group interactions — influence the design of the failure recovery algorithms. Knowledge of these application characteristics simplifies the recovery algorithms considerably.

RPC is the abstraction chosen for client-server interactions. The failure semantics of RPC has been specified. The semantics requires proper handling of orphans arising due to failures. As an alternative to killing the orphans, techniques have been proposed to adopt such orphans and avoid roll backs wherever possible. The techniques involve controlled call re-executions and event log based call replay. The underlying issues of call idempotency and mutually interfering calls have also been examined.

A shared memory-like abstraction, which we refer to as ADSV, is introduced to map onto intra-server group interactions for managing shared resources. Primitive operations on the ADSV have been specified. We have shown that the consistency requirements of shared variables (associated with shared resources) such as information on name bindings and leadership within a server group can be relaxed. Thus the algorithms and protocols to realize the ADSV operations enforce consistency of the variable only to the extent required by the underlying resource. This allows use of a weak form of group communication among the various members of the server group in realizing the operations. Examples are given to illustrate the ADSV abstraction.

The above abstractions strive to mask partial failures inherent in the distributed environment. The high level specifications of such abstractions provide the system designer with a concise set of functional requirements of the client-server interface for building reliable systems. The specifications are applicable to a

variety of the underlying hardware and software architectures.

References

[1] K. Ravindran. **Reliable Client-Server Communication in Distributed Programs.** Technical Report, University of British Columbia, July '87.

[2] M. Paul B. W. Lampson and H. J. Siegert, editors. **Distributed Systems: Architecture and Implementation.** Springer Verlag Publishing Co., '81.

[3] K. P. Birman and et al. **Implementing Fault-Tolerant Distributed Objects.** *IEEE Transactions on Software Engineering*, SE-11(6):502–508, June '85.

[4] K. P. Birman and T. A. Joseph. **Reliable communication in the presence of failures.** Technical Report TR85-694, Dept. of Computer Science, Cornell University, July, revised Aug.'86 '85.

[5] A. D. Birrell and B. J. Nelson. **Implementing Remote Procedure Calls.** *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. '84.

[6] R. H. Campbell and B. Randell. **Error Recovery in asynchronous Systems.** *IEEE Transactions on Software Engineering*, SE-12(8):811–826, May '86.

[7] S. T. Chanson and K. Ravindran. **A distributed kernel model for reliable group communication.** In *6-th Symposium on Real Time Systems*, pages 138–146, IEEE CS, Dec. '86.

[8] S. T. Chanson and K. Ravindran. **Host identification in reliable distributed kernels.** *Computer Networks and ISDN Systems*, 15(1988) 159–175, Aug. '88.

[9] D. R. Cheriton. **Problem-oriented shared memory: A decentralised approach to distributed system design.** In *6-th International Conference on Distributed Computing Systems*, pages 190–197, IEEE CS, May '86.

[10] D. R. Cheriton. **VMTP: A Transport Protocol for the next generation of Communication Systems.** In *Symposium on Communication Architectures and Protocols*, pages 406–415, ACM SIGCOMM, Aug. '86.

[11] D. R. Cheriton and W. Zwaenopoel. **Distributed process groups in the V-Kernel.** *ACM Transactions on Computer Systems*, 3(2):77–107, May '85.

[12] F. Cristain and et al. **Atomic broadcast: From simple diffusion to byzantine agreement.** Technical Report RJ4540(48668), IBM Research Laboratory, San Jose, Calif., Dec. '84.

[13] B. W. Lampson. **Designing a global name service.** In *5-th Symposium on Principles of Distributed Computing*, pages 1–10, ACM SIGOPS-SIGACT, Aug. '86.

[14] F. C. M. Lau and E. G. Manning. **Cluster-based addressing for Reliable Distributed Systems.** In *4-th Symposium on Reliability in Distributed Software and Database Systems*, pages 146–154, IEEE CS, Oct. '84.

[15] K. J. Lin and J. D. Gannon. **Atomic Remote Procedure Call.** *IEEE Transactions on Software Engineering*, SE-11(10):1121–1135, Oct. '85.

[16] B. Liskov and R. Scheifler. **Guardians and Actions: Linguistic support for Robust Distributed Programs.** *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July '83.

[17] M. A. Malcolm and R. Vasudevan. **Coping with network partitions and failures in a distributed system.** In *4-th Symposium on Reliability in Distributed Software and Database Systems*, pages 36–44, IEEE CS, Oct. '84.

[18] M. S. Mckendry. **Ordering actions for visibility.** *IEEE Transactions on Software Engineering*, SE-11(6):509–519, June '85.

[19] M. S. Mckendry and M. Herlihy. **Time-driven orphan elimination.** In *6-th Symposium on Reliability in Distributed Software and Database Systems*, pages 42–48, IEEE CS, Jan. '86.

[20] M. L. Powell and D. L. Presotto. **PUBLISHING: A Reliable Broadcast Communication Mechanism.** In *9-th Symposium on Operating System Principles*, pages 100–109, ACM SIGOPS, June '83.

[21] K. Ravindran and S. T. Chanson. **Orphan Adoption-based Failure Recovery in Remote Procedure Calls.** Technical Report 87-3 (to be published), Dept. of Computer Science, Univ. of British Columbia, Jan. '87.

[22] K. Ravindran and S. T. Chanson. **State inconsistency issues in local area network based distributed kernels.** In *5-th Symposium on Reliability in Distributed Software and Database Systems*, pages 188–195, IEEE CS, Jan. '86.

[23] S. K. Shrivastava. **On the treatment of orphans in a distributed system.** In *3-rd Symposium on Reliability in Distributed Software and Database Systems*, pages 155–162, IEEE CS, Oct. '83.

[24] Liba Svobodova. **File Servers for Network-based Distributed Systems.** *ACM Computing Surveys*, 16(4):350–398, Dec. '84.

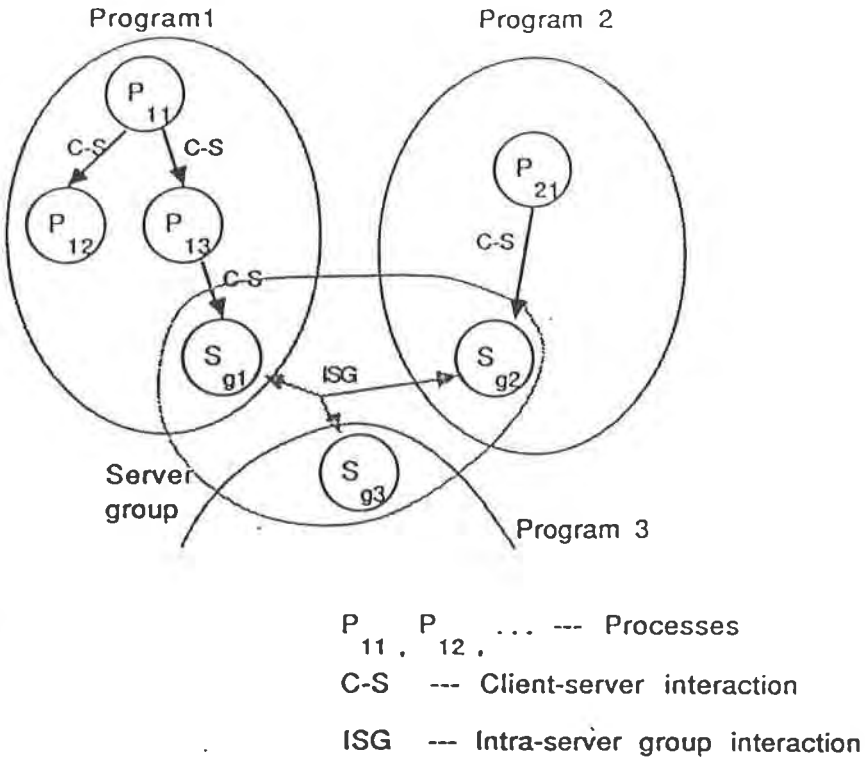

Figure 1. Logical view of a distributed program

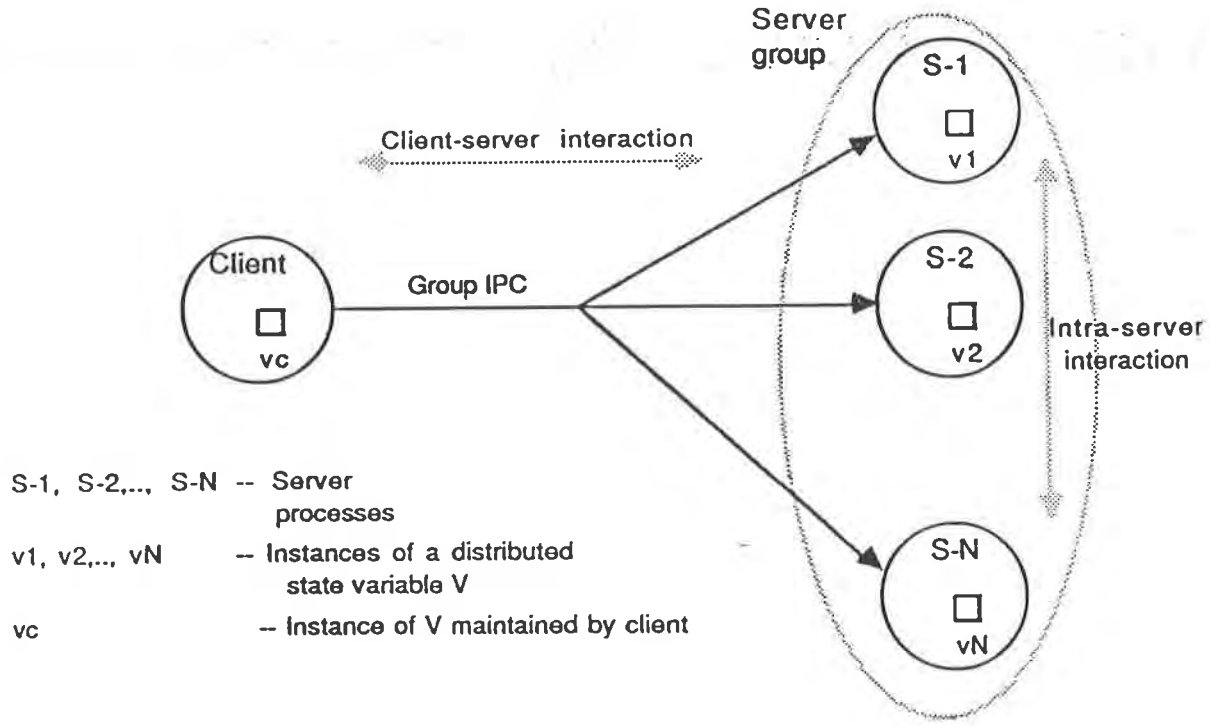

Figure 2. Logical model of a distributed server

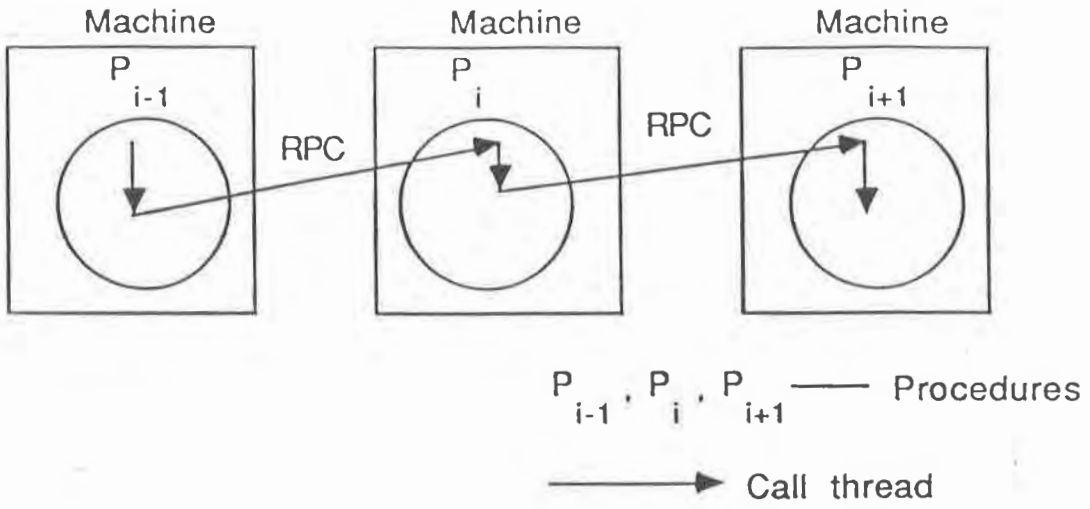

Figure 3. Remote procedure call

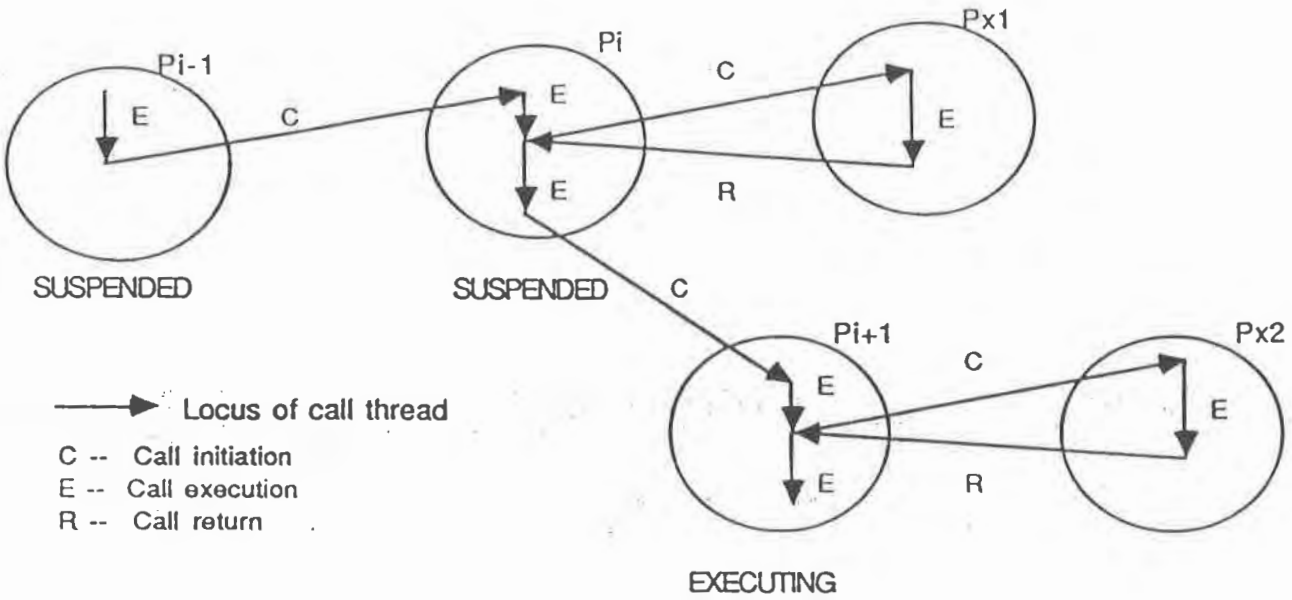

Figure 4. Locus of the remote procedure call thread

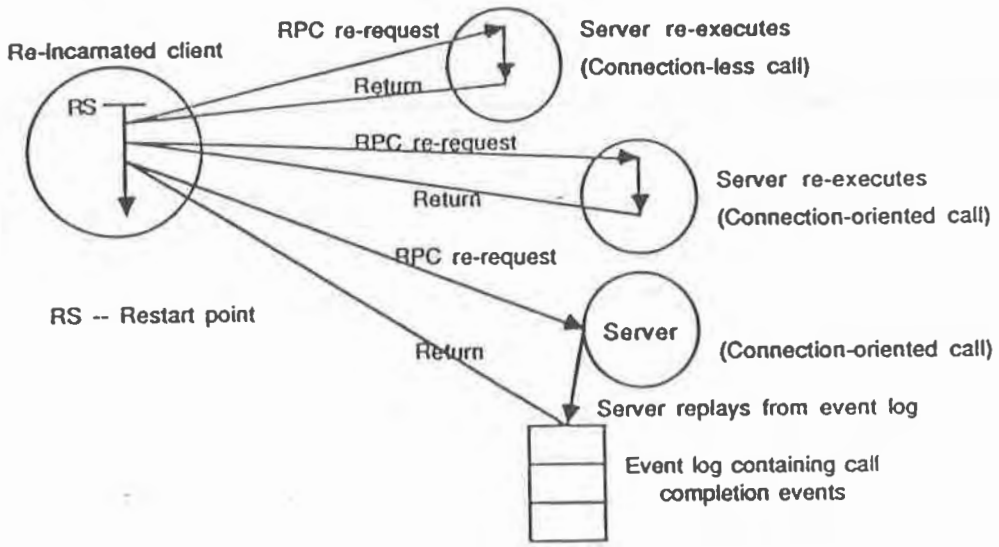

Figure 5. Recovery of a re-incarnated procedure