RELIABLE CLIENT-SERVER COMMUNICATION IN DISTRIBUTED PROGRAMS

By

K. RAVINDRAN

B. Eng., Indian Institute of Science, Bangalore, 1976

M. Eng., Indian Institute of Science, Bangalore, 1978

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES
(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

July 1987

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date 24th September 1987

Abstract

Remote procedure call (RPC) and shared variable are communication abstractions which allow the various processes of a distributed program, often modelled as clients and servers, to communicate with one another across machine boundaries. A key requirement of the abstractions is to mask the machine and communication failures that may occur during the client-server communications.

In practice, many distributed applications can inherently tolerate failures under certain situations. If such application layer information is available to the client-server communication layer (RPC and shared variable), the failure masking algorithms in the communication layer may relax the constraints under which the algorithms may have to operate if the information is not available. The relaxation significantly simplifies the algorithms and the underlying message transport layer and allows formulation of efficient algorithms. This *application-driven approach* forms the backbone of the failure masking techniques described in the thesis, as outlined below:

Orphan handling in RPCs: Using the application-driven approach, the thesis introduces a new technique of *adopting* the orphans caused by failures during RPCs. The adoption technique is preferable to orphan killing because orphan killing wastes any work already completed and requires rollback which may be expensive and sometimes not meaningful. The thesis incorporates orphan adoption into two schemes of replicating a server: i) *Primary-secondary* scheme in which one of the replicas of the server acts as the primary and executes RPCs from clients while the other replicas stand by as secondaries. When the primary fails, one of the secondaries becomes the primary, restarts the server execution from the most recent checkpoint and adopts the orphan. ii) *Replicated execution* scheme in which an RPC on the server is executed by more than one replica of the server. When any of the replicas fails, the orphan generated by the failure is adopted by the surviving replicas. Both schemes employ call re-executions by servers based on the application-level idempotency properties of the calls.

Access to shared variables: Contemporary distributed programs deal with a new class of shared variables such as information on name bindings, distributed load and leadership within a service

group. Since the consistency constraints on such system variables need not be as strong as those for user data, the access operations on the variables may be made simpler using this application layer information. Along this direction, the thesis introduces an abstraction, which we call *application-driven shared variable*, to govern access operations on the variables. The algorithms for the access operations on a variable use intra-server group communication and enforce consistency of the variable to the extent required by the application.

The thesis describes complete communication models incorporating the application-driven approach to mask failures.

Contents

List of Figures

ACKNOWLEDGEMENT

I would like to thank all of the people who have helped me in one way or another during my entire research. I can only acknowledge a few of them here:

1. My thesis advisor Dr. Samuel T. Chanson for providing excellent support, encouragement and advice for the past four years.

2. Members of my Ph. D. Committee — Dr. Son T. Vuong, Dr. Mabo R. Ito, Mr. Alan Ballard and Dr. Gerald Neufeld — who spent significant amount of time reading the thesis and discussing it despite their busy schedules.

3. The external examiner Dr. K. P. Birman of the Cornell University whose comments (sometimes not very pleasant!) helped to make the thesis more presentable.

4. The System Staff in the department who were helpful during the prototype implementations for the thesis, and fellow graduate students whose constructive comments helped to emphasize certain key aspects of the thesis.

5. My friend Dr. K. K. Ramakrishnan who spent long hours on phone from DEC discussing problems of mutual interest which helped to refine certain ideas in the thesis.

6. And my wife Sundari for her love, support and perseverance, my sons Karthik and Chandran, my parents and brothers for their moral support.

I also thank the Canadian Commonwealth Fellowship Administration for their continued support without which I would not have embarked on the research in the first place. I also thank the Government of India for sponsoring me for the fellowship, and the ISRO Satellite Centre, Bangalore for their support in the form of providing me leave of absence from my job.

Chapter 1

Introduction

This thesis is concerned with problems and solution techniques of providing failure transparency in distributed programs. The motivation of this research is presented in section 1.1. Section 1.2 provides a brief description of the client-server model on which the solution techniques are based. In section 1.3, the concepts of failure transparency and replication as the key in providing failure transparency are introduced. Section 1.4 describes how failures are treated at different layers of the operating system. Our premise of using application-level information in the solution techniques are outlined in section 1.5. Finally, general assumptions made in our solutions are presented in section 1.6.

1.1 Motivation for the research

Distributed systems consisting of personal computers and high performance workstations interconnected by a fast local area network (LAN) are becoming popular. Such systems offer a new mode of computation not practical with conventional systems consisting of mainframe and supermini computers interconnected by wide area networks (WAN). This is primarily due to faster inter-machine communication, availability of inexpensive broadcast capability, lower cost-performance ratio of workstations relative to mainframes, and the more controllable environment in LAN-based systems[1]. These characteristics make possible an

[1]Some of these characteristics may be possible even in a WAN environment in the future with the availability of fiber optic networks.

extensive distribution of the computing resources (typically data, hardware and software) across different machines in the LAN and a transparent, cost-effective sharing of the resources. Thus in such distributed systems, often more than one machine will take part in the execution of a program. An example of such a system consists of diskless machines accessing, over the Ethernet [52], files implemented by high speed machines on their local disks [1].

However, with this new mode of computing, new problems arise. Machine failures and failures in the communication path between machines (communication failures) are inherent in a distributed environment (c.f. section 1.6.1). Machines may fail independently, networks may fail or messages may be lost disrupting the on-going interprocess communications (IPC) between the various processes of the program. The system should recover from such failures and mask them from the communicating processes so that they may continue to execute despite the failures — a feature known as *failure transparency* in the program. Failure transparency is important to support *network transparency*, a requirement that the communicating processes should not perceive any machine and network boundary intervening between them[2]. The issue of failure transparency assumes an added importance in LAN-based distributed systems because of the extensive IPCs across machines in these systems. This provided us with the motivation to research the issue.

The software communication model chosen as the basis of this work is the *client-server* model because it maps well onto this type of architecture, and is widely used for describing the IPC in distributed systems.

1.2 Client-Server communication

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*. A server exports an abstract view of

[2]Some other flavors of network transparency are operation transparency and performance transparency. For a general discussion, see [23].

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 exchange is fundamental to client-server communications (also referred to as client-server interaction) [5,37,15]. The clients and the servers do not assume the existence of shared memory among them because they may possibly reside on different machines. Thus, they communicate typically by exchanging messages.

A resource can be viewed as an abstraction implemented on top of a low level resource. Thus, a server implementing a resource for its clients often needs to communicate as a client with another server. 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, a process acting as the server to a client in one message exchange may act as the client of another server in another exchange. In other words, the role of a process as a client or a server is dynamic, and is valid only for the duration of the particular request-response exchange.

For reasons of reliability and availability, a service may sometimes be provided by a group of server processes with each process running on a different machine and providing the same or a different subset of the service. The failure of one or more of the processes may result in a degraded service instead of total loss of the service. An example is a distributed file service which consists of a group of server processes with each one managing a subset of the file name space. If a process in the group fails, only the files managed by the failed process will be unavailable to clients.

A *distributed operating system* for this type of LAN-based system architecture may then be viewed as providing two types of abstractions: *resource abstractions* implemented by servers (e.g., terminal server, print server) and a *communication abstraction* implemented by the lower layers of the operating system. These layers allow clients to communicate with servers across machine boundaries. We refer to these layers collectively as the run-time system or the client-server communication interface. See Figure 1.1.

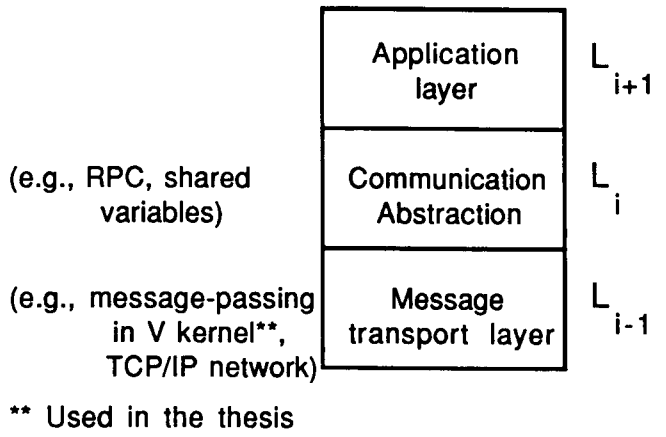


Figure 1.1: Layers in the client-server interface

The communication abstractions (layer L_i) provide a natural and easily understandable interface to the clients and servers (also referred to as the application layer — layer L_{i+1}). A well-known communication abstraction is the *remote procedure call* (RPC) which may be used by a client to communicate with a server across machine boundaries with the same semantics as the procedure call mechanism used within a single machine [8] (see section 2.7.1 for more details). The communication abstractions reside on top of a message transport layer (layer L_{i-1}) which provides low level message transport between processes on different machines. Examples of the message transport layer are the TCP/IP backbone network used in the UNIX operating system [1] and the message-passing kernel used in the V distributed operating system [14][3].

A program running under such a distributed operating system consists of clients and servers communicating extensively with one another across machine boundaries, so we refer to it as a *distributed program*[4]. The thesis centers primarily on the communication abstractions used in distributed pro-

[3]The application layer consists of 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 applications (e.g., numerical program, database access program). The RPC layer is written by system programmers to provide the communication interface to the applications. The message transport layer is written by system programmers to provide network device driver level support.

[4]Programs on WAN-based systems cross machine boundaries usually for specific applications, such as accessing remote

grams, with specific focus on handling machine and network failures that may occur during client-server communications. In the following sections, we discuss the issues related to such failures.

1.3 Failure transparency in distributed programs

Failures that occur in some but not all processes in a distributed program are referred to as *partial failures* in the program. Partial failures considered in the thesis are the death of a process, the failure of a machine destroying the local processes, and failures in the communication paths between the various processes in the program (c.f. section 1.6.1). Effectively, such failures may cause some processes in the program to fail, and sometimes may lead to total failure of the program. Certain failures may be observed by all the relevant processes in the program, yet other failures may only be observed by some of the processes. A failure of the latter type may occur when the conditions that caused the failure cease to exist before all processes in the program have noticed it.

Because of partial failures, a server execution caused by a client request may sometimes continue to exist even after the client request has been terminated. Such executions are called *orphans*, and they may interfere with normal executions by wasting resources and introducing inconsistencies in the system. Since inter-machine client-server communication occurs frequently in distributed programs, the issue of failure transparency is critical to network transparency. Our approach to provide failure transparency is to require the run-time system recover from the partial failures and mask their effects from other processes in the program. This will allow the program to continue to function normally in the presence of such failures without the application software seeing the effects of the failures.

The key to providing failure transparency is *replication*. We outline the concept in the following section.

---

databases and sending mail. We exclude such programs from our discussion.

1.3.1 Replication of services

Replicating a service refers to providing the same service (to clients) from more than one machine[5]. Assuming that each machine is an independent unit of failure (be it a failure of the machine or a communication failure partitioning the machine from the rest of the system) and the run-time system uses some recovery techniques to mask the failure of a server from its clients, the clients will continue to execute unaware of the failures as long as at least one of the replicated servers is operational and accessible.

Though the recovery techniques may have many similarities to that proposed for distributed databases, they also have different requirements and considerations for distributed programs. It is useful to note the differences between distributed programs and distributed databases in the context of failure recovery.

1.3.2 Distributed programs and distributed databases

The primary consideration for a distributed database system is to keep the *secondary storage data* available and consistent. The issues are how to distribute or replicate the data to survive failures. Given the support mechanisms, rollback on the contents of the data and replication of the data are usually possible. In the overall context of an operating system, a database is a resource abstraction implemented by a server. The latter may choose to implement its own internal file system, failure recovery protocols and IPC techniques. In other words, failure transparency is done at the resource abstraction level.

In a distributed program, the servers may be replicated to increase *program availability* (or service availability). The goal is to keep the program state consistent despite failures, where program state refers to:

1. Information maintained in the client-server communication interface. Examples are information

[5]Note that replicating a service is orthogonal to distributing the service.

on name bindings, leadership within a group of servers, and status of communicating machines. The consistency requirements on such information need not be as strong as those for databases because inconsistencies can usually be detected on usage of the information and corrected at that time.

2. The permanent and temporary variables maintained by the servers. An example of permanent variables is the information on the attributes and the status of a printer (or a terminal) managed by a server. Even though support mechanisms may be provided, rollback or replication of permanent variables may not be possible or meaningful.

Since the program states are largely independent of the resources, this suggests that failure transparency should be provided at the communication abstraction level, i.e., in the operating system layers that allow clients and servers to communicate.

1.4 A framework for handling failure events

Failure transparency in distributed programs requires correct handling of the failure events in the various operating system layers beneath the programs. The requirement is described in the following subsection.

1.4.1 Failures and event ordering

Refer to Figure 1.1. As mentioned earlier, L_{i+1}, L_i and L_{i-1} are layers of abstractions with L_{i+1} on top of L_i and L_i on top of L_{i-1}. The layer L_i is characterized by a set of events, including failures, it receives from L_{i-1} and L_{i+1}. The general premise used in existing works on failure tolerance is that the layer L_i should ensure *atomicity* and *ordering* of the events it receives [5,7]. This approach allows L_i to run as a completely asynchronous layer with respect to L_{i-1} and L_{i+1}. In other words, the structures, protocols and algorithms used within L_i are not visible to L_{i-1} and L_{i+1}; only the interfaces are known[6]. The

[6]The approach is widely accepted as a good software engineering practice because it allows a complex layer to be broken down into more easily manageable layers, and a modular interface between them [5]. It also makes proving correctness of operations manageable.

approach requires uniform treatment of the events in enforcing the atomicity and ordering constraints. Violation of the constraints may lead to incorrect actions taken by the various communicating processes in L_i.

Suppose during a client-server communication, the client observes a temporary network failure and aborts its request. A correct order of the events at the server will be for the server to receive the request, then observe the failure and abort the requested operation. If the server does not observe the failure, it will complete the operation. In this case, or, if the server observes the failure after it has completed the operation, the operation is incorrect because the client has aborted its request and does not expect the server to complete the operation.

1.4.2 Failure recovery and failure masking

Suppose a failure event is observed by the layer L_{i-1}. The failure is unrecoverable if it cannot be hidden from the layer L_i. Such an unrecoverable failure shows up as an event in layer L_i. Failure recovery in the layer L_i refers to the activity initiated by this layer to handle the failure. The recovery is successful if L_i can hide the effects of the failure from L_{i+1}, otherwise the recovery is unsuccessful and the failure event should be exposed to L_{i+1}. Thus failure masking in layer L_i refers to a successful failure recovery in this layer. Consider the example of a file server. Any error encountered during a read or write operation on the disk is hidden by the underlying disk server from the file server if the error can be recovered (e.g., by a retry of the operation). If the error is unrecoverable, then the disk server delivers an exception event to the file server which then performs recovery.

1.4.3 Failure semantics

Given the view of how failures may propagate upwards across layer boundaries, the *failure semantics* of the layer L_i specifies the characteristics of recoverable failures in L_i and what the unrecoverable failures delivered to L_{i+1} are. The semantics specifies the requirements on the failure recovery technique

used in L_i, however, it does not specify the choice of any particular technique. Alternatively, the semantics specifies when a failure in L_i is considered to have been masked from L_{i+1}, and hence what the failure transparency in L_i means. To ensure failure transparency, L_i should employ a suitable recovery technique.

Thus if RPC is used for client-server communication, then failure transparency in RPC requires addressing the failure recovery issues in the various layers as follows:

Message transport layer: This layer deals with the detection of machine and network failures, the handling of message loss, delay and ordering. Machine and network failures cannot be recovered at this level. In some cases, failures related to message loss, delay and ordering are also unrecoverable. Such unrecoverable failures are passed on to the RPC layer above.

RPC layer: This layer deals with the recovery from machine and communication failures propagated from the message transport layer. It attempts to mask the failures from the application layer. Since these failures inherently generate orphans, the layer should handle any state inconsistency caused by the orphans. Unrecoverable failures are passed on to the application layer as exceptions.

Application layer: This layer deals with the handling of the exceptions delivered from the RPC layer. The system may handle the exceptions by simply aborting the program, or, the user may supply appropriate exception handlers. The language in which the programs are written may provide some form of exception handling constructs (such as those in Mesa [36]).

As seen above, a failure semantics is definable at each layer. The issue is to choose the right semantics for a given layer, i.e., to define exactly what the failure transparency in the layer means. Given a failure semantics of the RPC layer, the underlying recovery may employ techniques such as rollback and logging. The choice of the techniques is primarily based on the desired level of failure transparency, the

frequency and form the failures can occur in the given system environment, and the characteristics of the underlying message layer. Additionally, the characteristics of the application layer also may influence the recovery algorithms, as described in the next section.

1.5 Application-driven approach to failure recovery

In practice, many distributed applications can tolerate certain types of failures. A typical example is the broadcast of a message by a client to a selected group of server processes, say to search for a file or to get time information. In both cases, the message need not be delivered to every server in the group. For the file search, it suffices if the client gets a reply from the particular server that manages the file. For the time request, reply from any of the servers will do. Suppose during the file search, a server in the group fails or is partitioned from the client. The communication with the group is not affected by the failure if the failed server does not manage the file requested by the client or if the client can access the file from some other server in the group. Thus the communication layer supporting client-server communication need not expose the failure event to the client. As another example, consider the multiple executions of a server caused by re-transmissions of a call request message to the server from a client, i.e., *message orphans*; the re-transmissions may be caused by message losses or communication break-downs during return of the call when it was executed earlier [46,53]. Usually the RPC layer attempts to detect and eliminate the orphans. Instead, the application-layer may tolerate such orphans when the server executions are *idempotent*, i.e., the executions do not change the state of the server. Consider the network failure example of section 1.4.1. If the server execution is idempotent, then it does not matter whether the server observes the failure before or after completing the execution. In some cases, it does not matter if the failure is observed at all by the server. These examples suggest that the events generated by such applications need not satisfy the atomicity and ordering constraints. In this sense, attempting to enforce the constraints for such events tend to over-constrain the underlying

algorithms.

The observations that many applications can inherently tolerate certain types of failures under certain situations led us to believe that the characteristics of the application layer may significantly influence the recovery algorithms. We believe that the ordering and atomicity constraints on events in the RPC layer need not be subsumed into this layer but may be specified by the application layer above it. This premise allows relaxation of the constraints in the RPC layer using application-level information.

Based on our premise, the thesis uses an alternative approach which *softens* the boundaries of the RPC layer. This allows the RPC layer to get information about the application and the message layers. Using the information, the RPC layer may employ special purpose protocols that exploit the characteristics of the application layer and the special features of the message layer. For this reason, we refer to our approach as *application-driven.* The basic issue underlying our approach is what type of information should permeate from the application layer into the RPC layer, and how the information can be used in the recovery algorithms. Specifically, the information may be used to relax the constraints under which the protocols may have to operate if such information is not available (c.f. sections 3.3, 4.3 and 5.1). Such relaxation may simplify and optimize the recovery algorithms.

To our knowledge, no other work on failure transparency has systematically attempted to design failure recovery algorithms using the application-driven approach as presented in this thesis. In fact, many existing works do not use this approach at all. They do not make use of the application level information which may otherwise simplify failure recovery. This motivated us to study the approach in depth.

1.6 Thesis goal and assumptions

The goal of the thesis is to study the application-driven approach to provide failure transparency in distributed programs. It requires:

1. Specifying the failure semantics of the communication abstractions,

2. Formulating the recovery algorithms as part of the abstractions to provide failure transparency. Application-level characteristics may be used in formulating the algorithms.

Some recent works have explored the idea of employing the failure masking techniques as part of the communication abstraction. Examples along this line include ARGUS which provides atomic actions as part of the RPC level construct [31]. The underlying issues of orphan detection and rollback are handled by the run-time system. Lin provides a model of RPC in which rollback-based failure recovery is part of the RPC[30]; orphan killing is used as the underlying technique in the model. CIRCUS uses replication of server executions to mask failures during the RPC [17]. When a client makes a RPC, each replica of the server executes the call; when all the replicas complete the execution, the call is considered to be completed. Thus recent works have experimented with employing failure masking techniques in the RPC layer.

The major difference between the above and our work is that our approach is application-driven, which allows usage of application level characteristics in the failure recovery algorithms.

The thesis makes certain assumptions about the environment and the software architecture around which the issues are considered and their solution techniques applied. These assumptions are given below:

1.6.1 Operating environment

The hardware base consists of a set of workstation-class machines (e.g., SUN workstations) interconnected by a fast LAN with a broadcast capability. The network may be of the bus type such as the Ethernet [52] or the ring type such as the Token ring [26], or a locally interconnected collection of such media through fast gateways [12]. The hardware allows efficient receiving and sending of packets by the software. The client machines may or may not have a local disk (c.f. section 6.2.4), and are provided

with a booting and a file service by a collection of server machines which have one or more fast local disks. The V Distributed Kernel supports the type of architecture described above and provides inexpensive processes interacting with one another through a message-based IPC mechanism [14]. The IPC primitives allow a process on any machine to send, receive and reply to a message from a process on any other machine. In addition, multicast or group communication is also supported so that a process may communicate with a group of processes using a single message [16]. Thus, the V kernel provides a suitable base for building distributed operating systems.

The V kernel does not enforce strong message delivery in the low level IPC primitives. To expand on this, the 'reply' primitive which allows a server to send a reply message to a client does not enforce atomic delivery of the message. Similarly, the group communication primitives do not enforce atomicity or ordering of the message delivery events, or recover from failures that may occur during a group communication [46,10]. Since the application-driven approach to failure recovery proposed in the thesis does not require strong message delivery, the V kernel's IPC maps well as the underlying message transport layer for the high level communication abstractions (e.g., RPC, shared variables) — see Figure 1.1. For the above reasons, prototype implementations of the schemes proposed in this thesis were done on top of the V kernel.

Types of failures considered

The thesis considers the effects of machine and communication failures only. Machines are assumed to exhibit a fail-stop behavior [51], i.e., if a machine fails, all the information in its volatile storage are lost, and it does not produce any messages after its failure.

A communication failure may be a communication break-down or a message failure. Typical situations where the break-down in communication may be perceived are the failure of the network or the gateway through which processes on the various machines communicate. This could result in the parti-

tioning of these processes. Messages may be lost due to channel errors, buffer overflow in the network interfaces or lack of high level resources such as message descriptors [16,54]. Messages may be received by a communicant more than once or out of order with respect to other messages from the same process, and delayed arbitrarily. However, the communication path may not corrupt messages undetectably.

Since a wide spectrum of other types of failures may also occur in a distributed program, it is appropriate to point out some of the failures **not considered** in the thesis. They are byzantine failures, application-level interpretation of an event or outcome as a failure (application-level failures — c.f 2.5), timing failures and intra-process failures [19,42]. These failures are best dealt with by the applications themselves.

1.7 Outline of the thesis

As mentioned earlier, the thesis focusses on an application-driven approach to failure transparency in a distributed program. The organization of the thesis is as follows:

Chapter 2 characterizes the interaction patterns in a distributed program from the application point of view. Client-server interactions may be connection-oriented or connection-less depending on the ordering requirement among the interactions. Additionally, when a service is distributed, the service is provided by a group of processes. The processes interact with one another to manage shared variables which may have application specific consistency requirements. We introduce a shared memory-like abstraction, which we refer to as *application-driven shared variable* (ADSV), to map onto the intra-server group interactions. We also specify the failure semantics of the communication abstractions — RPC and ADSV. Our above view forms the basis of the recovery algorithms described in the subsequent chapters.

Chapter 3 describes an algorithm to mask failures using the primary-secondary scheme of replicating a server in which one of the replicas acts as the primary and executes an RPC issued by a client while the

other replicas stand by as secondaries. The orphan generated by a failure is adopted by using event logs and call re-executions. Call re-executions require application-level information about the idempotency properties of the calls [54].

Chapter 4 describes an algorithm whereby more than one replica of the server execute the RPC. The algorithm uses the idempotency properties of the calls to maintain the replicas in identical states. It employs lateral coordination among the replicas, replay of call events from a log and call re-executions. The synchronization requirements among the replicas are relaxed wherever possible based on the idempotency properties of calls thereby allowing calls to be completed faster.

Chapter 5 describes a model for access to shared operating system variables. It relaxes the consistency requirements on the variables depending on their application-level characteristics, and formulates simple algorithms and protocols for accessing the variables. Examples are given to illustrate the algorithms.

Note that because the various algorithms and protocols use application-level information, they are built on top of a weaker form of low level message transport such as that provided by the V kernel. See Figure 1.1.

Chapter 6 concludes by pointing out the contributions of the thesis in distributed operating systems research, and identifying issues and areas for future research. The contributions take the form of formulating an application-driven approach to reliable client-server communication in distributed programs (c.f. 6.1). The approach allows:

- Comprehensive treatment of orphans in client-server communications from a new perspective,
- Introduction of a conceptual framework for managing shared operating system variables whose consistency requirements can be relaxed based on the application characteristics.

The approach is broadly applicable to contemporary distributed system architectures.

Chapter 2

Program model

This chapter characterizes the communication patterns in a distributed program based on the client-server model. The characterization is derived from the application level requirements. A formal description of the communication patterns is also given to allow the model to be concisely defined and to give the reader a clear understanding of the model. Communication abstractions which map well onto the communication patterns are presented and their failure semantics discussed. The program model forms the basis for the various recovery techniques described in the subsequent chapters. We also assume programs are deterministic (see section 2.6.2). Thus, the thesis deals with inconsistency issues arising only due to partial failures, but does not deal with those due to byzantine and non-deterministic behavior of the processes in the program.

We first formalize some concepts about atomicity and order among the events that characterize a distributed system (excerpts from [18]). The concepts are used in the treatment of partial failures as communication level events and in the formulation of the failure recovery algorithms presented in the various chapters.

2.1 Atomicity and ordering of events

An event sequence **EV_SEQ** is a set of distinct events $[e_0, e_1, \ldots, e_i, \ldots]$[1]. The ordering on **EV_SEQ** is denoted by $e_0 \succ e_1 \succ \ldots \succ e_i \succ \ldots$, where $e_0 \succ e_1$ represents the order 'e_0 happens before e_1'. A subsequence **EV_SUB_SEQ**(e_j, e_k) is any subset of **EV_SEQ** with the inherited ordering among the events in the interval (e_j, e_k), but not including e_j and e_k. The successor and predecessor functions, **succ**(e) and **pred**(e) respectively, are defined on all events $e \in$ **EV_SEQ** (except for the final event for **succ**(e) and the initial event for **pred**(e) when **EV_SEQ** is finite):

$$\textbf{succ}(e_i) = e_j \in \textbf{EV\_SEQ} \mid (e_i \succ e_j) \wedge (\textbf{EV\_SUB\_SEQ}(e_i, e_j) = \emptyset),$$

$$\textbf{pred}(e_i) = e_j \in \textbf{EV\_SEQ} \mid (e_j \succ e_i) \wedge (\textbf{EV\_SUB\_SEQ}(e_j, e_i) = \emptyset).$$

Two event sequences $\textbf{EV\_SEQ}_1$ and $\textbf{EV\_SEQ}_2$ are atomic with respect to each other if $\textbf{EV\_SEQ}_1 = \textbf{EV\_SEQ}_2$, i.e., every event observed in $\textbf{EV\_SEQ}_1$ is also observed in $\textbf{EV\_SEQ}_2$, and vice versa. The sequences are ordered with respect to each other iff, given the events $e_{j1} \in \textbf{EV\_SEQ}_1$ and $e_{j2} \in \textbf{EV\_SEQ}_2$ such that $e_{j1} = e_{j2}$, **succ**(e_{j1}) = **succ**(e_{j2}) and **pred**(e_{j1}) = **pred**(e_{j2}), and the above constraint holds for all events in $\textbf{EV\_SEQ}_1$ and $\textbf{EV\_SEQ}_2$. Thus if two sequences are ordered, it implies that they are atomic but not vice versa.

Let $[c_1, c_2, \ldots, c_k]$ and $[s_1, s_2, \ldots, s_{k'}]$ be the sequence of call events that originate at a client and arrive at a server respectively. Then $[c_1, c_2, \ldots, c_k]$ is referred to as the *causal sequence* of $[s_1, s_2, \ldots, s_{k'}]$. The properties of atomicity and order are usually defined between such sequences.

We now proceed to describe our model of a distributed program.

[1] Events are defined at a given level of abstraction and are uniquely identifiable.

2.2 Communication patterns

As described in section 1.2, server processes implement resources and client processes communicate with the servers to access the resources. Such a request-response style of communication is a major communication pattern in the program. For simplicity, we assume there are no recursive interactions between clients and servers.

In contemporary distributed system architectures, 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 for reasons of failure tolerance and availability. In this architecture, the client accesses the distributed service as a single 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 management of the resource by the server processes underscores some form of resource sharing among the processes, which requires communication among them to co-ordinate the sharing. We organize such server processes into a process group [16,47], 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 communicate among themselves to provide a unified interface to the clients. 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[2]. We refer to the communication among the members as *intra-server group communication.*

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

[2]At the minimum, the members share the group name $srvr\_gid$.

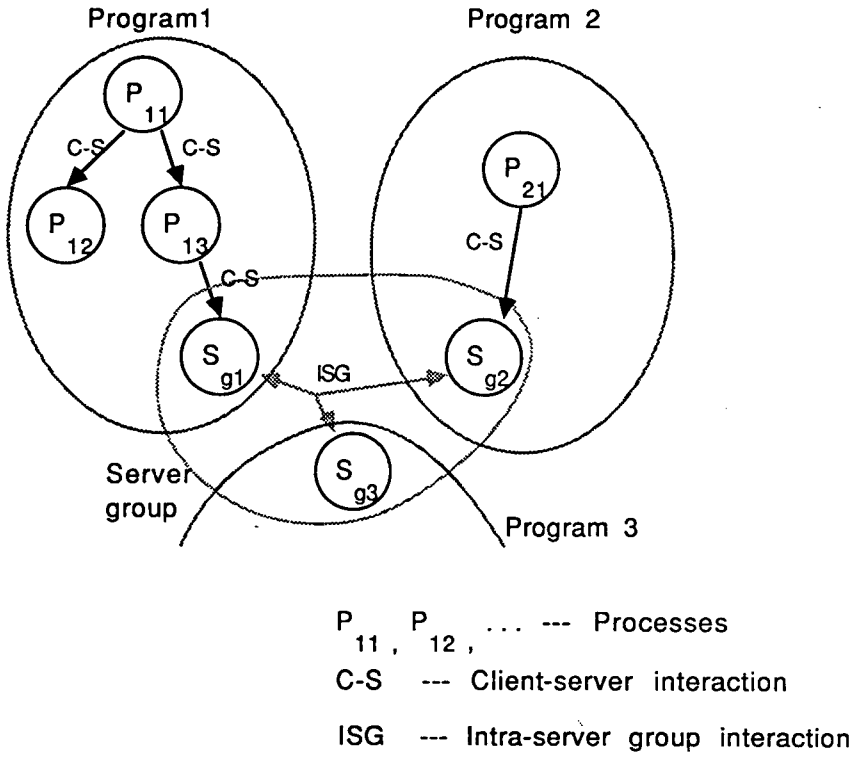


Figure 2.1: Logical view of a distributed program

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 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 interactions interspersed with intra-server group interactions. 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 2.1). Each of the two communication patterns is further discussed in the following sections.

2.3 Client-server interactions

Client-server interactions may be of two types — *connection-oriented* and *connection-less* [46,33] — as described below:

A client-server interaction is connection-oriented if in a sequence of such interactions, the server should maintain 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 the interactions throughout the duration of the connection. Among other things, 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[3] maintained by the server is the seek pointer. As another example, consider the static variables supported in a distributed implementation of the 'C' language [27]. 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. An anthropomorphic example is a bank customer operating on his account by interacting with a teller.

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. An anthropomorphic example is a bank customer asking a teller about currency exchange rates.

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 (c.f. section 3.4.4).

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

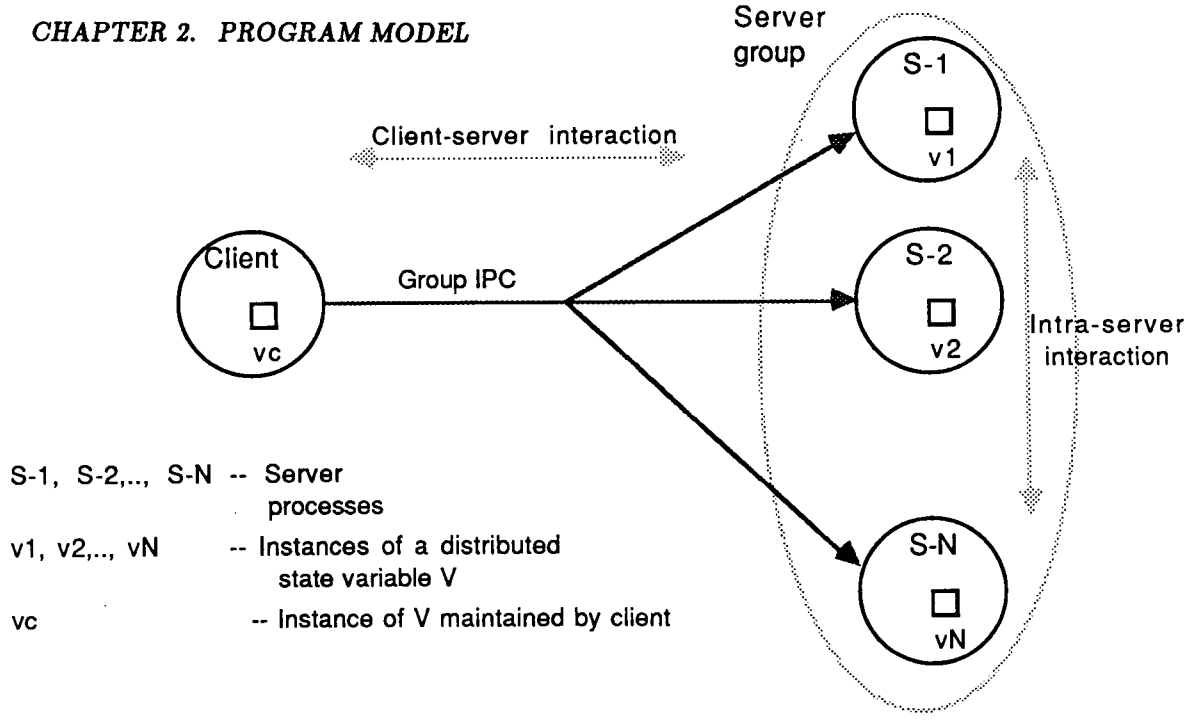


Figure 2.2: Logical model of a distributed server

2.4 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 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 2.2). V may assume a set of distinct 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. We suggest 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 5.1), and that the inconsistencies may be handled by the IPC abstraction that governs access operations on V.

The resource managed by a server group may be of two types — *intrinsic* and *extrinsic*. As we shall see later in chapter 5, whether a resource is intrinsic or extrinsic influences the techniques to maintain the consistency of the resource.

2.4.1 Intrinsic resources

An intrinsic resource may change its state without any relationship to client activities, i.e., the state change is asynchronous to a member's interactions with its clients. The resource should always be held by some member of the group under normal operations. An example is the leadership within the group arbitrated among the members in some way [20]. The arbitration and the failure recovery associated with leadership management are asynchronous to client activities. Another example is the name binding information maintained by a name server group. Except for name registrations, the binding information does not change in relationship to the client access activities. Thus, it is an intrinsic resource as far as name resolution activities are concerned.

2.4.2 Extrinsic resources

An extrinsic resource may change state in relationship to client access activities, i.e., the state change occurs synchronously with client access activities. An example is a printer managed by a spooler group.

The printer can change its state when clients access the printer for printouts and release it after the usage. When no client uses the printer, the printer is not held by any member of the spooler group. Consider the example of the name binding information given in the previous section 2.4.1. The information is an extrinsic resource as far as the name registration clients are concerned.

2.5 Formal characterization

We now formally characterize the interactions between the various processes in the distributed program. The characterization is useful to analyze the properties of the interactions concisely, and allows unambiguous formulation of failure recovery algorithms in the run-time system using these properties (see chapters 3 and 4).

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. An interaction (or a call) TR requested by the client and executed by the server is denoted by

$$(C_{bef}, S_{bef}) \stackrel{TR}{\rightarrow} (C_{aft}, S_{aft}) \tag{2.1}$$

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

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

2.5.1 State transitions

Consider a client-server interaction. The server may change its state in the following ways:

- Interactions as a client with other servers; the state transition is caused by the returned values p_val.
- Local executions operating on its internal variables.
- Intra-server group interactions when a shared resource is manipulated.

We examine each in detail below:

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. Let $\{att\_nm_j, (j = 1, 2, \ldots, K)\}$ be the set of attributes specified in TR, and att_val_j be a value returned by the server for att_nm_j. Then

$$p\_val = \left\{ \begin{array}{l} (att\_nm_1, att\_val_1), \\ (att\_nm_2, att\_val_2), \\ \quad \vdots \\ (att\_nm_K, att\_val_K) \end{array} \right\}.$$

The above (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, consider a client request to a server to create a process. The client may specify an attribute name CREATE_PROCESS in TR to indicate the request. The possible return values for the attribute may be CREATE_SUCCESS and CREATE_FAILED. If the returned value is, say CREATE_FAILED, then

$$p\_val = \{(CREATE\_PROCESS, CREATE\_FAILED)\}.$$

As another 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 (c.f. section 5.2).

The client level interpretation of the outcome *p_val* as a successful completion of TR or otherwise depends on the application. Take the second example given above where 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 as either a success if the search is a prelude to creating a new file under the name, or a non-success 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 thesis.

Local executions

We assume local executions in a server to cause deterministic state transitions in the server. This means that given an initial state, a local execution in the server will always take the server to the same final state irrespective of whether the execution is carried out at different times or on different machines in the system. In other words, the results of the execution are reproducible in time and space.

Operations on shared resources

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

Based on the above discussion of how a server may change state, we formalize the properties of client-server interactions in the next section. The properties are used in the failure recovery algorithms in the later chapters.

2.6 Properties of client-server interactions

Consider a client-server interaction (or call) TR, as given by the relation (2.1)

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

The ordering relationship of TR with respect to a sequence of calls exposes the idempotency properties of TR [53,54] as described below:

2.6.1 Idempotency

The idempotency property of the call TR relates to the effect of TR on the state maintained by the server. 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 a read operation on a file which does not change the seek pointer, time request to a time server group and file search request to a file server group[4]. 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.

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

To expose additional properties of TR that may be useful in the recovery algorithms, we introduce two concepts — *re-enactment* of TR and *re-execution* of TR.

2.6.2 Re-enactment

In a re-enactment 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 (2.1), then TR' is defined as

$$(C_{bef}, S_{bef}) \stackrel{TR'}{\rightarrow} (C_{aft'}, S_{aft'}),$$

where $C_{aft'}$ depends on (C_{bef}, TR', p_val') and $S_{aft'}$ depends on (S_{bef}, TR'). The concept of call re-enactment is useful in backward recovery schemes in which the server rolls back the effect of the call, and subsequently the client re-issues the call (c.f. sections 3.4.4 and 3.1). The idea is to be able to reproduce the effect of the call (i.e., $S_{aft'} = S_{aft}$ and $C_{aft'} = C_{aft}$). In order to accomplish this, the server should change state deterministically and the call TR should be deterministic. The former condition ensures $S_{aft'} = S_{aft}$ while the latter ensures $C_{aft'} = C_{aft}$. Since $C_{aft'}$ depends on (C_{bef}, TR', p_val') and since TR' has the same properties as TR, it follows that p_val' should be the same as p_val for TR to be deterministic. 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 re-enactment of the operation returns the same value as the original operation. Suppose the 'read' operation also returns a time stamp, then it is non-deterministic since every re-enactment 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 non-deterministic behavior

of program executions need not interfere with those provided to tackle the idempotency issues. Thus, for simplicity and without loss of generality, we will consider only deterministic programs in the thesis.

2.6.3 Re-execution

In a re-execution 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 (2.1), then $TR"$ is defined as

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

The concept of call re-execution is useful in the forward recovery scheme described in chapter 3 and the replicated execution scheme described in chapter 4. It is also useful in dealing with message orphans (see section 3.4.2).

In order for a re-execution to be useful, TR should be idempotent. It follows from the definition of idempotent calls (section 2.6.1) that if TR (and therefore $TR"$) is idempotent, then $S_{aft"} = S_{aft} = S_{bef}$. In other words, the server state does not change under re-executions of an idempotent call. Also, since TR is deterministic[5], $C_{aft"} = C_{aft}$. If TR is non-idempotent, then $S_{aft"}$, S_{aft} and S_{bef} may be different; also, $C_{aft"}$ 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.

2.7 Communication abstractions

Having characterized the interaction patterns in a distributed program from an application point of view, we now identify suitable communication abstractions which map naturally onto these patterns.

[5] As mentioned earlier, we will consider only deterministic programs in the thesis.

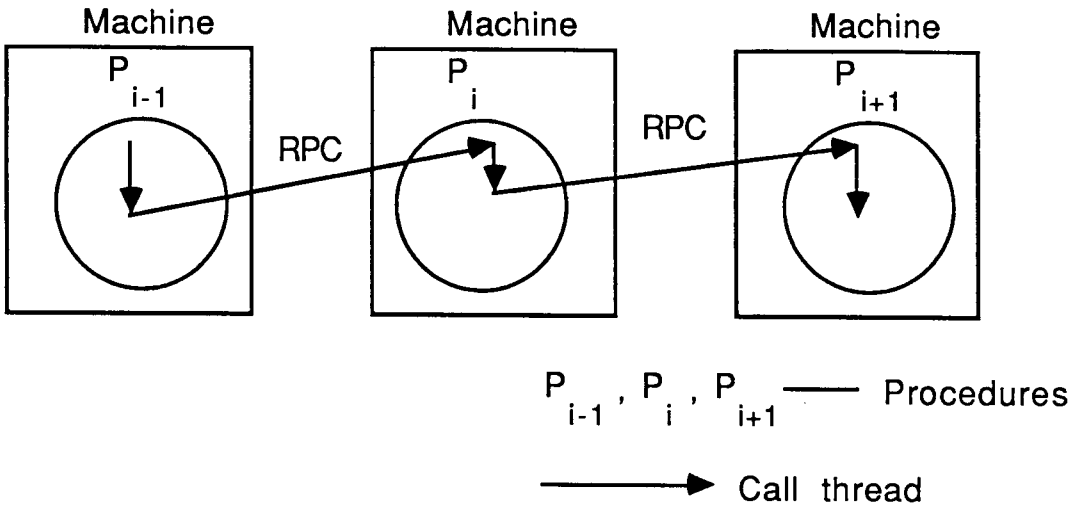


Figure 2.3: **Remote procedure call**

Failure transparency will be discussed in the context of these abstractions in the subsequent chapters of the thesis.

2.7.1 Remote procedure call

RPC is a high level communication abstraction by which a client may interact with a server on a different machine[6] [8]. RPC 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 remotely located procedures and system services much the same way as local procedures.

Refer to Figure 2.3. The P_i's are the processes (or procedures) in the program. Suppose P_{i-1} calls P_i which in turn calls P_{i+1}, then P_{i-1} is the client (also referred to as the caller) of P_i and P_i is the server (also referred to as the 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

[6] In our prototype implementation, RPC is realized using the low level message-passing (over an Ethernet) provided by the V kernel [14]. Similar implementation has been made elsewhere [3].

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 access to an extrinsic 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. Additionally, contentions by the server to access an intrinsic 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 in conjunction with RPC for access to extrinsic resources, or independently for access to intrinsic resources.

2.7.2 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 2.2) 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.

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 create groups, join and leave them [2,16]). In such a realization, the 'address' which 'points' to a shared variable V may be the group id of the server group whose members share the variable. The details of the protocols used to implement the operations are given later in the chapter 5. 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 (if it is not a member of the group that manages V) and acquires the state of the shared resource.

val = Read(V). The operation returns the value of the variable V in *val*. Note that the 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.

Write(V, val). The operation writes the value *val* into the variable V. Procedurally, the member may write 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 (e.g., not already locked); otherwise it is queued up waiting for V to become allocatable (e.g., when the current holder of V releases it). 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 collisions, i.e., 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.

The **Lock** and the **Unlock** operations are similar in semantics to the **P** and **V** operations on semaphores [41]. However, as we shall show later, simple arbitration mechanisms that do not guarantee any specific ordering among the operations are 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[7]. In this sense, they are weaker than the **P** and the **V** operations where a well-defined arbitration order (such as FIFO or LIFO) is usually specified as part of the semantics.

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 chapters.

2.8 Failure semantics of RPC

Refer to Figure 2.4. 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

[7] Concurrency control and serializability are not addressed in the thesis.

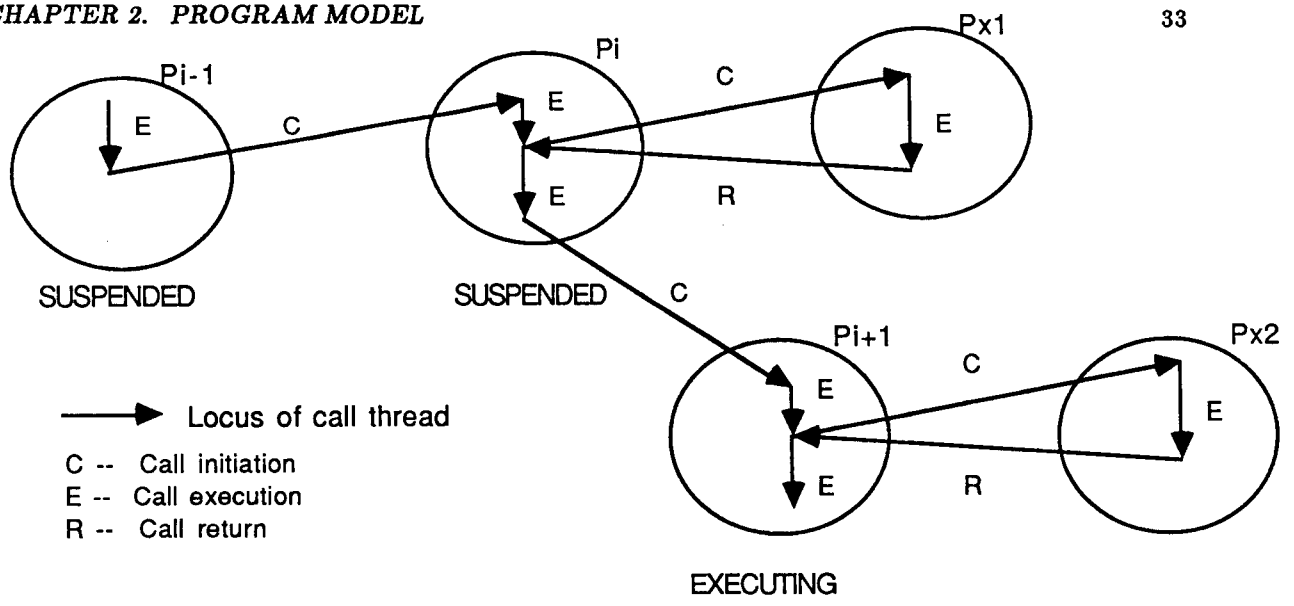


Figure 2.4: Locus of the remote procedure call thread

to change states. We refer to the state of all such servers as the state of the (execution) 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:

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

$$CALL\_FAIL(TR) \equiv (\underline{X}, \underline{X}) \tag{2.2}$$

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[31].

It means that the overall effect of the call should be all-or-nothing[8].

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 procedure 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 procedure, then the failure semantics requires that P_{i-1} sees the outcome $CALL\_FAIL(TR)$.

The failure semantics implies two activities on the part of the run-time system — (i) detecting the failure of P_i, and (ii) 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 just detecting the failure of P_i because P_i does not maintain any state. In the discussion that follows, we assume the availability of some mechanism, such as that presented in appendix A, by which the failure of P_i may be detected.

2.8.1 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} — $\underline{X}$ and $\underline{X'}$ are the state of the environment when TR was initiated and when P_i failed respectively. 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

[8] In sequential programs, recoverability of the call is sufficient to guarantee call atomicity.

back the state of the environment from $\underline{X}'$ to $\underline{X}$ to provide the required $CALL\_FAIL(TR)$ outcome. This requires, among other things, killing the orphan P_{i+1}[30,35]. In general, if the failure of a procedure 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 [8,56] and killed. 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[53].

2.8.2 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 foreign system, i.e., a system that does not support our RPC model). In some applications, rollback may not be meaningful [56,34] such as rolling back a print operation. 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. As another example, consider a remote login connection to a foreign system (e.g., UNIX); rollback on the connection may affect the foreign system, and the latter may not even support rollback in its domain; even if rollback is supported, the semantics may be different in the two systems. Even when rollback is possible, it may be so expensive as to be impractical. 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. When such an outcome is delivered, the caller should be aware that the state of the environment may

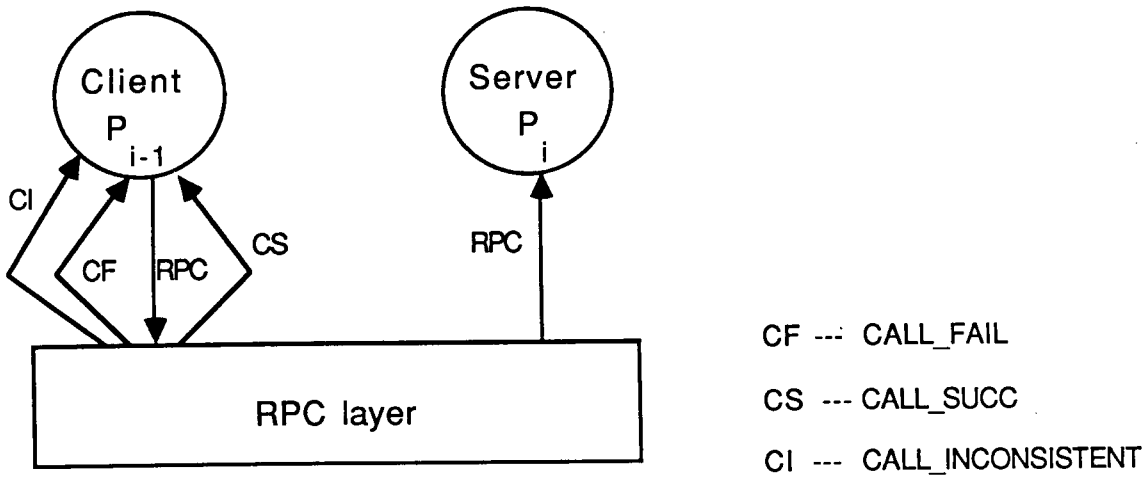


Figure 2.5: **Communication exceptions delivered to applications**

be inconsistent.

2.8.3 Communication exceptions

Consider the call TR on P_i initiated by P_{i-1}. The $CALL\_FAIL(TR)$ and $CALL\_INCONSISTENT(TR)$ outcomes of TR, when they occur, are delivered to P_{i-1} as *communication exceptions*. See Figure 2.5. The run-time system may provide built-in handlers (that may be compiled into P_{i-1} — see section 2.10) to deal with the communication exceptions. The handlers usually abort the program using an appropriate technique to propagate the exceptions across machines. For a detailed discussion on exception propagation techniques, see Stella Atkin's thesis [4]. Needless to say, the run-time system should minimize the occurrence of such exceptions to provide a high level of failure transparency.

Since the semantics of the communication exceptions are well-defined, the caller P_{i-1} may optionally deal with the exceptions in an application-dependent manner. To do so, P_{i-1} may provide hooks into its own exception handlers to trap the exceptions and deal with them. Suppose, for example, P_i is a mail server and P_{i-1} initiates TR on P_i to send mail to a remote site. If TR fails with the $CALL\_FAIL(TR)$ outcome (say P_i fails after preparing the mail but before sending it), P_{i-1} may trap the exception into

an exception handler, deal with the exception, say by skipping the send operation, and continue the execution. If TR fails with the $CALL\_INCONSISTENT(TR)$ outcome (say P_i fails after sending a portion of the mail), P_{i-1} may deal with the exception by sending a request for cancellation of the garbled mail. As another 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[9]. If a call TR fails with the $CALL\_FAIL(TR)$ outcome, P_{i-1} may deal with the exception, say, by tolerating the failure and hoping to correct the time at the next call.

2.9 Failure semantics of ADSV

Recall that we use the concept of ADSV in the context of a server group. This section outlines the failure semantics of ADSV specific to the group. The implications of the failure of a member of the group 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 (see section 5.3). For extrinsic resources, the lock recovery becomes part of a rollback activity that may be initiated by the member if its client fails (refer to section 2.8.1). 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[10]. This view constitutes the shared variable of the group. If it is required that all members have a consistent

[9]Note the calls are connection-less.

[10]In some applications, it may be sufficient for a member to know only about its two neighbors (members occupying adjacent positions) in the ranking.

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.

2.10 Flow of information between the application and communication layers

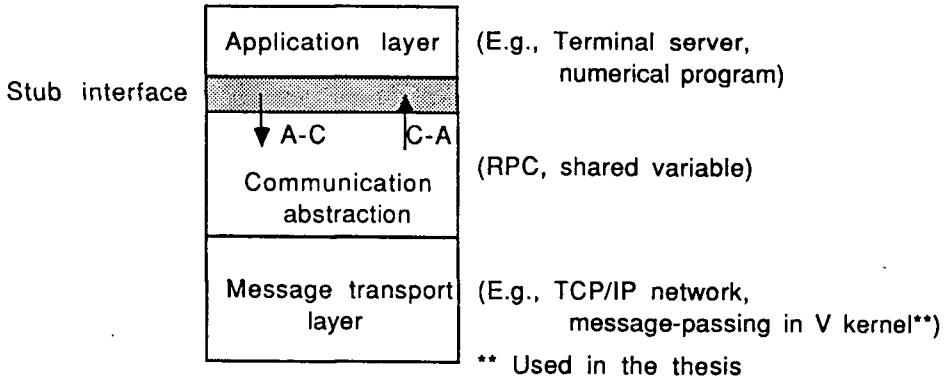
We described in the previous sections how the semantics of communication abstractions (RPC and shared variable) may reflect certain application-level properties. We now describe how the information about the properties may permeate into the communication layer to realize the abstractions.

With reference to Figure 2.6, the application layer exchanges information with the communication layer through a *stub interface* which consists of a set of stub procedures (or simply stubs). The stubs interface between a language level invocation of an IPC and the underlying communication layer[11]. The system programmer who implements a server makes static declarations about certain properties of the server, including: i) the idempotency properties of the various operations supported by the server, ii) the resource type (e.g., name binding information, leadership in a group), and iii) whether the resource is intrinsic or extrinsic if it is shared. These declarations are used by a pre-processor for the language in which the server is implemented to generate the appropriate stubs (c.f. section 6.2.3). The stubs form part of the executable images of the client and the server that run on the various machines.

The communication layer obtains the application-level information from the stubs during run-time and structures its internal algorithms and protocols accordingly as described in chapters 3, 4 and 5.

As pointed out earlier, unrecoverable failures in the communication layer are delivered to the stubs as exceptions which are then dealt with by either the built-in handlers in the stubs or the user supplied handlers hooked to the stubs.

[11] The system programmer who implements the communication layer also provides the stubs.



C-A --- Flow of communication exceptions

A-C --- Flow of information* from the application layer to the communication layer

* Typical information:

1. Idempotency properties of calls
2. RPC type --- Connection-oriented / connection-less
3. Type of shared resource (e.g., leadership, name binding information, distributed load information)

Figure 2.6: Interface between application and communication layers

2.11 Summary

In this chapter, we have presented a model of a distributed program based on application-level characteristics. The model forms the basis of the various recovery techniques discussed in the following chapters. A program is viewed as consisting of client-server interactions interspersed with intra-server interactions. A client-server interaction may be connection-oriented or connection-less depending on the ordering relationship it should maintain among a sequence of such interactions. Its idempotency and determinism properties were described. An intra-server interaction deals with a shared resource. The resource may be intrinsic or extrinsic depending on whether access to the resource is asynchronous to client interactions or not. RPC is identified as a suitable communication abstraction to map onto client-server interactions. We have introduced a different communication abstraction, which we refer to as ADSV (application-driven shared variable), to map onto intra-server interactions and to encapsulate

the underlying consistency issues. We have specified the failure semantics of RPC and ADSV, and introduced the notion of communication exceptions in distributed programs.

Our 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. As we shall see in the later chapters, 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.

Chapter 3

Reconfigurable execution of RPC

Chapters 3 and 4 discuss failure transparency in client-server interactions with RPC being the underlying communication abstraction.

In this chapter, we describe a scheme to mask the failure of a server in which one of the replicas of the server, known as the primary, executes a client call while the other replicas, known as secondaries, are standing by. When the primary fails, failure recovery requires one of the secondaries to reconfigure as the primary and continue the server execution from the point where the erstwhile primary failed. Such a reconfiguration, also referred to as the *re-incarnation* of the failed procedure, allows the program to continue functioning despite the failure. The scheme uses a new recovery technique which we refer to as *orphan adoption* to deal with orphans. The run-time system uses event logs and call re-executions to realize orphan adoption; rollback is used only where essential. The idempotency properties of the calls are used in the recovery algorithms and protocols. The adoption technique saves any work already completed and minimizes rollback which may otherwise be required. Finally, we compare the orphan adoption technique with recovery techniques proposed elsewhere.

3.1 Failure masking based on orphan killing

Refer to Figures 2.4 and 2.5. Consider the failure scenario described earlier in section 2.8, 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, then the recovery of P_i is transparent to P_{i-1} and P_{i+1} if, when the re-incarnated call thread reaches the point where P_i failed, the state of the environment is $\underline{X'}$ and the state of P_i is the same as its pre-failure state. The conventional way to achieve this transparency is to rollback the environment state from $\underline{X'}$ to $\underline{X}$ before the re-incarnated P_i starts execution (c.f. section 2.6.2). This requires, among other things, killing the orphan P_{i+1} [30,35].

At this point, we wish to distinguish between the requirement for the rollback described above (based on orphan killing) and that proposed in section 2.8.1:

1. The rollback described above is part of the failure masking technique based on orphan killing, so whenever a failure occurs, rollback is employed to mask the failure.

2. The rollback described in section 2.8.1 is needed only to provide the $CALL\_FAIL(TR)$ outcome. Thus, it is used only if the failure cannot be masked. If rollback is not possible, then the run-time system may simply provide the $CALL\_INCONSISTENT$ (instead of $CALL\_FAIL$) outcome. On the other hand, if the $CALL\_FAIL$ outcome is not part of the failure semantics, then rollback is not required at all.

Thus, rollback is more frequent and critical in the first case than the second. These differences influence the choice of the recovery technique and the failure semantics.

3.2 Failure masking based on orphan adoption

Orphan killing may be undesirable because the rollback required may be quite expensive or not possible (c.f. section 2.8.2); secondly, orphan killing wastes useful work that has been done. These considerations motivated us to propose an alternative technique based on orphan adoption. The idea that underlies the technique is the following:

> Suppose a procedure fails and recovers. Whenever possible, killing of the orphans (and hence rollback) resulting from the failure should be avoided. In other words, orphans are allowed a *controlled survival* rather than being indiscriminately killed. Such surviving orphans are subsequently adopted by the recovering procedure.

The adoption may occur when the recovering procedure re-executes from its restart point to the *adoption point* (typically the point where the procedure failed) in the same state as the failed procedure. During the re-execution, the recovering procedure (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 2.6.3 and 3.3). The re-execution of the recovering procedure allows it to *roll forward*, and may be categorized as a forward error recovery technique [9].

Referring to the failure situation described in section 3.1, 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 $CALL\_INCONSISTENT$ is delivered.

The run-time system effects a roll forward based on two ideas (refer to Figure 3.1):

1. Controlled re-execution of the calls, if necessary, based on their idempotency properties.

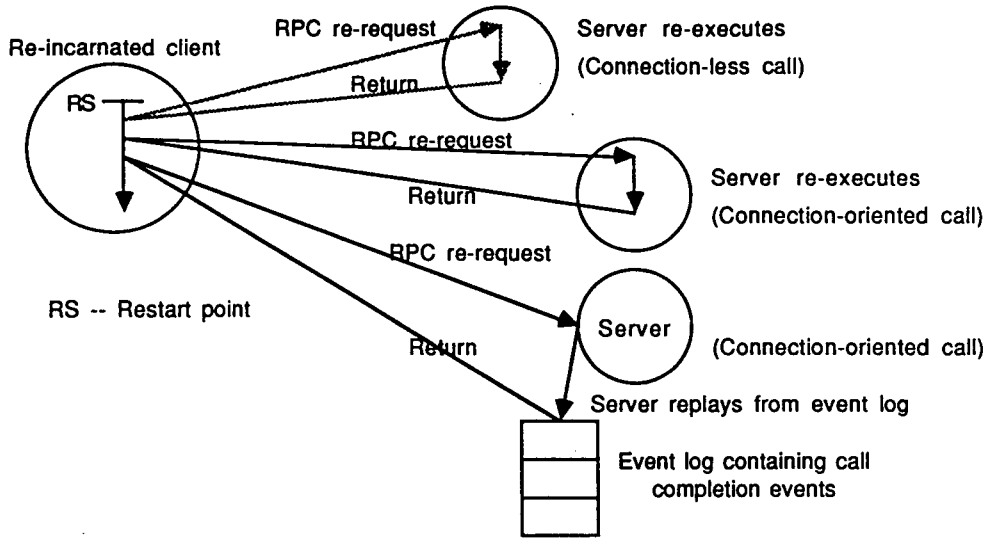


Figure 3.1: **Recovery of a re-incarnated procedure**

2. Event logs that contain call completion events allow a recovering procedure to get in step and become consistent with other procedures without actually re-executing the calls (see section 3.3.3).

Since the concept of call re-executions is central to the orphan adoption scheme, it is examined in the following section. The concept is relevant also in dealing with message orphans (see section 3.4.2) and in the context of replicated execution of a server (see chapter 4).

3.3 Call re-executions

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 call TR^i is represented as

$$(C_{i-1}, S_{i-1}) \stackrel{TR^i}{\rightarrow} (C_i, S_i), \tag{3.1}$$

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, i.e., $TR^k \succ TR^{i'}$. Then the call sequences EV_SEQ and $[EV\_SEQ \succ TR^{i'}]$ are not ordered sequences with respect to one another (refer to section 2.1). 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 and the determinism properties associated with the call (c.f. sections 2.6.1 and 2.6.2) as described in the next section.

3.3.1 Interfering calls

Refer to the example in the previous section. 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 (3.1)) that

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

Because the program is deterministic in behavior, the first requirement is satisfied. Thus the effect of $TR^{i'}$ may be given by

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

Pattern matching this relation with (2.1), 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

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

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 (c.f. sections

3.4.2, 4.6.2 and 4.7.4).

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 lends insight into other commutative properties of calls as well such as ordering of calls, missed calls and interspersing of calls from multiple clients on the server. These properties are more relevant in the context of replicated execution of servers (see chapter 4), and are discussed in section 4.3.1.

3.3.2 Call identifiers

To allow a server to determine whether a call can be re-executed, the call is identified by a global call identifier *G_tid* and a non-idempotent call identifier *nI_tid*. These id's are based on sequence numbers. They are assigned by the client, and are maintained by the server as part of the data structure used for the connection. *G_tid* is assigned the next sequence number for every new call on the connection, while *nI_tid* is assigned the next sequence number for every new non-idempotent or 1-idempotent call (c.f. sections 3.4.2, 4.6 and 4.7). Using these id's, the server may check for the interference condition derived earlier in section 3.3.1, and conditionally re-execute calls.

3.3.3 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 the call id and 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 (refer to 3.1)

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

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

In the following sections, we describe the algorithms and protocols used by the run-time system to realize orphan adoption. Only those essential to describe the adoption technique are given here. Other details of the implementation may be found in [44].

3.4 Failure recovery algorithms

Refer to Figure 2.4. The procedure P_i[1] 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 (i.e., no thread is passing through P_i) otherwise[2].

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)) should P_i fail because P_{i-1} has the name binding information for P_i (c.f. section 3.4.2) that is necessary for failure recovery. Note that P_i acts as the primary among its replicas.

When P_i completes a non-idempotent 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. Our choice of P_{i-1} as the checkpoint site is a design decision based on two reasons: i) 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. ii) Since we assume a system environment which may consist of diskless machines (c.f. section 1.6.1), P_i may not have a local disk to use as stable storage, so the buffer space in the client's (i.e., P_{i-1}'s) run-time system is a suitable choice for checkpointing (see section 6.2.4). This permits recovery in the event P_i fails.

[1] Actually, we refer to the run-time system at P_i's site.

[2] The IDLE state is not needed for connection-less procedures.

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 [6] (c.f. section 3.7).

Suppose P_i fails, RI(P_i) detects the failure using a *death-will* abstraction that allows a process to notify its failure to another process it is communicating with. The notification is done by a message whenever a process fails or observes a communication break-down. The message enables all processes communicating with the failed process to detect the failure (see appendix A).

After detecting the failure, RI(P_i) 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, selection is based on which of the secondaries responds first to a message broadcast by RI(P_i) to locate the new primary. After the selection, RI(P_i) initializes the new P_i to the state last checkpointed in its site, and then 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 a special type of 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 pre-failure 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 based on its pre-failure state.

3.4.1 RESTART activity

RI(P_i) restarts the new P_i which then starts (re-)issuing the calls embedded between the last checkpoint to the point where the erstwhile P_i failed (see Figure 3.1). 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

re-executing the call if this will not cause state inconsistencies (see section 3.3.1). If the server has all the calls sent to it in its log, no re-execution of the calls is necessary. Ideally, the size of the log should be large enough to retain all the calls since the last checkpoint[3]. However, the finite size of the log in any implementation means there is a possibility that a non-idempotent call cannot be logged by the server (during the call return) as the log is full. We consider the following options in handling this problem:

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 (non-idempotent) call P_i itself 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 non-idempotent calls arriving after the log is full. 2) The state checkpointed need 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.

Option 2: Rollback of the unlogged call

The implications of the server being unable to log the non-idempotent call it returns are as follows: If the client (P_i in our case) fails and recovers, the calls which are re-issued from the re-incarnated 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 non-idempotent call should be rolled back before P_i can re-issue the calls. Assuming the run-time system already maintains data structures to support rollback and provide the *CALL_FAIL* outcome, the rollback of the unlogged call does not require any

[3]The size of the log may be chosen depending on the type of the resource implemented by the server and other factors such as buffer availability and data size.

additional data structures. However, as described earlier, rollback has adverse effects under certain situations. If the rollback cannot be carried out, P_i cannot roll forward whereby it may fail the call it executes by delivering the *CALL_INCONSISTENT* outcome to P_{i-1}.

The run-time system designer may choose one of the above options after weighing their implications in the 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 available any way to provide the *CALL_FAIL* outcome, so any rollback of unlogged calls may be realized without additional data structures in the run-time system.

Connection-less calls

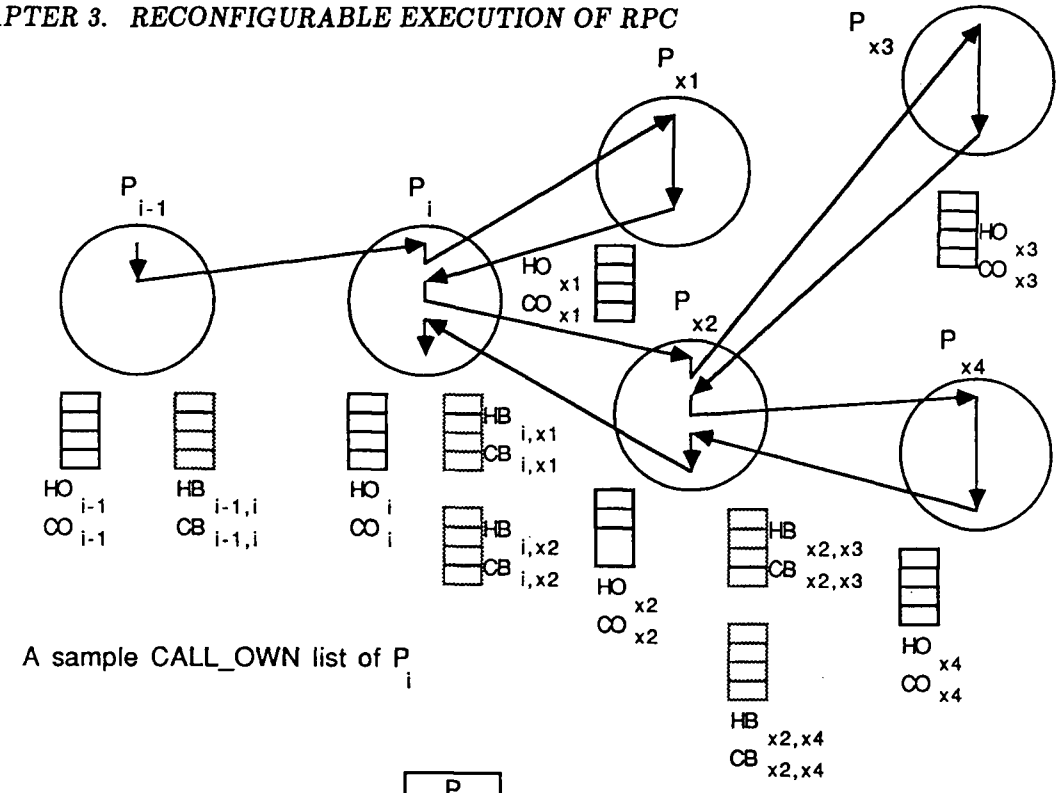
The connection-less calls on a server are not logged by the server, so these calls when re-issued by the new P_i are invariably re-executed by the server. Also if a server fails during a connection-less call on it, the client simply re-issues the call on another replica of the server. Since our program model encapsulates connection-less calls, such recoveries are required in our algorithm. In this aspect, the algorithm is distinct from that used elsewhere (see section 3.7).

3.4.2 RPC data structures and protocols

The run-time system maintains a set of variables and data structures for recovery purposes (see Figure 3.2). Only those essential to describe the adoption technique are given below:

CALL_REF($\mathbf{P_i}$, $\mathbf{P_j}$): It is a 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[4]. 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.

[4]In other words, CALL_REF contains name binding information.



A sample CALL_OWN list of P_i

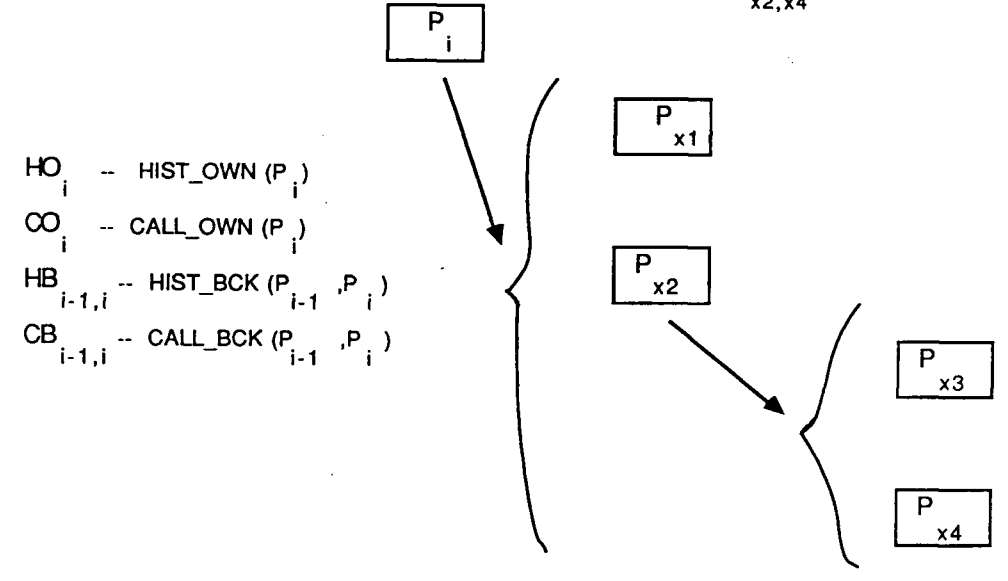


Figure 3.2: Data structures used in the RPC run-time system

($G\_tid_{rqst,i,j}$, $nI\_tid_{rqst,i,j}$): It is the call id pair maintained by P_i (as a client) for 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 had 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 *non-idempotent* 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 re-execution 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 variable PV_i maintained by P_i[5] and its thread state; a value is stored when P_i completes a non-idempotent call. It constitutes the history

[5] PV_i may be represented in a machine-independent form by using external data representation techniques [24,1].

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) should P_j fail and recover.

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 call requests and returns, see [44]. We describe below only the call validation phase of the RPC protocols.

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 re-issued 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 re-issued call. This may occur either because P_i is a recovering procedure or the request is a message orphan (c.f. section 1.5). If the call may be honored by P_{i+1} from its event log, it replays the appropriate completion 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 re-execute the requested call and return the results to P_i. If the call is non-idempotent 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 roll-back. If P_{i+1} rolls back, P_i may re-issue 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 non-reusable (see section 5.7).

3.4.3 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 can 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.

3.4.4 Recovery of P_i

If P_i was EXECUTING when it failed, RI(P_i) initiates the rollback activity (see the 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 2.6.2). If the rollback completes successfully but the INIT is unsuccessful[6], 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 well. RI(P_i) which carries out the INIT activity to re-incarnate P_i, should then co-ordinate the RESTART of P_i with the handling of the orphan P_{i+1}. Orphan handling consists of two steps – orphan detection and orphan adoption. P_{i+1} (in the EXECUTING or in the SUSPENDED state) detects that it is an orphan on detecting the failure of its caller P_i.

Orphan adoption algorithm

The orphan adoption algorithm has two components:

1. Determining the adoptability of an orphan — an orphan is adoptable if its continued existence in the system does not interfere with the recovering procedure.

2. If the orphan is adoptable, then the adoption protocols are invoked, otherwise the environment state is rolled back to provide the *CALL_FAIL* outcome[7].

The idea behind the adoption protocols is as follows: On detecting the failure of P_i, the orphan P_{i+1} executes a *brake* algorithm to determine its adoptability. In this algorithm, if P_{i+1} finds that the

[6] A typical reason for unsuccessful completion of the INIT activity is the non-availability of a replacement procedure.

[7] The rollback is not necessary if the *CALL_FAIL* outcome is not required.

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 ($P_{i+1}, P_{i+2}, \ldots$) to suspend the tip of the orphaned thread. This phase is followed by a rollback along the orphan chain if necessary. If the orphaned thread will not interfere with the recovering thread, the orphan is allowed to continue. 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. We now present the details of the adoption protocols.

Let P_j ($j \geq i+1$) be a callee in the orphaned call chain. When P_{i+1} detects the failure of P_i, it generates a BRAKE_RQST event. For each of the other P_j's, the event occurs on arrival of a BRAKE message from its immediate caller P_{j-1}. P_{i+1} generates a BRAKE_COMPLETED event upon completion of the brake algorithm (see the **BR_DN** and the **BR_UP** algorithms given below). Upon observing the BRAKE_COMPLETED event, RI(P_i) may initiate its part of the recovery, namely performing any rollback required and restarting P_i (see sections 3.4.3 and 3.4.4).

A procedure P_j maintains three boolean variables. When true, the flags have the following meanings:

brake_flag(P_j): Brake condition is set at P_j.

adoption_flag(P_j): Adoption is still to be completed at P_j.

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

P_j is an orphan if the **brake_flag(P_j)** or the **adoption_flag(P_j)** is true.

Brake algorithm downstream (BR_DN)

P_j detects that it is an orphan upon occurrence of the BRAKE_RQST event[8]. It sets **brake_flag(P_j)** and **adoption_flag(P_j)** to true. The **brake_flag(P_j)** is set to false by a brake release event (BRAKE_RELEASE) which is generated when the recovering P_i starts the adoption algorithm; the event is detected by the arrival of an ADOPT message at P_j. The **adoption_flag(P_j)** is set to false when the adoption is completed, as indicated by the arrival of an ADOPT_ACK message at P_j (see the **AD_DN** algorithm given later in this section). If P_j should send the BRAKE message to its callee P_{j+1}, it piggybacks a bit given by

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

on the message, where **last_nI_call($\mathbf{P_j}$, *X)** is a function that operates on CALL_OWN(P_j) and returns the global call id of the last non-idempotent 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 j = i+1, i.e., the first procedure P_{i+1} in the orphaned call chain. P_{i+1} checks **CALR_ENV_INTRCT(P_{i+1})**, which may have one of two values:

False: P_{i+1} is allowed to continue (concurrently with the re-incarnated P_i) irrespective of whether the call is idempotent or not, because no call originates from the re-incarnated P_i (between the start and the failure points), and hence P_i does not interfere with P_{i+1}'s execution. P_{i+1} also generates the BRAKE_COMPLETED event.

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., $j \geq i + 1$).

The orphaned call on P_j may be idempotent or non-idempotent:

[8] For P_{i+1}, generation and observation of the BRAKE_RQST event collapse into a single activity.

- **Idempotent**

Suppose **cum_clr_rl_flag**(P_j) is false, i.e., no rollback is required by any P_k ($i \leq k \leq j-1$) when the failed P_i recovers. P_j may continue to execute (concurrently with P_k) since the call is idempotent[9] and there is no pending rollback that may interfere with the execution of the call. P_j sends a BRAKE_ACK message up the call chain[10] (see the **BR_UP** algorithm given later).

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 is applied to its execution and a BRAKE_ACK message is sent to its caller P_{j-1} in the call chain, where it invokes the **BR_UP** algorithm. If P_j is in the SUSPENDED state, it sends a BRAKE message to the next callee P_{j+1} down the chain. This message, upon arrival at P_{j+1}, causes the BRAKE_RQST event and invokes the **BR_DN** algorithm at P_{j+1}.

- **Non-idempotent**

The actions taken depend on P_j's current state of execution:

1. SUSPENDED state – P_j sends a BRAKE message to P_{j+1} down the call chain to invoke the **BR_DN** algorithm.
2. EXECUTING state – P_j immediately applies a brake to its execution and executes the **BR_UP** algorithm.

- *End of* **BR_DN algorithm**

- **Brake algorithm upstream (BR_UP)**

[9] A non-idempotent call may embed both idempotent and non-idempotent calls in it. However, an idempotent call may embed only idempotent calls.

[10] For P_{i+1}, sending a BRAKE_ACK and generating a BRAKE_COMPLETED event collapse into a single activity.

Suppose P_j receives the BRAKE_ACK message as the message traverses up the call chain. If the call on P_j is idempotent, P_j simply passes the message on to P_{j-1} up the call chain. If the call is non-idempotent and if CALL_OWN(P_j) contains at least one returned entry, P_j completes the rollback algorithm, sends the BRAKE_ACK message up the call chain, and waits for the BRAKE_RELEASE event. The BRAKE_ACK message arriving at P_{j-1} invokes the **BR_UP** algorithm at P_{j-1}.

- *End of* **BR_UP algorithm**

Upon occurrence of the BRAKE_COMPLETED event (which implies that P_{i+1} has completed any required rollback up to the point where the failed P_i was suspended), 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 dispatching an ADOPT message down the orphaned thread thereby 'stitching' the latter with the recovering thread. The activity of thread stitching is described below:

Thread stitching

The arrival of the ADOPT message at P_j ($j \geq i+1$) invokes the **AD_DN** algorithm given below.

- **AD_DN algorithm**

The first step in adopting P_j is to release any brake (BRAKE_RELEASE event) that has been applied to the orphan execution at P_j, i.e., the **brake_flag(P_j)** is set to false and the execution is resumed from the point where the brake was applied earlier. Consider j = i+1, i.e., the first procedure (P_{i+1}) in the

orphaned call chain. P_{i+1} checks its CALR_ENV_INTRCT(P_{i+1}) and carries out the operations given below:

False: Execute the **AD_CON** algorithm (given below) to adopt the concurrently executing P_{i+1}.

True: (The rest of the algorithm applies to all procedures down the orphaned call chain, i.e., $j \geq i+1$.)

The following cases are possible:

- P_j **is idempotent** The **AD_CON** algorithm is executed to adopt the concurrently executing P_j.
- P_j **is non-idempotent** The actions depend on the state of P_j:

1. EXECUTING state – P_j sets **adoption_flag(P_j)** to false and sends an ADOPT_ACK message up the call chain.
2. SUSPENDED state – The adoption point is the point where P_j got suspended. When the (re-)execution thread reaches the adoption point, an ADOPT message is sent to P_{j+1} down the orphaned call chain.

A procedure P_j that receives the ADOPT_ACK message sets its **adoption_flag(P_j)** to false and passes the message onto P_{j-1} up the call chain.

- *End of* **AD_DN algorithm**

AD_CON algorithm (Adopting concurrently executing orphans)

In the **BR_DN** algorithm, the recovering caller P_k ($i \leq k \leq j-1$) may, under certain situations, execute in parallel with P_j. If P_j completes its execution first, it applies a brake to its execution at that point and awaits the BRAKE_RELEASE event, i.e., adoption by P_k, as indicated by the arrival of the ADOPT message from P_{j-1}, to resume execution. If the BRAKE_RELEASE event occurs first, P_j sets **brake_flag(P_j)** to false, as described earlier, completing the brake release phase.

Suppose the BRAKE_RELEASE event occurs at P_j's site. If P_j has already completed its execution (and is awaiting adoption), the call returns immediately to P_{j-1}[11]. In this case, or if P_j is in the EXECUTING state, P_j sets **adoption_flag(P_j)** to false and sends an ADOPT_ACK message to P_{j-1} for traversal up the call chain. Otherwise, the ADOPT message is sent to P_{j+1} down the call chain.

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 (c.f. section 5.3) and forces a CALL_FAIL or CALL_INCONSISTENT exception, as the case may be, to the client of the failed procedure.

3.5 Analysis of the RPC algorithm

We now provide a quantitative analysis of our failure recovery technique. We introduce two indices to characterize the recovery activities carried out by the run-time system. The extent of rollback required to recover from a failure is the criterion underscoring these indices.

3.5.1 Catch up distance

A *catch up distance* is defined for a caller-callee pair. It is the maximum number of calls the caller may make to the 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.

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

Let $TR^1, TR^2, \ldots, TR^i$ 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 re-execution of TR^1 by the caller violates idempotency requirements. If, on the other hand, TR^1 can be re-executed without rollback, then the entire sequence can be re-executed without rollback.

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

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

The mean size of the catch up distance $\overline{N}_{ctchup}$, 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

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

$\overline{N}_{ctchup}$ is a static characterization of the program under the given run-time system. Figure 3.3 shows the variation of $\overline{N}_{ctchup}$ 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 the figure, it is clear that the level of failure tolerance is higher when a server re-executes calls (based on the idempotency properties) than when it does not.

3.5.2 Rollback distance

Rollback distance is the number of non-idempotent client calls after the last checkpoint (call return in our case) whose effects a callee should rollback when the client fails and recovers[12]. Assume S calls have been completed by the client, and there is no on-going call. Suppose the client fails and then recovers.

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

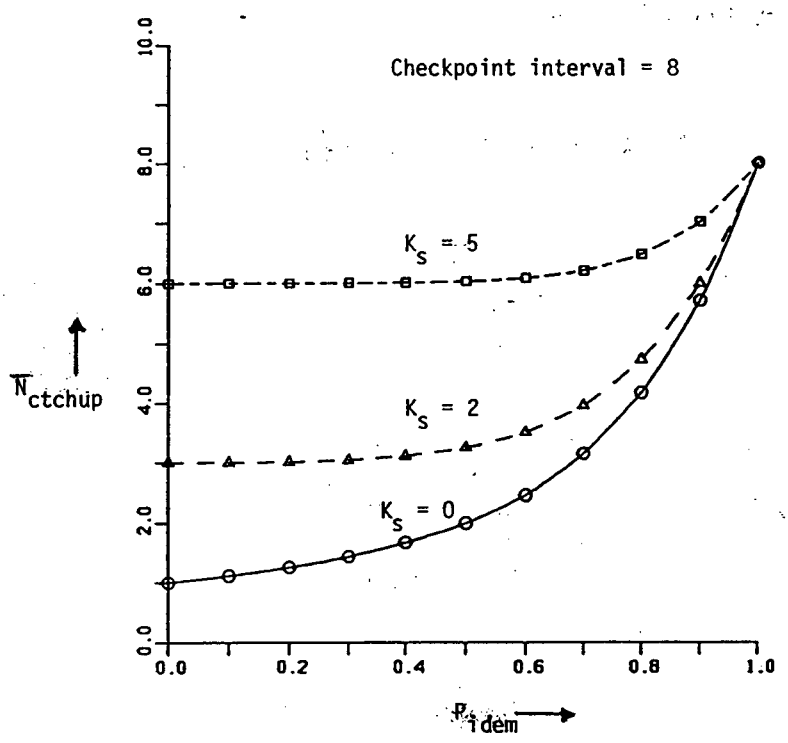


Figure 3.3: Variation of catch up distance with respect to P_{idem}

The probability that the rollback distance is R ($0 \leq R \leq (S - K_s)$) is given by

$$P_{rlbck,S}(R) = \begin{pmatrix} S-K_s \\ R \end{pmatrix} .(1-P_{idem})^R .(P_{idem})^{S-K_s-R} \qquad \text{for} \qquad S \geq (K_s+1)$$

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

$$\underline{R}(S) = \sum_{R=0}^{S-K_s} R. \begin{pmatrix} S-K_s \\ R \end{pmatrix} .(1-P_{idem})^R .(P_{idem})^{S-K_s-R}$$

The graphs in Figure 3.4 illustrate the variation of $\underline{R}(S)$ with respect to P_{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

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

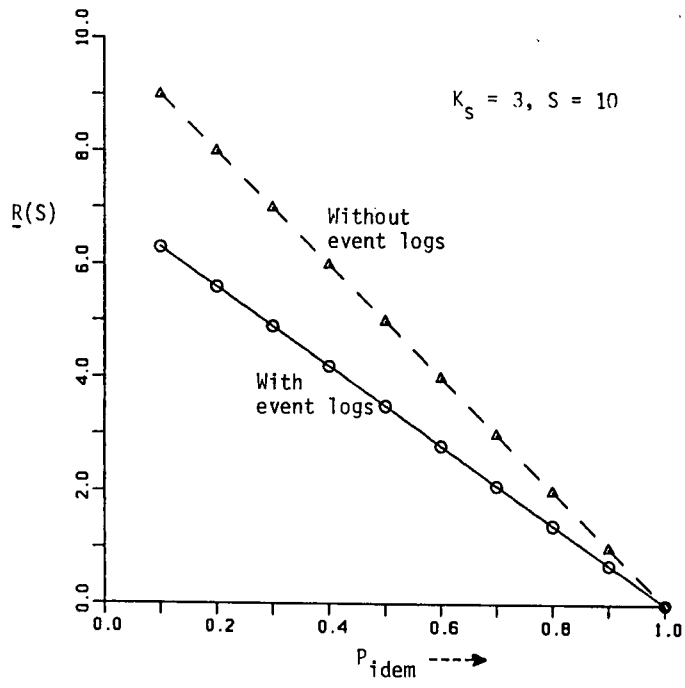


Figure 3.4: **Variation of rollback distance with respect to P_{idem}**

When no logging is done, i.e., $K_s = 0$, the probability is $(1 - (P_{idem})^S)$. The graphs in Figure 3.5 illustrate the variation of the probability of rollback with respect to S for a given P_{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 indices lend insight into the extent of rollback required for given checkpoint intervals.

We now give some indications about the performance of the orphan adoption technique in terms of the number of messages required for the various algorithms.

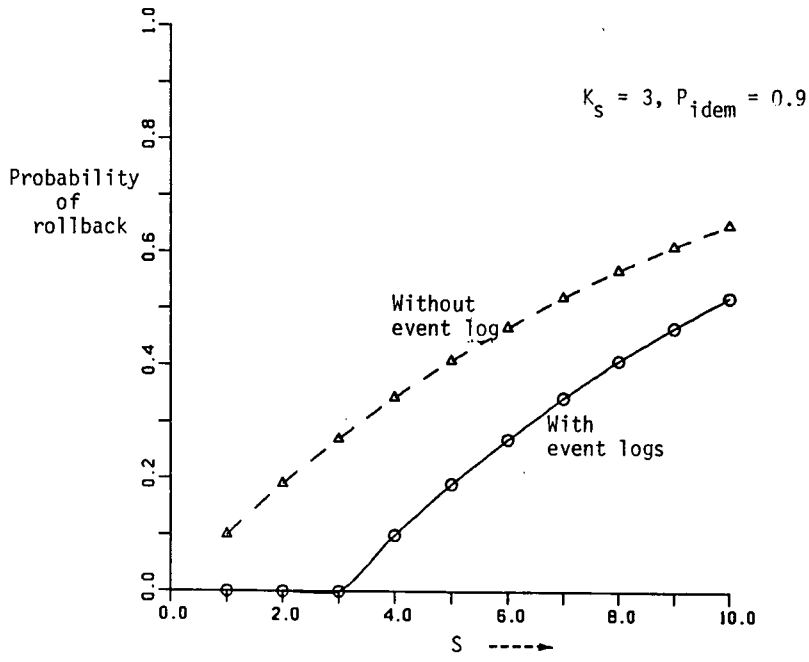


Figure 3.5: **Variation of the probability of rollback with respect to P_{idem}**

3.6 Performance indications

Prototypes of the RPC model have 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 (c.f. section 1.6.1). The performance indications are given 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.

3.6.1 Sending call request and call return

Refer to Figure 3.2. Suppose P_i makes a call on P_{i+1}. Sending the call request requires 3 messages: i) a message from P_i to P_{i+1} containing the call request and the call arguments, ii) 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 iii) an acknowledgement message from P_{i-1} to P_i. Returning the call requires 3 messages: i) a message from P_{i+1} to P_i containing the results of the call and the thread state of P_{i+1}, ii) a message from P_i to P_{i-1} to delete the checkpointed $CALL\_REF(P_i, P_{i+1})$, and iii) an acknowledgement message from P_{i-1} to P_i. In addition, the return of a non-idempotent 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, the 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. 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.

3.6.2 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 2 messages (in one or more segments). On completion of the recovery of P_i, 2 messages are required to notify the completion (one message for notification and the other for acknowledgement) 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 2 messages.

Suppose P_i was SUSPENDED when it failed. The brake algorithm requires 2 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 2 messages for the procedure.

3.7 Related works

We now compare our adoption technique with techniques proposed elsewhere and used in some experimental systems.

3.7.1 ISIS

In ISIS [6], one of the replicas of a server is designated to be the coordinator while the others act as cohorts. The coordinator is the one that actually executes the calls from a client. 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 re-issues 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 descriptors to retain results of the calls. In other words, connection 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. The recovery of such calls is simple in our technique — the calls are simply re-executed. Secondly, it is not clear that ISIS deals with an on-going call thread that may be orphaned due to a failure. Our technique deals with the orphan by using explicit algorithms to suspend the orphaned thread and adopt it by the recovering thread.

Besides these fundamental differences, there are other procedural differences as well. In ISIS, the checkpoints contain the instruction pointer and the execution stack in addition to the application level

state. In our model, the checkpoints need not contain them unless intermediate checkpoints are taken. As for the choice of checkpointing frequency, both ISIS and our technique are equally flexible. Concerning checkpoint sites, ISIS takes a checkpoint at all the cohorts, so each cohort needs to maintain a connection descriptor associating the checkpoints to the client. In our technique, checkpointing is done only at the client site. Since the client and the executing server maintain a connection descriptor any way, the checkpoint is easily associated with the descriptor. In ISIS, choosing the cohort to take over as the next coordinator is based on a pre-assigned static ordering among the replicas. The choice is not difficult if the coordinator alone fails. If one or more cohorts next to the coordinator in the ordering also fail, the choice requires running consensus protocols among the rest of the cohorts. In our technique, the choice is dynamically made by the client upon occurrence of a failure and no specific ordering among the servers is necessary (c.f. section 3.4). Hence no message exchange among the servers themselves is needed irrespective of the number of servers that may fail. Thus the technique handles failures of multiple servers just as easily.

3.7.2 DEMOS/MP

In the publishing model of computation used in DEMOS/MP [42], checkpoints are established for every process periodically 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 very low level (the central site monitors the broadcast network). For this reason, there is no need to distinguish between connectionless and connection-oriented calls as far as failure recovery is concerned. 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. Secondly, it requires the abstraction of a reliable broadcast bus because every message put on the bus (sent or received by a process) needs to be logged by the central site. It is not clear how such an abstraction may efficiently be realized. Our technique, in contrast, is driven by application level requirements. It works at a much higher level of abstraction.

3.7.3 ARGUS

ARGUS is a distributed programming language supporting *guardians* and *atomic actions* whereby client guardians can invoke atomic actions on server guardians [31]. 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 that of ARGUS.

3.7.4 Lin's model of RPC

Lin provides a model of RPC which ensures call atomicity by orphan killing and rollback [30]. Though his notion of atomic and non-atomic calls is similar to that of non-idempotent 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 that of Lin.

3.8 Summary

In this chapter, we described a scheme to mask the failure of a server in which one of the replicas of the server acts as the primary at any time and executes client calls while the other replicas stand-by as secondaries. When the primary fails, failure recovery is initiated whereby one of the secondaries reconfigures as the primary to continue the server execution from the point where the erstwhile primary failed. We introduced a new recovery technique based on adopting the orphans rather than killing them.

The run-time system uses event logs and call re-executions (based on the idempotency properties of the calls) to realize orphan adoption. Rollback is used only where essential. By suitable choice of the log size in the system, rollback can be completely avoided. The adoption technique saves work already completed. We also introduced quantitative indices to evaluate the technique and compared it with techniques proposed elsewhere.

The RPC model incorporating the adoption technique has been implemented as a prototype on top of the V distributed kernel running on a network of SUN workstations interconnected by an Ethernet. Based on the implementation, some indications about the performance of the technique in terms of the number of messages required by the various algorithms were presented.

Chapter 4

Replicated execution of RPC

In this chapter we describe a different scheme to mask the failure of a server whereby the call from a client on the server may be executed at the same time by more than one replica of the server each residing on a different machine. In other words, the call initiates a thread that may propagate on all the replicas simultaneously and is referred to as a *replicated remote procedure call* (RRPC) [17]. There is no primary-secondary relationship among the replicas, instead they are equals. When any of the executing replicas fails, the call may be completed by the surviving replicas without explicit failure recovery initiated by the run-time system. The failure is then transparent to the client if the *replicated execution* of the server is itself transparent. A major requirement underlying this scheme is that the replicas have identical state at the end of each call — we shall use the term 'same-end-state' to denote this condition. We describe a new technique whereby the requirement is satisfied by distributed coordination among the replicas. The coordination requires: 1) Logging of call information by the replicas and communication among them to exchange the logged information, and 2) Controlled re-execution of calls (by servers) based on their idempotency properties. We also introduce a quantitative index to characterize the coordination.

In the next section, we describe the general characteristics of RRPC and the issues arising due to the replicated execution of the server caused by RRPC. In section 4.2, we briefly describe Cooper's model of RRPC which deals with the issues without any application-level information. From section

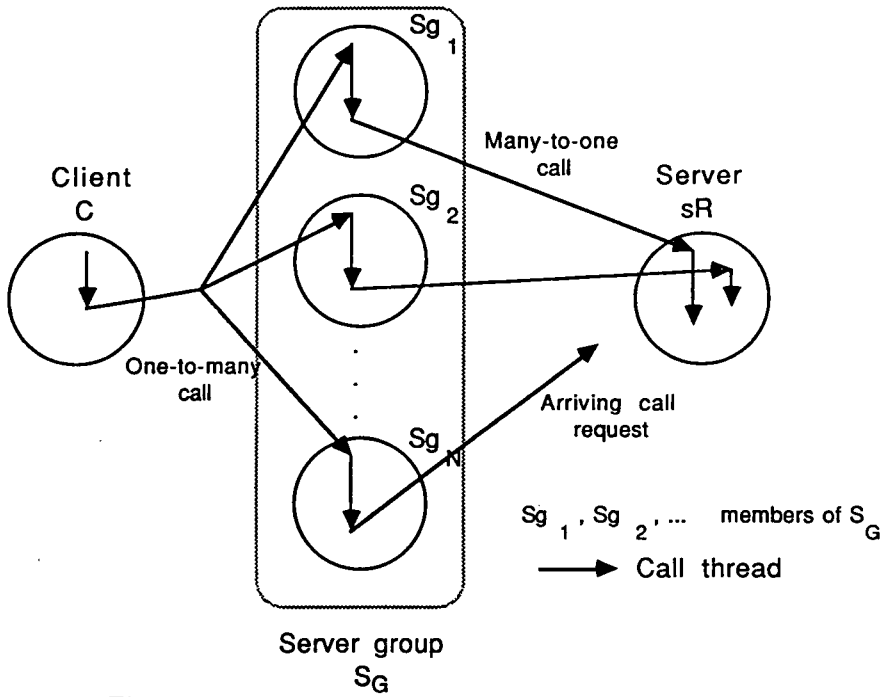


Figure 4.1: Basic components of a RRPC

4.3 onwards, we describe our model of RRPC which makes use of application-level information to deal with the issues. We present algorithms and protocols underlying the model to coordinate and maintain 'same-end-state' among the replicas.

4.1 Replicated remote procedure calls

A replicated remote procedure call on a server causes an execution by one or more replicas of the server. Two types of calls — a *one-to-many* call and a *many-to-one* call — form the basic components of the RRPC. With reference to Figure 4.1, suppose C makes a call on the server S_G which in turn makes one or more calls on the server sR. The call from C on S_G constitutes a one-to-many call in which the call event triggered by C may cause an execution at each replica of S_G (let N be the number of replicas). A call on sR from the replicated executions of S_G constitutes a many-to-one call in which the call event triggered by each of the executions in S_G may occur at sR. When the one-to-many call completes, C should perceive the effect of a single logical call on S_G.

Since the call events are often delivered to more than one replica of S_G, group communication lends itself well for such a delivery. The various replicas may be made members of a process group and a message containing the call event sent to the members of the group by a single IPC activity[1]. Thus, the reference to S_G may be given by the pair $(service\_name_G, srvr\_gid_G)$, where $service\_name_G$ uniquely identifies the service provided by S_G and $srvr\_gid_G$ is the process group id of S_G (c.f. section 3.4.2). Using this reference information, C may make one-to-many calls on S_G by sending group messages.

When the one-to-many call is in progress, every replica of S_G is equally capable of completing the call because it executes the call identically and independently with respect to the other replicas. If a replica initiates a many-to-one call on sR and then fails, the orphan caused by the failure is adopted by the surviving replicas without explicit failure recovery by the run-time system. Thus, unlike the primary-secondary scheme of replication described in the previous chapter, the RRPC scheme inherently eliminates the need for checkpointing or restarting. However, since every replica executes the one-to-many call independently, maintaining 'same-end-state' among the replicas is a necessary condition for C to perceive the effect of a single call on S_G.

4.1.1 'Same-end-state' among replicas

By 'same-end-state' among the replicas of S_G, we mean the following: given that all the replicas of S_G have identical states when C initiates a one-to-many call on S_G and that the call on S_G and the various many-to-one calls from S_G on sR are deterministic, the replicas should have identical states at the completion of the call.

However, idempotency violations caused by the many-to-one calls on sR from the replicas of S_G may violate the 'same-end-state' requirement among the replicas. Consider, for example, the code skeleton given in Figure 4.2 implemented by each replica of S_G. Statements (2), (4), (5) and (7) specify 'read'

[1] In terms of the layering described in section 1.4.1, RRPC and group communication constitute the layers L_i and L_{i-1} respectively.

```
function replicated_procedure(argument_list)
  (1) < Some computations >;
  (2) read(resource, buffer1);
  (3) < Computation on buffer1 >;
  (4) write(resource, buffer1);
  (5) read(resource, buffer2);
  (6) buffer2 = < Computations on buffer1 and buffer2 >;
  (7) write(resource, buffer2);
  (8) < Some computations >;
```

Figure 4.2: **Code skeleton of a sample remote procedure**

and 'write' operations on the resource managed by sR. Let us suppose sR manages a terminal. Since the operations are non-idempotent[2], they should be executed only once in response to the many-to-one call requests from the replicas of S_G. Otherwise, a 'read' operation may cause the terminal to hang and a 'write' operation may garble the output on the terminal. Consider another example where sR manages a disk file. If a 'read' operation advances the seek pointer of the file (i.e., the operation is non-idempotent), execution of the operation more than once in response to the many-to-one call requests from the replicas of S_G may lead to non-identical states of the replicas because the pointer value (as well as the data) returned by the different executions of the operation may be different.

We now present a formal treatment of the 'same-end-state' condition in RRPC. Let S_{gi}^{init} and S_{gi}^{compl} be the states of a replica S_{gi} of S_G ($i = 1, 2, \ldots, N$) when a (one-to-many) call is initiated by C on S_G and when the call is returned to C on completion respectively. To ensure 'same-end-state', the replicas should be in the same states when the call is initiated and when the call is completed. These requirements are represented as

$$S_{g1}^{init} = S_{g2}^{init} = \ldots = S_{gN}^{init}, \tag{4.1}$$

$$S_{g1}^{compl} = S_{g2}^{compl} = \ldots = S_{gN}^{compl}. \tag{4.2}$$

[2]It is assumed that the terminal server does not save the characters of a 'read' operation past its completion, so the operation is non-idempotent.

Assume that the requirement (4.1) is satisfied. Let S_{gi}, during its execution, make a sequence of (many-to-one) calls $[TR^k]_{k=1,2,...}$ on sR, as given by the relation (2.1), i.e.,

$$(S_{gi(k-1)}, sR_{k-1}) \stackrel{TR^k}{\rightarrow} (S_{gik}, sR_k),$$

where S_{gik} depends on $(S_{gi(k-1)}, TR^k, p\_val^k)$ and p_val^k is the value returned from sR for TR^k (for TR^1, $S_{gi0} = S_{gi}^{init}$). The final state S_{gi}^{compl} depends on $(S_{gi}^{init}, \{S_{gik}\}_{k=1,2,...})$. Since TR^k and the various dependency relationships are independent of i because the program is deterministic, requirement (4.2) can be satisfied iff $S_{g1k} = S_{g2k} = \ldots = S_{gNk}$ for $k = 1, 2, \ldots$, i.e.,

$$(S_{g1(k-1)}, TR^k, p\_val^k) = (S_{g2(k-1)}, TR^k, p\_val^k) = \ldots = (S_{gN(k-1)}, TR^k, p\_val^k). \quad (4.3)$$

Suppose the replicas of S_G have identical states before TR^k is executed by sR, i.e., $S_{g1(k-1)} = S_{g2(k-1)} = \ldots = S_{gN(k-1)}$. Then, requirement (4.3) is satisfied if the logical effect of the request TR^k from each of the S_{gi}'s is to return the same p_val^k from sR. Since p_val^k depends on sR_{k-1}, the run-time system should ensure the idempotency properties of the calls are not violated.

Note that the above requirement for 'same-end-state' is independent of the actual value of p_val^k returned from sR. Thus it is immaterial whether the TR^k's succeed or fail, the only requirement is that the same value should be observed by each of the S_{gi}'s.

Since the S_{gi}'s and sR change states in relation to the one-to-many and the many-to-one call events, the 'same-end-state' requirement is affected by the interactions among these two types of events. Thus, the 'same-end-state' requirement imposes certain constraints on the call events. The constraints are stringent when the run-time system does not use any application-level information, as in Cooper's model of RRPC [17] described in the next section. In our model of RRPC described in section 4.3, the run-time system uses information about the idempotency properties of the calls to relax the constraints.

4.2 Cooper's model of RRPC

Refer to Figure 4.1. Let C make a one-to-many call on S_G which in turn makes one or more many-to-one calls on sR. Cooper's model ensures 'same-end-state' among the replicas of S_G by enforcing atomicity and order in the sequence of call events arriving at S_G and sR, as stated below:

S_G: Call events observed by a replica of S_G should be observed by all the other replicas, and in the same order. Though this is a necessary condition for 'same-end-state' among the replicas, exactly-once execution of a call [39,53] by a replica requires atomicity and order with respect to the causal sequence at C.

sR: The multiple call events generated by the various replicas of S_G should have the same effect on sR irrespective of the number of replicas of S_G. Such call events from the replicas should be correctly ordered at sR. Thus, the sequence of call events actually observed by sR should satisfy atomicity and order with respect to the causal sequence at S_G.

The above constraints on the call events dictate how the one-to-many and the many-to-one calls may be executed. The constraints are enforced as follows: i) In a one-to-many call, every S_{gi} of S_G must complete its execution before the call is declared to be completed. ii) For many-to-one calls on sR, the latter must wait until the call requests from all the S_{gi}'s have been received; sR must then execute the call once and send the completion event to the S_{gi}'s, thereby completing the call at each of them. We refer to the one-to-many and the many-to-one calls realized in the above manner as *tightly coupled calls.*

The above constraints on the RRPC events require that ordered and atomic message delivery be provided by the underlying group communication layer. For this purpose, Cooper's model uses repeated one-to-one message exchanges to deliver the call events to the replicas, and assumes perfect reliability in the exchanges.

4.3 Our model of RRPC

Our model attempts to meet the 'same-end-state' requirement while relaxing the atomicity and ordering constraints on the call events because these constraints need not be satisfied in many situations and hence are too restrictive for the following reasons:

(i) The tight coupling required in one-to-many and many-to-one calls may result in a large mean value with high variance on the time to complete a RRPC. This is undesirable since the replicated execution may become non-transparent and real-time response of some applications may suffer[3].

(ii) Ordered and atomic delivery of messages in group communication requires extensive management of group membership information [7,19], and often does not allow efficient use of the hardware multicast feature currently available in LANs.

The relaxation of the atomicity and ordering constraints on the call events is based on the commutative characteristics (c.f. section 3.3.1) of the calls.

4.3.1 Commutative characteristics of calls

We make the following observations regarding idempotent sequences:

1. If EV_SEQ_1 and EV_SEQ_2 are idempotent sequences, then $EV\_SEQ_1 \succ EV\_SEQ_2$ is an idempotent sequence.
2. If $EV\_SEQ_1 \succ EV\_SEQ_2 \succ EV\_SEQ_3$ is an idempotent sequence, so is $EV\_SEQ_1 \succ EV\_SEQ_3$.
3. If $EV\_SEQ_1 \succ EV\_SEQ_2$ is an idempotent sequence, so is $EV\_SEQ_2 \succ EV\_SEQ_1$.

The above observations are relevant in RRPC as follows (refer to Figure 4.1): Observation 1 relates to the effect of call re-executions at a server (see section 3.3.1) which may be a replica S_{gj} of S_G or sR.

[3]In ISIS [25] which deals with replicated data, a technique to reduce the user level response time is based on relaxing the synchronization requirements.

Let $EV\_SEQ = [TR^1, TR^2, \ldots, TR^{i-1}, TR^i, TR^{i+1}, \ldots, TR^k]$ be a causal sequence of call events, and suppose EV_SEQ originates from each replica of S_G. The calls from each replica can be interspersed in any order at sR if EV_SEQ is an idempotent sequence. In general, even though a replica S_{gj} may issue a sequence of idempotent calls on sR, if there is at least one non-idempotent call from another replica interspersed in the sequence, then S_{gj} perceives the effect of a non-idempotent call.

Observation 2 relates to missed calls. Suppose EV_SEQ originates at C, and $[TR^1, TR^2, \ldots, TR^{i-1}, TR^{i+1}, \ldots, TR^k]$ is the call sequence seen by S_{gj}. Then S_{gj} has the same end state as the other replicas of S_G if the missed call TR^i is idempotent.

Observation 3 relates to the ordering in a call sequence. Let $[TR^1, TR^2, \ldots, TR^{i-1}, TR^{i+1}, TR^i, \ldots, TR^k]$ be the call sequence seen by S_{gj} while the other replicas see EV_SEQ. Then S_{gj} has the same end state as the other replicas if both TR^i and TR^{i+1} are idempotent. Thus calls in an idempotent sequence may commute in any order at S_{gj} (or sR). A corollary to this observation is that the order of calls in an idempotent sequence seen by S_{gj} need not be the same as that seen by other replicas of S_G.

Thus the RRPC run-time system may relax the constraints on the call events and still ensure the 'same-end-state' condition among the replicas if it uses the (application-level) information about the idempotency properties of calls. The relaxation of the constraints allows i) loose coupling in one-to-many and many-to-one calls wherever possible, and ii) weaker semantics of message delivery provided by the underlying transport layer whereby the sender of a message may continue with incomplete information about the delivery of the message to the recipients. These features of our model are described below:

Loosely coupled calls

Refer to Figure 4.3. C initiates a one-to-many call on S_G by sending a group message to S_G. Let Q be the subset of the group S_G which has received the call message from C. After receipt of call completion messages from at least R_1 members ($\in Q$), for some $R_1 \leq N$, C considers the call to be completed and

continues with its computation[4].

A many-to-one call may be executed by sR when the first call request TR_{gl} from one of the replicas S_{gl} in Q arrives at sR. The completion event TC_{gl} for the call is returned by sR in a message to all the replicas of S_G.

Semantics of message delivery

When a sender (which may be C, sR or a member of S_G) invokes group communication on S_G, the sender may not know (and often does not need to know) exactly what will happen at the various members of S_G. This is because the sender usually has no knowledge of the identity of the group members. The problem is compounded by independent failures of the members and partitioning of the group by communication break-downs. However, often the important information the sender needs to know is not exactly how many group members have carried out the operation but a reasonable lower bound. Thus, the outcome of group communication may be represented by **ATLEAST(r)** indicating that at least a certain number of members, specified by **r**, are known to have carried out the requested operation [10]. The **r** used in the above representation is called the *degree of message delivery*, and may be given as either a fraction or an integer (for more details on the semantics, see section 5.2).

When group communication is used in RRPC's, the **r** may be construed as specifying the required level of coupling in the calls. The sender may specify **r** using its knowledge of the application and the environment.

In the next section, we analyze how the 'same-end-state' condition among the members of S_G may be affected by the loose coupling in calls and the weak semantics of group communication. The analysis exposes the possible undesirable executions in RRPCs and helps to formulate the algorithms and the protocols to ensure 'same-end-state'.

[4] R_1 is usually set to 1

4.4 Undesired executions in RRPC

A snap-shot of the execution states of the S_{gj}'s illustrates the interactions between the various call events and the resulting undesirable executions. Refer to Figure 4.3. The undesirable executions should be detected and handled by the RRPC run-time system.

4.4.1 Null and starved executions

Let Q' be the subset of S_G which has received the completion event TC_{gl} returned by sR. Consider the following cases with respect to the member S_{gj} of S_G:

Case 1. $S_{gj} \in Q \cap Q'$.

For $j \neq l$, when S_{gj} initiates the (many-to-one) call request TR_{gj} on sR, the completion event TC_{gl} is replayed, i.e., the waiting message is paired with the request message, and TR_{gj} completes immediately. For $j = l$, TR_{gl} is completed when sR returns TC_{gl}. Thus,

$$TR_{gl} \succ TC_{gl} \succ T\_ARR_{gj,l} \succ TR_{gj},$$

where $T\_ARR_{gj,l}$ is the arrival of TC_{gl} at S_{gj}. In this case ($j \neq l$), the effect of the causal event TR_{gj} is already available with S_{gj} in the form of $T\_ARR_{gj,l}$.

Case 2. $S_{gj} \in (S_G - Q) \cap Q'$.

S_{gj} has not received the (one-to-many) call request from C and hence no (many-to-one) call request to sR will originate from S_{gj}, but completion events from sR for the many-to-one calls are buffered at S_{gj}. The buffered completion events are meant for later replay which in this case will never occur. Suppose $T\_ARR_{gj,l}$ (see case 1) is such an event as given by

$$TR_{gl} \succ TC_{gl} \succ T\_ARR_{gj,l}.$$

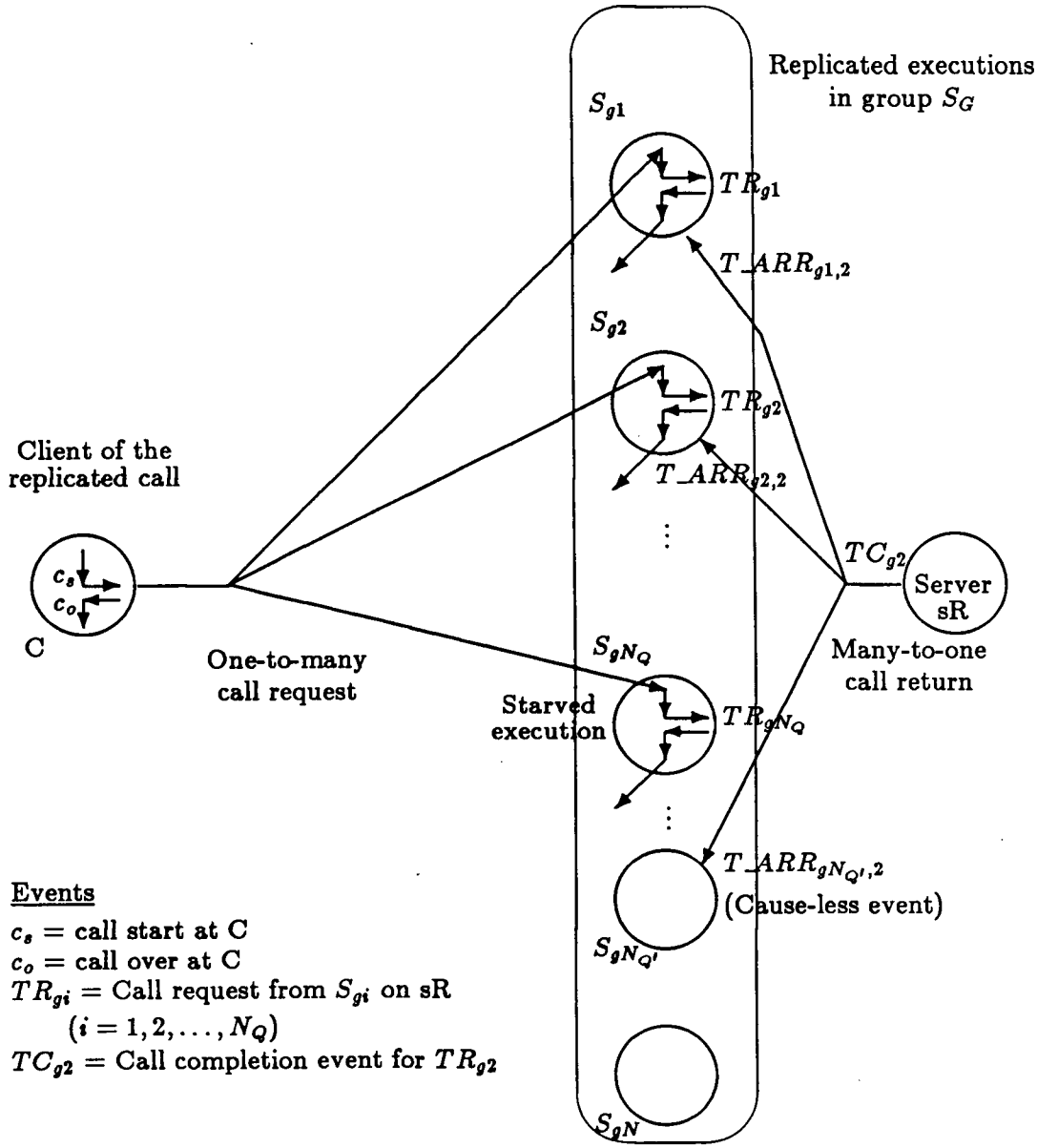


Figure 4.3: **Diagram to illustrate the undesired executions**

We refer to $T\_ARR_{gj,l}$ as a *cause-less event* because it has occurred at S_{gj} (due to the causal event TR_{gl} from S_{gl}) but the local causal event TR_{gj} will not occur. The states of such S_{gj}'s may be different from that of the other members.

Case 3. $S_{gj} \in (S_G - Q) \cap (S_G - Q')$.

S_{gj} has not received the call event from C or the completion event from sR. Its state may be different from that of the other members.

Case 4. $S_{gj} \in Q \cap (S_G - Q')$.

S_{gj} executes the one-to-many call, but because it has not received the completion event TC_{gl} for the many-to-one call, it will initiate the call request on sR causing more than one execution of the call by sR. We refer to the execution by S_{gj} as a *starved execution* because it did not receive the completion event TC_{gl}. This may be due to a communication break-down when sR sends TC_{gl} to S_G.

4.4.2 Orphaned executions

The execution by a member S_{gj} is an orphan if the call request from C that caused the execution is no longer out-standing. This may occur either because C has assumed the call is completed or failures have occurred. We categorise these orphans based on their causes:

Failure orphans

Suppose S_{gj} initiates a (many-to-one) call on sR and then fails. The failure of S_{gj} makes sR a failure orphan. Such orphans should be adopted by one of the surviving members.

Lazy orphans

The execution by S_{gj} is a lazy orphan if C has already completed its (one-to-many) call on S_G. The orphan may occur due to loose coupling in the call — some members of S_G may still be executing the call when C considers the call to be completed on receiving call returns from the other members. Such orphans may interfere with subsequent calls from C and violate the 'same-end-state' requirement.

Partitioned orphans

The execution by S_{gj} is a partitioned orphan if S_{gj} has been partitioned from the on-going call activities due to a communication break-down. We consider the following cases of partitioning:

- S_{gj} partitioned from C.

Since C may have noted the failure due to lack of sufficient number of responses from S_G, the continued execution of S_{gj} may interfere with other calls.

- S_{gj} partitioned from sR.

Suppose S_{gj} requests a call on sR during the time it is partitioned from sR. The call will be noted as failed by S_{gj} while that from some other member of S_G will be noted as succeeded. Thus S_{gj} will be in a different state from the other members. Note that this type of partitioning may also result in a starved execution at S_{gj} described earlier in section 4.4.1.

- S_{gj} partitioned from other members in S_G.

S_{gj} remains connected to C and sR. Since communication among the members do not exist at the application level, the execution by S_{gj} is largely independent of that by other members, so the partitioning of S_{gj} from the rest of S_G may not affect the progress of the call.

4.5 Solution approach

The run-time system should ensure the 'same-end-state' condition among the members of S_G against the above undesired executions which may surface amidst normal executions at random points in time. It should also prevent members from becoming un-coordinated among one another which may lead to members dropping out of the group thereby reducing program availability. Our solution approach encapsulates the following techniques to meet these goals:

- Controlled re-execution of the calls if necessary (c.f. section 3.3).
- Replaying call completion events from logs at both the client and the server — the logs allow un-coordinated executions to get in step with other executions without actually re-executing the calls.
- Lateral coordination among the replicated executions.
- Detection and handling of orphans.

The solution techniques make use of the commutative characteristics of calls analyzed in section 4.3.1, and are described in the remaining sections of the chapter.

4.6 Protocols for handling one-to-many calls

Consider a one-to-many call from C to S_G. If the call is connection-less, C[5] waits until R1 members of S_G have returned the call and then continues with its computation. Suppose the call is connection-oriented. C maintains the T_id of the last call, referred to as $(G\_tid_{rqst,C,S_G}, nI\_tid_{rqst,C,S_G})$, for its connection to S_G. In addition, C registers a death-will in favor of S_G (refer to appendix A.1). S_{gj} ascertains the inclusion of S_G in the death-will before accepting any calls. The death-will allows orphans due to failures to be detected.

[5] Actually, we refer to the run-time system at C's site.

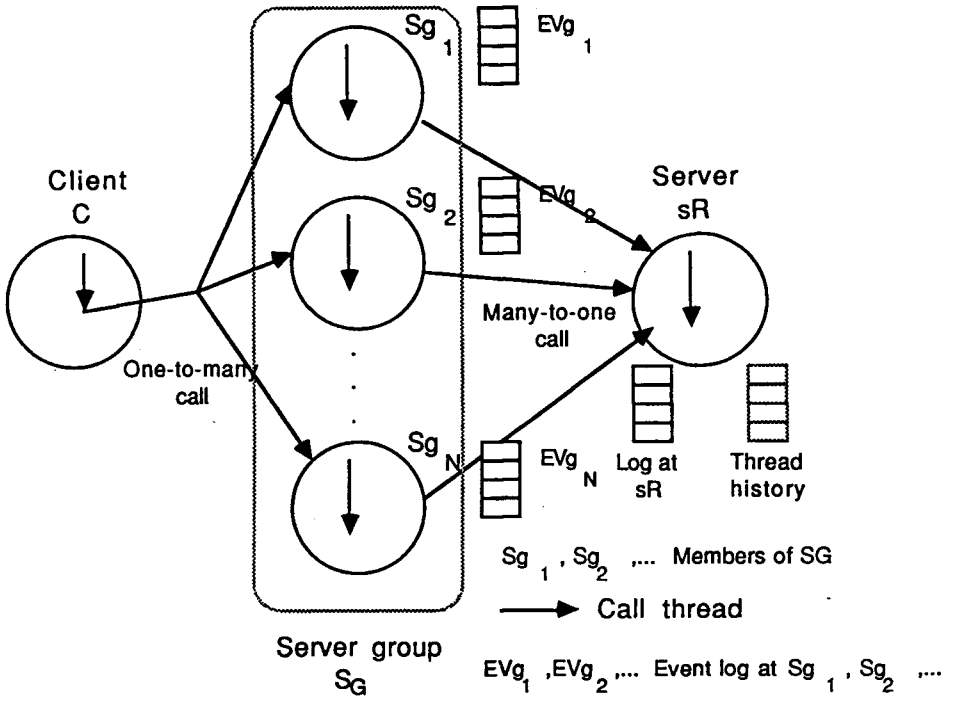


Figure 4.4: Structure of the RRPC run-time system

Refer to Figure 4.4 which illustrates the structure of the run-time system.

4.6.1 Call initiation by C

C assigns the id pair $(G\_tid_{cur,C,S_G}, nI\_tid_{cur,C,S_G})$ to the call request, where $G\_tid_{cur,C,S_G} = (G\_tid_{rqst,C,S_G} + 1)$ and $nI\_tid_{cur,C,S_G} = (nI\_tid_{rqst,C,S_G} + 1)$ if the call is non-idempotent or 1-idempotent, and equal to nI_tid_{rqst,C,S_G} for idempotent calls. C then moves from the EXECUTING to the SUSPENDED state.

4.6.2 Call validation at S_{gj}

T_id_{cur,C,S_G} is compared to $T\_id_{last,j}$, the last call completed by S_{gj}. The following situations are possible:

Case 1. $G\_tid_{cur,C,S_G} = (G\_tid_{last,j} + 1)$, i.e., no calls have been missed.

The requested call is a new one, sR carries out the call and sends a completion message to S_G.

Case 2. $G\_tid_{cur,C,S_G} > (G\_tid_{last,j} + 1)$, i.e., one or more calls have been missed.

If $nI\_tid_{cur,C,S_G} = nI\_tid_{last,j}$, or if the call is non-idempotent or 1-idempotent and $nI\_tid_{cur,C,S_G} = (nI\_tid_{last,j} + 1)$, i.e., no non-idempotent or 1-idempotent calls have been missed by S_{gj}, then S_{gj} carries out the requested operation; otherwise, it attempts a recovery as described in section 4.7.5.

Case 3. $G\_tid_{cur,C,S_G} < (G\_tid_{last,j} + 1)$, i.e., the call is being re-issued.

If the call is idempotent or 1-idempotent, and $nI\_tid_{cur,C,S_G} = nI\_tid_{last,j}$, then S_{gj} re-executes the call and sends the return message to C. If the call is non-idempotent or $nI\_tid_{cur,C,S_G} < nI\_tid_{last,j}$, then S_{gj} rejects the call.

If the call is valid, S_{gj} moves from the IDLE state to the EXECUTING state, and initiates the local thread of the call.

4.6.3 Call progress at S_{gj} and call return

A necessary condition for S_{gj} to continue the call (and hence to initiate a call on sR) is that S_{gj} is not an orphan. For this purpose, S_{gj} maintains a variable **call_active**. A true value indicates the call T_id_{cur,C,S_G} is active at S_{gj}; a false value indicates otherwise. S_{gj} is not an orphan only if C's death-will has not been executed and **call_active** is true.

If S_{gj} is not orphaned, it sends the call return to C and moves from the EXECUTING to the IDLE state. The call return includes the thread state of S_{gj} (i.e., the set of T_id's for the connections from S_G to the sR's). If S_{gj} is orphaned, it takes part in a completion protocol described in the later part of the next section (4.6.4).

If the call has not been completed, C may buffer the return value from S_{gj}. In a simple scheme, C buffers only the first value returned.

4.6.4 Call completion by C

C may wait until the desired level of coupling of the call is achieved, i.e., until at least R_1 members in S_G have returned the call[6]. Its next action depends on the call type:

- **The call is idempotent.**

C sends a CALL_OVER(T_id_{cur,C,S_G}) message to S_G indicating that the call has been completed by C; the message also includes the thread state contained in the call return. On receiving this message, each of the S_{gi}'s resets **call_active** to false, updates its thread state and aborts its on-going call if any. Non-receipt of the message by S_{gj} is not harmful since S_{gj} will detect its on-going call had been completed by C when the next call request arrives. In any case, the continuation of S_{gj} is harmless until the arrival of a non-idempotent or 1-idempotent call request from C.

Thus for idempotent calls, the scheme requires one group message to initiate a call, one group message (CALL_OVER) to terminate the call and one or more reply messages from the members of S_G.

- **The call is non-idempotent or 1-idempotent.**

C selects a S_{gj} that has returned the call[7] as a coordinator to commit the effects of the call with the other S_{gi}'s which act as cohorts [5]. It sends the message UPDATE to S_{gj}. On receiving the message, S_{gj} sends its permanent variable PV [1,24] and its thread state to the other members of S_G in an intra-group message with the degree of delivery set to 1.0 (i.e., confirmed delivery to every member of the group — refer to the semantics of message delivery discussed in section 4.3.1). The message advises any lazy orphans to update their PV's and the thread state to that contained in the message, and move to the same final state. On receiving the message, each of the S_{gi}'s aborts its on-going execution if any, updates its PV and thread state, resets **call_active** to false and replies to S_{gj}. After all the S_{gi}'s have

[6] A timeout may also be specified.

[7] A simple way is to select the first member that sent the return.

thus moved forward to the same (final) state, S_{gj} replies to C confirming completion of the commit operation, whereupon C considers the call to be completed.

Thus for non-idempotent calls, the scheme requires one group message to initiate a call, 2 messages between C and the selected coordinator, one group message from the coordinator to the server group, and N replies from the members.

4.7 Protocols for handling many-to-one call

Consider a many-to-one call (as part of the call thread initiated by C) from S_G to sR. Suppose the call is initiated by member S_{gj}. If the call is connection-less, sR simply executes the call and sends the return to S_{gj}.

Suppose the call is connection-oriented. Refer to Figure 4.4. The call from S_{gj} is identified by the pair $(G\_tid_{cur,j,sR}, nI\_tid_{cur,j,sR})$, referred to as $T\_id_{cur,j,sR}$, where $G\_tid_{cur,j,sR} = G\_tid_{rqst,j,sR} + 1$, and $nI\_tid_{cur,j,sR} = nI\_tid_{rqst,j,sR} + 1$ if the call is non-idempotent or 1-idempotent, and equal to $nI\_tid_{rqst,j,sR}$ if the call is idempotent. sR maintains a call id pair $(G\_tid_{last,sR}, nI\_tid_{last,sR})$ for the last call it has completed in response to a request from some member of S_G. sR compares $(G\_tid_{last,sR}, nI\_tid_{last,sR})$ with $(G\_tid_{cur,j,sR}, nI\_tid_{cur,j,sR})$ to decide on its course of action. Before we present the protocols, the event log structure used by the protocols will first be described.

4.7.1 Event logging in the replicated caller

The event log is distributed across the members of S_G. Besides replaying call completion events locally, the log also allows the various members of S_G to interact among themselves to exchange state information.

When S_{gj} requests a call on sR, the latter may execute the call if it is the first request, and send the completion event TC_{gj} (with a suitable degree of delivery) to the entire group S_G. On receiving the event, each of the S_{gi}'s may log it locally for later replay both to itself and to the other group members.

Let $K_j(\geq 1)$ be the maximum number of events that can be logged by S_{gj}. Assuming that the logged events are replaced using a FIFO algorithm, a call completion event remains in the log for the next K_j operations[8].

4.7.2 Conditions for call initiation

The fact that S_{gj} is not an orphan is not a sufficient reason for S_{gj} to initiate a call on sR as this only implies that the call from C is still active, but some other S_{gi}'s may have already initiated the call on sR. Thus, in principle, S_{gj} should initiate a call on sR only if it is not an orphan and if it is the first in S_G to execute the call. However, this may be too restrictive a condition because a re-execution of the call by sR may not always be harmful, or sR may be able to handle multiple requests for the same call. Thus, with proper support from sR, a less restricted condition is used which allows S_{gj} to initiate a call whenever it is not an orphan.

4.7.3 Call initiation by S_{gj}

On ascertaining it is not an orphan (as described in section 4.6.3), S_{gj} first checks for the existence of the corresponding completion event locally. If the event exists, implying that the call has already been carried out by sR in response to a request from some other member of S_G, the event is paired with the call request from S_{gj} and no message needs to be sent to sR.

The absence of the event, though a necessary condition, is not sufficient for S_{gj} to ascertain it is the first executor of the call since a starved execution may also encounter the same condition. In general, S_{gj} may not always be able to detect the first execution condition using local information alone. Thus, S_{gj} needs to interact with sR in the form of a forward probe. If it detects that it is a starved execution, then it should interact with other members in S_G (lateral probe) to acquire the completion event. Since execution of the call requires interaction with sR anyway, starvation detection and call initiation are

[8]CIRCUS assumes an unlimited capacity in the run-time system to log completion events when loose coupling is used in many-to-one calls.

combined. This results in the following order of probing — forward and lateral — which are presented in the next two sections.

4.7.4 Forward probe

S_{gj} sends the call $(G\_tid_{cur,j,sR}, nI\_tid_{cur,j,sR})$ to sR. In general, the following situations are possible at sR:

Case 1. $G\_tid_{cur,j,sR} = (G\_tid_{last,sR} + 1)$.

The requested call is a new one, so sR carries out the call and sends the completion message to S_G.

Case 2. $G\_tid_{cur,j,sR} < (G\_tid_{last,sR} + 1)$, i.e., S_{gj} is a starved execution.

If the requested call is idempotent or 1-idempotent, and $nI\_tid_{cur,j,sR} = nI\_tid_{last,sR}$, then sR re-executes the requested operation and returns the results to S_{gj}. If the call is non-idempotent or if $nI\_tid_{cur,j,sR} < nI\_tid_{last,sR}$, then sR returns an error message ALRDY_OVER to S_{gj}.

Thread history at the server

The re-execution of a call by sR requires that all resulting calls originating from sR should be identifiable as re-issued calls. For this purpose, sR maintains a history of the thread states in a buffer. When sR completes a call, it stores its thread state in the buffer. When sR decides to re-execute a call, it saves the current thread state and rolls the state back to when the execution of the call first occurred. After completion of the re-execution, it restores its current thread state. Thus, if the size of the buffer is K_h, a call may be re-executed only if its $G\_tid > (G\_tid_{last,sR} - K_h)$ even though its re-execution may otherwise be harmless.

Event logging at the server

The server sR may also optionally maintain a log of call completion events for possible replay. With such a log, a call request from a starved execution may be satisfied by sR by replaying the appropriate completion event. If sR cannot satisfy the request from its log, it then resorts to the protocols described above in section 4.7.4.

Handling of server responses

On receiving the call return, S_{gj} continues with its computation. If, on the other hand, an error message ALRDY_OVER is received, S_{gj} detects that it is a starved execution; or if sR is not reachable (due to a communication break-down), the underlying message layer returns an error message PROCESS_UNREACHABLE [46]. In either case, S_{gj} may employ the lateral probe described in the next section to get in step with the other members.

4.7.5 Lateral probe

S_{gj} resorts to this phase if it is a starved execution or its attempt to initiate the call on sR failed with the error message PROCESS_UNREACHABLE. To acquire the completion event for the call $T\_id_{cur,j,sR}$, S_{gj} sends an intra-group message PROBE($T\_id_{cur,j,sR}$, *probe_attr*) to acquire a replay of the completion event for the call; *probe_attr* is used to specify the event that triggers the probe. Each of the S_{gi}'s ($i \neq j$) checks its log of call completion events. If the completion event for $T\_id_{cur,j,sR}$ is available, then S_{gi} replies to S_{gj} with the event message. If the event is not found and ($T\_id_{last,i,sR} \geq T\_id_{cur,j,sR}$), or if ($T\_id_{last,i,sR} < T\_id_{cur,j,sR}$) and the reason for the probe is sR cannot be reached by S_{gj} (as specified in *probe_attr*), then S_{gi} replies with a message containing $T\_id_{last,i,sR}$. In all other cases, S_{gi} discards the PROBE message without replying.

In the above lateral probe protocol, the cause-less events[9] described in section 4.4 turn out to be useful. As seen, the cause-less events at the member S_{gi} are also used for replay in response to PROBE requests from other members (such as S_{gj}) even though the events do not get replayed locally (to S_{gi}).

Handling replies from other members

If one or more responses containing the call completion event is received, S_{gj} replays the event for the call $T\_id_{cur,j,sR}$. If none of the received responses contain the completion event and the smallest $T\_id_{last,i,sR}$ (contained in the responses) $\geq T\_id_{cur,j,sR}$, or there are no responses to its PROBE request, S_{gj} assumes that the event is not available in the log of any other member. This constitutes a potential condition for S_{gj} to drop out of S_G since it is not able to complete $T\_id_{cur,j,sR}$ and get in step with the other members (more on this in section 4.7.6 below). On the other hand, if $T\_id_{last,i,sR}$ in the replies is less than $T\_id_{cur,j,sR}$, then the call has not yet been initiated by the other members of the group; S_{gj} may probe again after a time out interval.

The lateral probe algorithm requires one group message (PROBE) followed by one or more responses to the message.

4.7.6 Lateral coordination

If a member S_{gj} of S_G is unable to keep in step with other members after its lateral probe attempts have failed (as described above in section 4.7.5), it will have to drop out of the group. This will in turn reduce the availability of the server. The following algorithms are designed to minimize this possibility. The idea behind the algorithms is to suspend the fastest member if a slower member finds itself in danger of dropping out of the group. When the slower member catches up with the suspended member, the latter is allowed to resume execution.

[9]The buffer space used by the cause-less events may be reclaimed by any standard garbage collection mechanism.

Brake algorithm

A slower member may exercise a control algorithm that allows it to coordinate with other members and prevent the conditions for drop out from arising. Suppose the completion event for the on-going call $T\_id_{cur,i,sR}$ (from some S_{gi}) arrives at S_{gj}. If $(T\_id_{cur,i,sR} - T\_id_{rqst,j,sR}) \geq (K_j - M)$ for some $M \geq 0$, S_{gj} initiates control by sending a BRAKE_R message to sR advising the latter to hold execution of the next non-idempotent or 1-idempotent call[10]. sR then sets a boolean variable **brake_flag** to true, indicating that a brake condition has been set. Under this condition, sR may execute a call only if the call is fully idempotent, otherwise it suspends the call until the brake condition is reset.

- **Brake release**

The brake may be abstracted as a resource shared among the members of S_G, so it should be protected against failures of the member (S_{gj}) currently holding it. Otherwise, the failure of S_{gj} may cause the brake to be in effect forever preventing the other members from executing. The failure recovery is integrated into the protocol that sets up the brake condition as follows: When S_{gj} sends the BRAKE_R message to sR, it registers a BRAKE_REMOVE message as its death-will on sR (see appendix A.1). When S_{gj} completes a call such that $(T\_id_{cur,i,sR} - T\_id_{rqst,j,sR}) < K_j$, S_{gj} releases the brake by sending the BRAKE_REMOVE message to sR and cancelling the death-will. If S_{gj} fails for whatever reason, the death-will (BRAKE_REMOVE) message is delivered to sR. In any event, sR resets its **brake_flag** (thus releasing the brake), and resumes the execution of any suspended call.

The lateral co-ordination algorithm requires 4 messages to set up the brake condition (BRAKE_R) and subsequently release it (BRAKE_REMOVE).

[10] The size of the event logs in all member sites is assumed to be the same; M is a constant of the algorithm.

4.8 Analysis of the lateral coordination

Since the level of availability of S_G is largely determined by the number of members in S_G, the algorithms strive to maintain the members in a sufficiently synchronized state and prevent them from dropping out. The effectiveness of the algorithms is thus given by how far a member S_{gj} can be out of synchronization with the other members and still not be required to drop out. We provide the following quantitative analysis to study this.

Suppose C makes a one-to-many call on S_G which in turn makes many-to-one calls on sR. Let TR^k be the last call carried out at sR by the fastest member in S_G and TR^r be the most recent call completed by the slowest member. The latter may drop out of the group if it is unable to carry out the next call (refer to section 4.7.5). Assume the slowest member is suspended from execution while the fastest member is allowed to continue. Let k = (r+i) and P_{idem} be the probability that a call requested is idempotent (independent of the other calls). If pR_i is the probability that the completion of the $(r+i)^{th}$ call by the fastest member forces the slowest member out of the group, then

$$pR_i = \begin{cases} 0 & \text{for } 0 \leq i \leq K_j \\ 1-(P_{idem})^i & \text{for } i = K_j+1 \\ (P_{idem})^{i-1}.(1-P_{idem}) & \text{for } i \geq K_j+2 \end{cases}$$

The mean number of calls $\overline{N}_{leav}$ that a member may fall behind the fastest member before being forced to leave the group is given by

$$\overline{N}_{leav} = (K_j+1).(1-(P_{idem})^{K_j+1}) + \sum_{i=K_j+2}^{\infty} i.(P_{idem})^{i-1}.(1-P_{idem}) \qquad (4.4)$$

Since $\overline{N}_{leav}$ represents the maximum 'distance' in terms of the number of (many-to-one) calls between the fastest and the slowest members, it may be construed as representing the maximum possible unsynchronization among the various members before some member will be required to leave the group. We introduce an index, which we call the *dispersion factor (DF)*, to characterize this behavior of the program under the given system[11]. DF is expressed in terms of $\overline{N}_{leav}$ in a normalized form ($0.0 \leq DF \leq 1.0$) as

[11]The index is generalizable to deal with more than one server (i.e., multiple sR's).

given below:

$$DF = (1 - \frac{1}{\overline{N}_{leav}}). \tag{4.5}$$

For $P_{idem} = 0.0$, $DF = 1 - \frac{1}{(K_j+1)}$ and the program has the least dispersion. For $P_{idem} \rightarrow 1.0$, $DF \rightarrow 1.0$ and the program has the maximum dispersion. In its normalized form, the DF reflects the probability that a member continues to remain in the group, and is useful in reliability analysis of replicated servers.

4.8.1 Effect of thread history

The finite size K_h ($\gg K_j$) of the buffer containing sR's thread history limits the number of calls, irrespective of their types, by which the members may remain un-coordinated. Thus, if $G\_tid_{rqst,j,sR} \leq (G\_tid_{cur,i,sR} - K_h)$, then the member S_{gi} may be forced to suspend the next call (see the brake algorithm in section 4.7.6) even though its execution may not cause any idempotency violation. Thus equation 4.4 may be modified as follows:

$$pR_i = \begin{cases} 0 & \text{for } 1 \leq i \leq K_j \\ 1 - (P_{idem})^i & \text{for } i = K_j + 1 \\ (P_{idem})^{i-1}.(1 - P_{idem}) & \text{for } K_j + 2 \leq i \leq K_h \\ (P_{idem})^{i-1} & \text{for } i = K_h + 1 \end{cases} \tag{4.6}$$

Thus $\overline{N}'_{leav}$ (limited by K_h) is given by

$$\overline{N}'_{leav} = (K_j + 1).(1 - (P_{idem})^{K_j+1}) + [\sum_{i=K_j+2}^{K_h} .i.(P_{idem})^{i-1}.(1 - P_{idem})] + (K_h + 1).(P_{idem})^{K_h} \tag{4.7}$$

Refer to the graph in Figure 4.5. Since $\overline{N}'_{leav}$ is a non-decreasing function of P_{idem}, the higher the mix of idempotent calls in a program, the better it is able to sustain variations in the operating environment (i.e., higher dispersion of the program). This is because the run-time system uses application-level information to re-execute calls if necessary and wherever possible. This allows a weaker (and hence less expensive) form of communication to be used, and also tends to reduce the probability of members dropping out of the group (which would decrease the failure tolerance of the program).

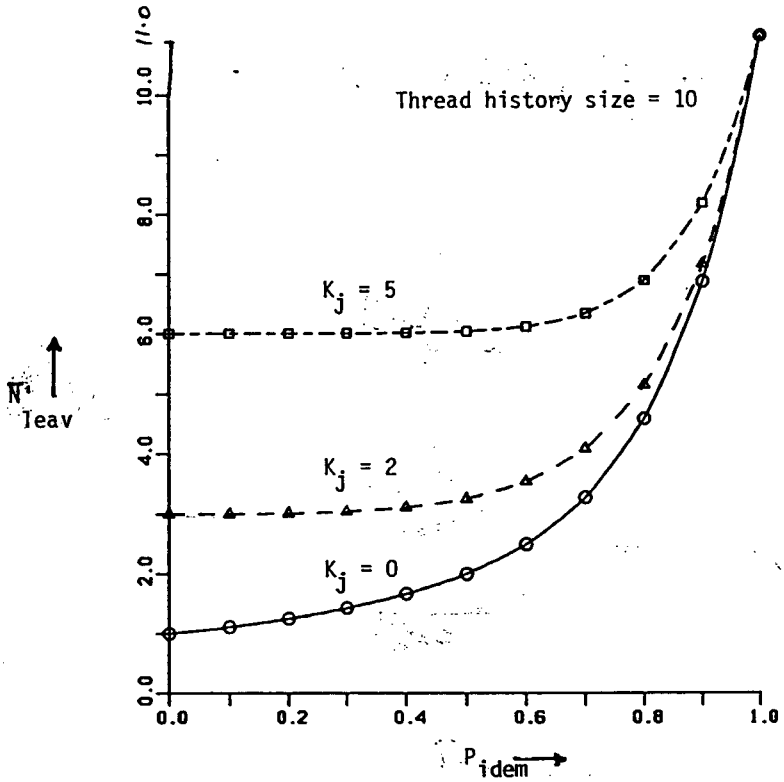


Figure 4.5: **Variation of the dispersion factor with respect to P_{idem}**

The analysis provides an indication of the size of the event logs required in the RRPC run-time system. For example, if in a data base application, 'read' calls constitute 90% of the total number of calls while 'write' calls constitute 10%, then since 'read' calls are usually idempotent, $P_{idem} = 0.9$. Given an acceptable level of dispersion specified in terms of $\overline{N}'_{leav}$, a suitable value of K_j may be obtained from the graph in Figure 4.5.

4.9 Summary

In this chapter, we have described a scheme to mask the failure of a server in which a call from a client on the server is executed by more than one replica of the server at the same time. When any of the executing replicas fails, the call may be completed by the surviving replicas without explicit failure recovery initiated by the run-time system. The failure is transparent to the client because the replicated execution of the server is itself transparent. The scheme thus avoids the checkpointing and restarting

required in the primary-secondary form of replication described in chapter 3.

The scheme makes use of the idempotency properties of calls to relax the atomicity and the ordering constraints on the calls but still maintains the 'same-end-state' condition among the replicas. The relaxation of the constraints allows a weak form of group communication to be used. Connection-less calls on a server are simply re-executed by the server irrespective of which replica requests the call. For connection-oriented calls, the scheme uses call re-executions, call replay from a server, and lateral coordination among the replicas.

The algorithms in the scheme do not use rollback, thus the outcome $CALL\_FAIL$ is not part of the failure semantics. Referring to Figure 4.1, if all members of S_G fail, then the outcome delivered to C is $CALL\_INCONSISTENT$. If the $CALL\_FAIL$ outcome is desirable, rollback may additionally be incorporated into the algorithms.

The prototype implementation of the primary-secondary scheme of replication in RPC (c.f. chapter 3) has been extended to replicate the execution of a server and incorporate the various algorithms to ensure 'same-end-state' among the replicas of the server [43]. The V-kernel's group communication, supplemented by our message delivery semantics, is used as the underlying message transport layer for RRPC. So far, with the few cases of error conditions introduced in the experiments, the scheme has worked as expected.

Chapter 5

Relaxation of consistency constraints on shared variables

Unlike the case of the RPC abstraction which deal with client-server interactions, the ADSV abstraction deals with intra-server group communications to operate on a distributed shared variable (refer to the program model of section 2.4). The ADSV abstraction should provide a logically consistent view of the shared variable from its potentially inconsistent instances.

In this chapter, we first make a case for relaxing the consistency constraints on a ADSV, i.e., the acceptable level of consistency (or inconsistency) among the instances of the variable. The relaxation of the constraints allows usage of a weak form of group communication in the underlying algorithms and protocols to realize the ADSV operations. We describe three examples to show how the algorithms and the protocols may be used in managing shared resources — the examples are management of the leadership in a server group, the printer in a spooler group, and the name space of machines in a distributed system.

5.1 Consistency requirements on ADSV

Let V be a distributed state variable shared among the members of a server group which may span across more than one distributed program. Issues such as concurrency control, atomic actions and

mutual exclusion affect the consistency of the variable V in some way. These issues have also been studied in the context of maintaining consistency of (secondary storage) data in distributed data bases. However, the solution techniques developed for one may not always be suitable or necessary for the other because the requirements are fundamentally different [13,40] (c.f. section 1.3.2). Typically, distributed server interfaces, which contain only operating system variables do not require absolute consistency with the attendant penalty of higher overhead. Occasional inconsistencies may be detected at a 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 databases such as banking systems. To be specific, incorrect updates to a banking file have far more serious consequences and are far less easy to detect than those to the name registry used to locate the file. Because of such requirements, algorithms used in databases usually enforce strong consistency and are quite complex [28,13].

In our approach, the consistency constraints on V depend on the resource V abstracts. The approach does not require absolute consistency but provides access operations with simple properties upon which applications can build additional properties if necessary. The relaxation of the constraints in turn reduces the complexity of the underlying protocols that maintain the consistency of V. In other words, the approach allows the use of a minimal set of protocols for the operations as required by the applications. Generally, this will result in better efficiency in the execution of the operations. Inconsistencies, if any, may be detected and handled (by the protocols realizing the operations) when the resource is accessed. The extent of relaxing the constraints is largely based on the frequency with which an inconsistency may occur, the implications of the inconsistency and how the higher level algorithms handle the inconsistency. Consider, for example, multiple update requests on a (distributed) name registry. Though absolute consistency requires all members of the name server group to observe the same order in which the locks on the registry are acquired and released, the above constraint may be relaxed pro-

vided the resulting inconsistency may be handled during the name resolution phase. To illustrate the example further, consider a service name bound to a process group id when the service is registered as a $(service\_name, srvr\_gid)$ pair with the name server group (c.f. section 2.2). If communication break-downs occur among the members of the name server group during the registration, the various instances of the binding information at the group members may be inconsistent, resulting in more than one group claiming to offer the service, e.g., $(service\_name, srvr\_gid_1)$ and $(service\_name, srvr\_gid_2)$. The inconsistency may be detected by a client on receipt of responses from both $srvr\_gid_1$ and $srvr\_gid_2$ to its service request. When the inconsistency is detected, the client may force the name server group to register the service under a single group id (see [47] for details). As another example, suppose an object migrates or changes its name causing the binding information for the object to change [29]. The migration or the name change activity is atomic only if all the externally-held references to the object are corrected as part of the activity. Such a correction requires atomic delivery of notification of the change in the binding information. Instead, if a client of the object can correct its reference to the object at the time of access using a search protocol [46], a weak delivery of the notification message or even no notification at all may be sufficient at the time of the change [48,55]. There are also many situations such as occasional failures to correctly serialize output to a shared printer that can be handled by the users. Thus, an *operational consistency* of the ADSV may be sufficient in many applications whereby the variable V need not be totally consistent so long as the correct operation of the group is not compromised.

To further illustrate our notion of operational consistency, let us compare it with *eventual consistency* whereby if we stop initiating new activities on the variable V, the algorithms to operate on V will eventually lead to consistent states of V because the algorithms are usually executed as atomic actions. For example, the group communication primitives proposed by Birman [7] and Cristian [19] provide atomicity, order and causality properties in the group communication layer; such primitives are used to

perform atomic and ordered operations on V and to ensure its eventual consistency. In a system that provides only operational consistency, inconsistencies among the instances of V may exist because the algorithms to operate on V may not be executed as atomic actions. However, when the applications perform operations which use the inconsistent information, the protocols that realize the operations deal with the inconsistencies. Atomic actions may be employed in our algorithms too, but not as a rule. Our 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 [13].

As pointed out earlier, the various operations on the ADSV may be realized using a weak form of group communication, the semantics of which is given in the next section (see also section 4.3.1).

5.2 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[1].

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.

Taking these into considerations, we introduce a parameter **R** in the group communication primitive [10]. **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,

[1]For deterministic programs, this may not be an issue.

value) pairs described earlier in section 2.5 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

$$[R, (att\_nm_1, att\_val_1), \ldots, (att\_nm_K, att\_val_K)]$$

in the group communication primitive. The sender may specify the values $\{att\_val_i\}_{i=1,2,\ldots,K}$ for the attributes $\{att\_nm_i\}$ (c.f. section 2.5). 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 RRPC's, 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 [16]. 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 [10] 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 how the ADSV operations proposed in section 2.7.2 may be realized using the above form of group communication. As we shall see, the underlying protocols exemplify the contention style of communication among the members of the group.

5.3 'Lock' and 'Unlock' operations

Mutual exclusion is a basic requirement for controlled access to a shared resource (e.g., acquiring a lock on a shared file for update operations). It consists of three steps, namely, i) acquiring a lock on the resource, ii) accessing the resource, and iii) releasing the lock [38,5]. Since access to the resource by members of the server group is distributed in our program model, inconsistencies in the state of the lock may arise as follows:

Issue 1 Partial failures associated with a member that has locked the resource (or is in the process of locking the resource) may result in other members waiting for the resource that will never be released.

Issue 2 Simultaneous access requests (i.e., collisions) by two or more members may result in erroneous updates to the resource and unpredicatable errors.

The mechanisms to handle partial failures and collisions are best built into the lock acquisition protocol as described in the following sections.

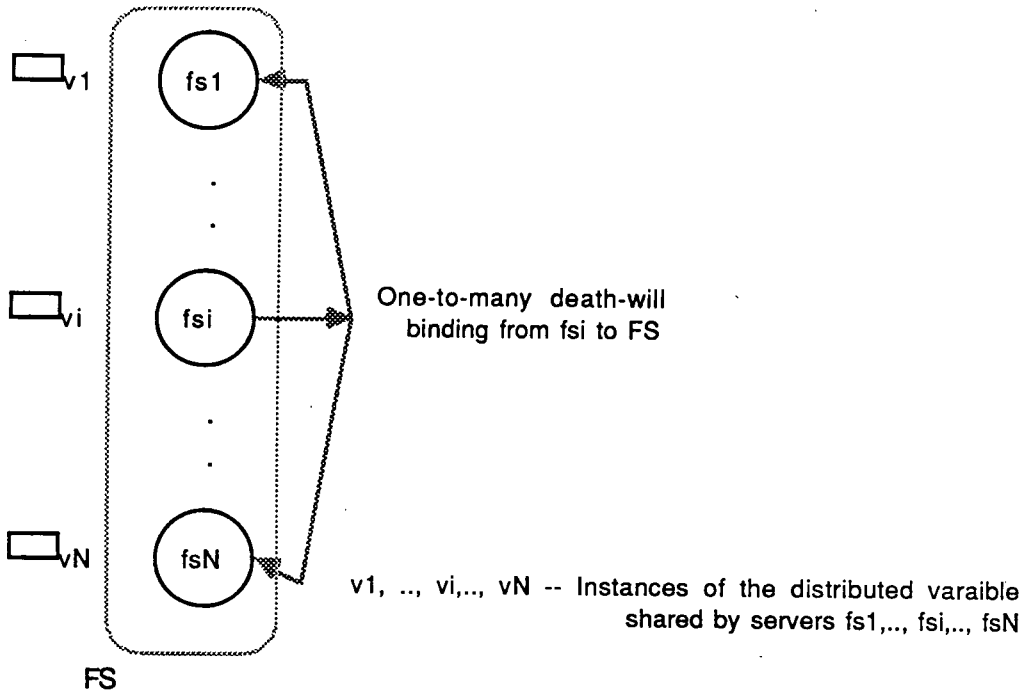


Figure 5.1: A structure to protect shared resources

5.3.1 Protection of shared resources against failures

Refer to Figures 5.1 and 5.2. Let fs_i be the member of the server group FS that has acquired the lock on a shared resource, i.e., in the LOCK_ACQUIRED state. The lock on the resource should be released at the occurrence of an external event that causes fs_i to unlock the resource, or a failure associated with fs_i. If these events are not handled properly, the lock may never be released and the resource is effectively lost. We propose a uniform technique based on the death-will scheme (refer to appendix A.1) which generates a message to be delivered to all members of the group when the lock is to be released.

Refer to Figure 5.1. As part of the lock acquisition protocol (i.e., to reach the LOCK_ACQUIRED state), fs_i arranges a death-will in favor of the group FS. A lock release message UNLOCK(rsrc_attr) is specified in the death-will (*rsrc_attr* refers to the attributes of the resource).

As part of the lock release protocol, fs_i updates its local state to UNLOCKED, sends the UN-

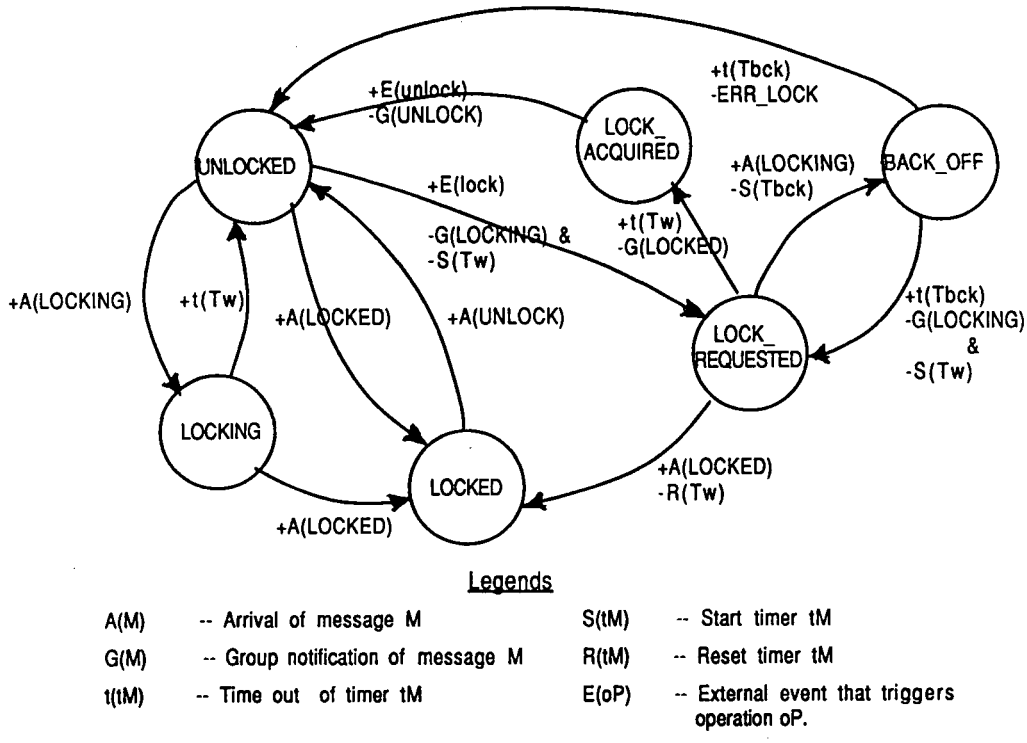


Figure 5.2: Finite State Machine (FSM) diagram of the lock acquisition protocol

LOCK(rsrc_attr) message to FS and cancels the death-will. If fs_i is destroyed, say due to an exception, or if the machine on which fs_i resides fails, the death-will is executed causing delivery of the UNLOCK(rsrc_attr) message to the other members of FS. On receipt of the lock release message, each of fs_k (k=1,2,..,N & $k \neq i$) updates its state to UNLOCKED, indicating the availability of the resource.

The above technique also enables recovery from the situation where fs_i is partitioned from the group FS due to communication break-down. On detecting such a failure, the other members of FS may recover the lock. However, if the break-down occurs in such a way that fs_i remains accessible to some members but partitioned from the rest of the members in FS, the partitioned members will eventually be delivered the UNLOCK(rsrc_attr) message upon which they may recover the lock. The problem arises when the network subsequently remerges since the members may no longer have a consistent view of the state of the resource. The implications of such an inconsistency are application-dependent.

It should be noted that in the case of an extrinsic resource (e.g., a printer) where an RPC has caused

fs_i to acquire the lock on the resource, fs_i should release the lock as part of the rollback initiated upon detection of failures up the RPC chain. However, lock release does not imply successful completion of the rollback which depends on the recoverability of the operations carried out on the resource after the lock was acquired (c.f. section 2.8.2).

Sample applications of the technique in structuring distributed services are described later in sections 5.5 and 5.6.

5.3.2 Collision control

Though priority assignment based on ordering among members of a server group [6] and time stamping [49] may, in principle, be used to resolve colliding lock requests, such techniques are more appropriate at higher levels such as data base systems. We introduce a simple scheme for collision detection and recovery that is sufficient for the situation at hand. The idea behind the scheme is that a member intending to access a shared resource multicasts its intention to the other members of the group and waits for a pre-determined period to see if the resource is available. If so, it locks the resource; otherwise, it backs off for a random interval of time and tries again.

Collision detection

When a member fs_i of FS wishes to access the resource, it first checks the local state of the resource (refer to Figure 5.2). If the state indicates that the resource is locked or in the process of being locked (LOCKED or LOCKING states), fs_i queues up its lock request. If the resource is not locked (UNLOCKED state), fs_i updates the local state to LOCK_REQUESTED and sends an intragroup message LOCKING(rsrc_attr) requesting a lock on the resource. fs_i then waits for an interval given by

$$T_w \geq 2 * T_{msg} \tag{5.1}$$

before proceeding to access the resource; T_{msg} should be sufficiently large to allow inter-machine message transfer. On receipt of this message, each of the fs_k's (k=1,2,..,N and $k \neq i$) updates its local state of the resource to LOCKING and withholds any subsequent attempt to lock the resource for the duration of timeout T_w. The use of timeout at fs_i guards against delays in message propagation, and hence enables fs_i to detect colliding requests to lock the resource. Time out at fs_k's enables subsequent release of the queued lock requests, if any, and restores the lock to a consistent state should fs_i fail during the lock acquisition phase. If during this interval, fs_i receives a LOCKING(rsrc_attr) message for the same resource from some other member, then a collision is said to have occurred. If a collision is detected, the members involved in the collision may execute an appropriate collision resolution protocol (see section 5.3.2). If no collision is detected during T_w, fs_i sends a LOCKED(rsrc_attr) intragroup message and updates its local state to LOCK_ACQUIRED; it may then initiate access to the resource. On receiving this message, each of fs_k's updates its local state to LOCKED.

In the above technique, each of the fs_i's 'senses' an on-going access to the shared resource. The technique is similar to the CSMA/CD technique used in Ethernet[52] but applied to higher level problems.

Depending on the requirements, a server may employ such a technique or a customized one to detect collisions. The choice may depend on such things as the number of potential contenders in the group, critical nature of colliding access to the resource and performance.

Collision resolution

The resolution is also modelled after the CSMA/CD. When fs_i detects a collision, it backs off for a random interval of time before trying to access the resource again. If it again detects a collision, it backs off for a longer interval. After a certain number (MAX_BCKOFF) of retries, it gives up and returns an ERR_LOCK error code; such failures are to be handled by higher level protocols depending on the

application.

The major advantages of our technique to handle collisions are that the technique is completely distributed and its generality of application. Sample applications of the technique in structuring distributed services are described later in sections 5.5, 5.6 and 5.7. The technique can also be used by system name servers during name registration [47].

The **Lock** primitive requires a group message to be sent after each timeout interval before the lock is acquired. Additionally, it requires creation of a death-will which consists of one group message and N replies from the members. The **Unlock** primitive causes the execution of the death-will and requires one group message.

5.4 'Create_instance' and 'Delete_instance' operations

The **Create_instance** and the **Delete_instance** operations on the ADSV require a merge protocol that allows a server to join a server group, take part in the group's activities and subsequently leave the group. Since every server group maintains one or more distributed shared variables (e.g., identity of a leader and lock on a resource), a merger of a new member into the group requires acquiring these variables from the other members of the group.

A server process P wishing to merge with a group *srvr_gid* goes through two logically distinct phases as described below:

5.4.1 Subscription phase

In this phase, P simply joins the group by executing a group management primitive of the form **join_group(srvr_gid)** [16]. This phase allows P to subscribe to messages (i.e., receive messages) delivered to the group and paves the way for state acquisition as described in the next section.

5.4.2 State acquisition phase

This phase allows P to merge with the rest of the group. Let V be the distributed state variable of the server group. Since P is not yet part of the group, its local instance of V will be uninitialized. P needs to employ protocols (which may be application-dependent) to derive a consistent view of V for the merge to take place. Suppose P is a name server process joining a name server group. After subscribing to the group, P should acquire the valid name bindings from other members of the group to merge into the group's activities. P may use a customized protocol to acquire the bindings [47].

Despite the application-dependent nature of the merge protocols, it is possible to outline some generic approaches that may embody such protocols:

Asynchronous probe and merge

P may asynchronously probe the rest of the group to acquire the state variable V. Group members may respond with their respective instances of V. P may construct its own instance of V from the responses. This approach is especially appropriate when the shared resource is of the intrinsic type.

Client-driven probe and merge

The acquisition of V is triggered by a client request to P to access the resource associated with V. P may then i) probe the rest of the group to acquire V, and ii) resort to a protocol to access the resource. This approach is appropriate if the resource is of the extrinsic type. An example is the service name binding information maintained by a name server process. When the latter joins the name server group, it may not possess the binding information for a given service, and on a client request to access the service, it may interact with the name server group to acquire the binding information. See [47] for details of this example.

Merge on the fly

P may contend for access to V without its local instance of V initialized. The protocols to acquire V are integrated into the resource access protocols. In other words, unlike the client-driven technique described above, there is no explicit state acquisition phase. This approach may be used for both intrinsic and extrinsic resource types. Suppose P is a spooler process in a program. P may contend with other members of the spooler group to use a shared printer without any prior knowledge of the printer state (see section 5.6). The contention mechanism used for access also allows P to acquire the state of the printer.

All of the above techniques may be supplemented by a passive *listen and merge* technique depending on the application. Since the subscription phase allows P to listen to messages destined to the group, P may acquire V based on such messages. However, this by itself is not a self-sufficient technique.

5.4.3 Handling first-ling

First-ling is a special case of solitude, and refers to the process P being the first member of a group. Whether P should explicitly check for first-ling during the merge phase is application-dependent. For example, if P is merging with a group that advertises a service, P should detect the first-ling condition whereby it may initialize its internal state and advertise the service on behalf of the group.

If the probe and merge techniques described in section 5.4.2 are used for state acquisition, P may detect first-ling by the lack of response to its probe messages. If P is the first to join the group, the logically distinct phases of subscription and merge collapse into a single phase whereby P may initialize V in an application-dependent manner. In the merge on the fly technique, first-ling handling is implicit in the protocols used to initialize V.

5.4.4 Exit from a group

The exit of a member from a group is the counterpart to merging with the group. The exit activity should not introduce inconsistencies in the distributed state maintained by the group. Typically, the member returns all the shared resources that it holds. This requires that the member notifies its exit to the other group members if it is holding any shared resource.

We now describe three extended examples to illustrate the use of the various algorithms and protocols described in sections 5.3 and 5.4.

5.5 Example 1: Distributed leadership in a server group

A leader of a server group coordinates the activities of the members of the group [20]. For example, when a client contacts the group for service, the leader of the group may respond providing the initial contact point to the client; subsequently, the client may interact with the leader to make connection to a resource. As another example, the leader may assume the role of a coordinator [38,5] to implement atomic transactions.

Many schemes have been proposed elsewhere concerning the management of leadership in a distributed system [20]. For example, arbitration of leadership among the members of the group may be based on some ordering among them. In this section, we describe a simple leadership management algorithm to illustrate our ADSV model.

Typically, when a leader intends to relinquish the leadership (based on some form of leadership transfer algorithm), it initiates a protocol as follows:

- The leader notifies other group members of its intention.
- The group members employ some form of a distributed election algorithm to elect the next leader.
- The leader hands over leadership to the elected member.

In terms of our ADSV model, the leadership is an intrinsic shared resource whose management is asynchronous to client activities. Inconsistency in the state of the leadership during an election or due to partial failures should be handled properly, otherwise the group may be left with more leaders than allowed or no leader at all.

5.5.1 A leadership management algorithm

In the algorithm, a server group FS is assumed to have a single leader (for simplicity). The algorithm fits into the framework of that described in section 5.3 (see Figures 5.1 and 5.2). Each of the fs_j's (j=1,2,..,N) maintains a local variable *ldrship_state* to describe the state of the leadership in FS as viewed by fs_j. *ldrship_state* constitutes the distributed state variable of the protocol, and hence is shared by all the group members. It is updated by election related messages received from the current leader and the contenders for leadership.

Election protocol

The election protocol is similar to that described in section 5.3.2 (refer to Figure 5.2) as far as collision handling is concerned but differs in the following respects: i) the initial contention mechanism, ii) the handling of solitude, and iii) the requirement to distinguish between failure-triggered and normal transfer of the leadership. An extra state ELECTING_NEW_LDR is also defined. These differences characterize the intrinsic nature of the resource, and are related to the leadership transfer phase.

Under normal condition, the leader, say fs_i, of the group FS is in LOCK_ACQUIRED state while the other members of FS are in the LOCKED state. When fs_i intends to relinquish leadership, it sends an intra-group message UNLOCK(ELECT_NEW_LDR) and moves to the ELECTING_NEW_LDR state. On detecting the intention of fs_i to relinquish leadership, each of the fs_k's ($k \neq i$) updates its state from LOCKED to UNLOCKED, and waits for a random interval of time T_r; this wait is required to avoid a *collision surge* among the members, i.e., members scrambling to become the leader and repeatedly

colliding with one another. If T_r expires in the UNLOCKED state, fs_k may contest for the leadership by sending an intra-group message LOCKING(CONTEST) and moving to the LOCK_REQUESTED state. fs_k also starts a timer T_w satisfying equation 5.1 to guard against collisions. If, in the UNLOCKED state, fs_k receives a LOCKING(CONTEST) message from another group member indicating it is already in the contest, fs_k updates the state to LOCKING and starts T_w. In this state, fs_k is prohibited from contesting. T_w in this case serves to allow fs_k a fair chance to contest for leadership should it expire with fs_k still in the LOCKING state. This may happen if other members give up the contest due to collisions or failures.

If, in the LOCK_REQUESTED state, fs_k gets a LOCKING(CONTEST) message from any other member indicating there is contention for the leadership, it backs off from the contest for a random interval of time T_r before contesting again. If T_w expires in the LOCK_REQUESTED state indicating that for the time being no one else is contesting for leadership, fs_k assumes leadership by sending a LOCKED(ELECT_ME) intra-group message and updates its state to LOCK_ACQUIRED. If it receives LOCKING(CONTEST) messages in this state indicating there are still contestants in the group, it sends a message LOCKED(ALRDY_ACQUIRED) to each of the contestants by one-to-one messages. These contestants, as well as others receiving the LOCKED(ELECT_ME) message, should then give up the contest and concede the leadership to fs_k, updating their respective states to LOCKED. The outgoing leader fs_i (if it exists) also moves to the LOCKED state on receiving the LOCKED(ELECT_ME) message. In addition, fs_k may transfer leadership related information from fs_i, if it exists, using one-to-one message exchanges. If fs_k is a first-ling, it initializes the information in an application-dependent manner.

Failure recovery

The recovery of leadership upon failures is identical to that described in section 5.3.1. It takes the form of delivering an UNLOCK(ELECT_NEW_LDR) message to the fs_k's in FS in case of failures associated with the leader fs_i. On receiving this message, each of the fs_k's takes part in the election algorithm described earlier. On detecting the non-existence of the old leader fs_i when attempting to transfer leadership related information from fs_i (see section 5.5.1), the new leader fs_k may asynchronously probe the members of FS to acquire these information. Thus if the leader fails, there is only a brief interruption of service. When a non-leader dies, it may not cause immediate concern as it does not hold any resource pertaining to the leadership, and hence the leadership is not in danger.

Handling solitude

To handle the case where fs_i is the sole member of the FS, fs_i monitors the on-going election activity while in the ELECTING_NEW_LDR state. If no activity is detected over a large time interval T_{Elect} ($\geq T_w$), fs_i assumes that there are no other members in FS and resumes the leadership.

Handling merging

A new member joining FS may asynchronously probe FS and acquire the state of the leadership from other members in FS (see section 5.4.2). First-ling may be detected by lack of response to the probes, in which case the new member may assume leadership by updating the local state to LOCK_ACQUIRED.

Resolving inconsistencies

Inconsistencies among the instances of *ldrship_state* may lead to either (i) the group FS having more than one leader, or (ii) no leader at all in FS. When a client requests service from FS, the client may detect situation (i) by the receipt of more than one response from the multiple leaders in FS, and

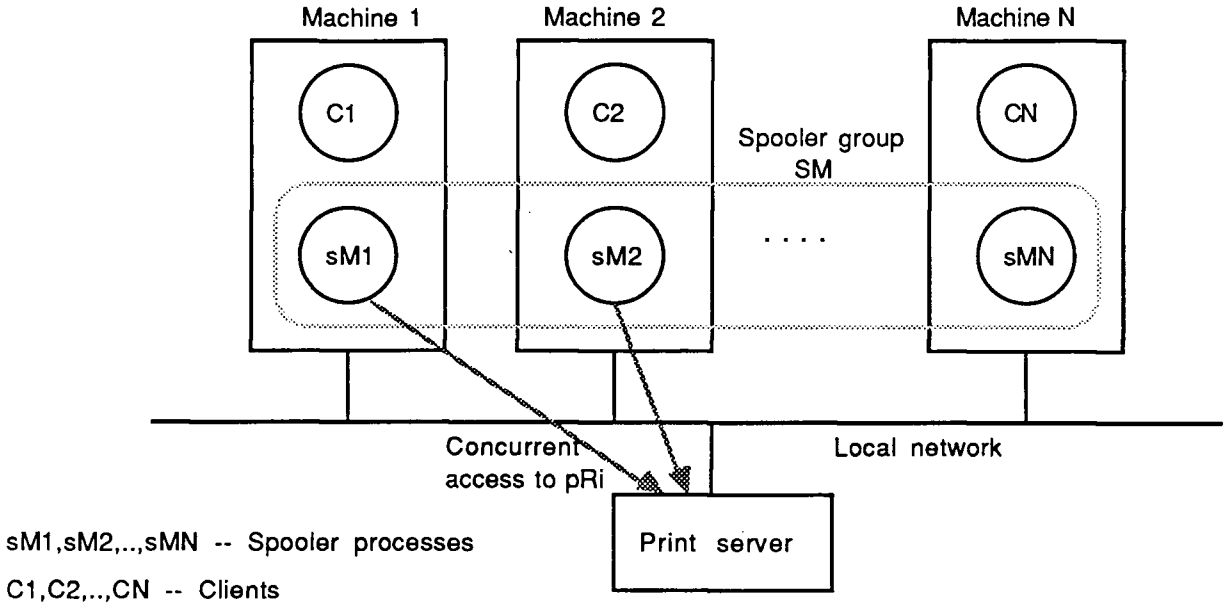


Figure 5.3: The structure of a distributed print spooler

situation (ii) by non-receipt of any response from FS. In either case, the client may force FS to elect a new leader.

5.6 Example 2: A distributed spooler

The function of a printer spooler is to serialize job requests from clients to a printer to avoid mixing up the output. A model of a distributed spooler is shown in Figure 5.3. In this model, a spooler process resides in every program that requires printing service. If we abstract the access to the printer as an access to a critical section, then mutual exclusion may be realized by acquiring a lock before the critical section is entered, and releasing the lock upon exit from the critical section.

5.6.1 A model of the spooler structure

sM_j is a spooler process which forms part of the printer interface to a client C_j. sM_j co-resides with C_j (on machine M_j). Thus the spooler is itself distributed across programs as the process group SM providing a unified printer interface to the clients. As opposed to the leadership discussed in the previous

example, the printer is an extrinsic resource since the requests originate from the clients.

Each sM_j maintains the state of a shared printer in a variable *prnt_state* which represents the current state of the printer as viewed by sM_j. The *prnt_state* variables across SM are updated at each **Lock** and **Unlock** request on the printer. Consistency (at a reasonable level) among the instances of *prnt_state* should be maintained, otherwise a mix up of the print jobs may result. Besides partial failures and collisions among the contenders during access to the printer, the spooler should also handle the death of the client C_j if it is holding the printer since the printer is an extrinsic resource.

5.6.2 A lock acquisition protocol

The process structure and the protocol are identical to that described in section 5.3.1. Suppose client C_j intends to lock the printer, it sends a message to sM_j requesting the state of the printer. If the local instance of *prnt_state* indicates that the printer is already locked (LOCKED state), sM_j returns the message ALREADY_LOCKED to C_j. If the printer is in the UNLOCKED state, sM_j interacts with the other members of SM by sending group messages, as per the protocol described in section 5.3.2, to acquire a lock on the printer.

Collision handling

If collision is detected during lock acquisition, sM_j backs off for a random interval of time, and tries to acquire the lock again. If failure persists after a certain number of retries, a failure code UNABLE_TO_LOCK is returned to trigger high level recovery by C_j[2]. If the lock is acquired successfully, sM_j updates *prnt_state* to LOCK_ACQUIRED and returns a success code to C_j which may then enter its critical section, i.e., output onto the printer.

[2]The return of the result of a **Lock** or **Unlock** request within a finite time provides clients with error information, viz., ALREADY_LOCKED and UNABLE_TO_LOCK with which they may build deadlock avoidance mechanisms[38].

Handling first-ling

If the state of the lock is uninitialized (i.e., sM_j has not yet merged with SM), sM_j may still go ahead with the above access protocol when requested by C_j to access the printer, and acquire the state of the printer during execution of the protocol (i.e., merge on the fly — see section 5.4.2). sM_j need not engage in an explicit first-ling detection protocol since the merge on the fly technique subsumes first-ling handling.

Lock recovery

The recovery mechanism is identical to that described in section 5.3.1, and is integrated into the lock release mechanism. It consists of delivering an UNLOCK message to the group SM in the event sM_j that is holding the printer fails. Since the printer is an extrinsic resource, lock recovery is also part of the rollback that may be necessary if an orphan on the printer needs to be killed to handle the failure of the call from C_j. The failure of any of the sM_k's which is not holding the printer does not cause immediate concern.

5.7 Example 3: Host identification

The host name (id) space in a network managed by a distributed kernel is a shared resource. Host name allocation to machines joining the network should be arbitrated to ensure uniqueness. Non-reusability of the allocated id's is another desirable property for reliability reasons [46]. A distributed scheme for the dynamic allocation of host id's satisfying these properties is given below:

5.7.1 Overview of the scheme

The kernel on each machine is considered as a server process. The collection of these kernels constitutes a well-known server group KRNL_GRP; the group is statically bound to the broadcast address of the underlying network. Then, in terms of the ADSV model, a kernel that gets instantiated on a new

machine merges into KRNL_GRP (**Create_instance** operation) through the following logically distinct phases: i) a subscription phase that allows the kernel to broadcast messages to KRNL_GRP and to receive messages addressed to its network address (because it does not yet have a host id) as well as those addressed to KRNL_GRP, and ii) a state acquisition phase by which the kernel acquires its host id from other members of KRNL_GRP. The subscription phase is implicit because the membership of the kernel in KRNL_GRP is usually a hard-wired feature of the kernel. The protocols employed by the kernel for the second phase (host id acquisition) are dependent on the properties required of the host id's.

The name space for host id's consists of a range of unsigned numbers between MIN_ID and MAX_ID which are the lowest and the highest values respectively of host id allowed in the system. Each of the machines joining the system is assigned an id from this name space based on the chronological order of the time of joining. Thus a machine joining the system is allocated an id whose numerical value is greater than that of a machine that joined at an earlier time. The system maintains a distributed state variable *highest_host_id* which represents the highest host id that has been assigned from the name space. To acquire a host id, the kernel **'Reads'** this variable, uses the next higher value as its host id, and increments (**Write** operation) this variable. 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).

5.7.2 Generation of host id's

Each host maintains a state-pair $(self\_id, highest\_host\_id)$, where *self_id* is the id of the host and *highest_host_id* represents the local instance of the distributed state variable maintained by the host; the instance represents the host's view of the highest host id across the entire system, and is updated

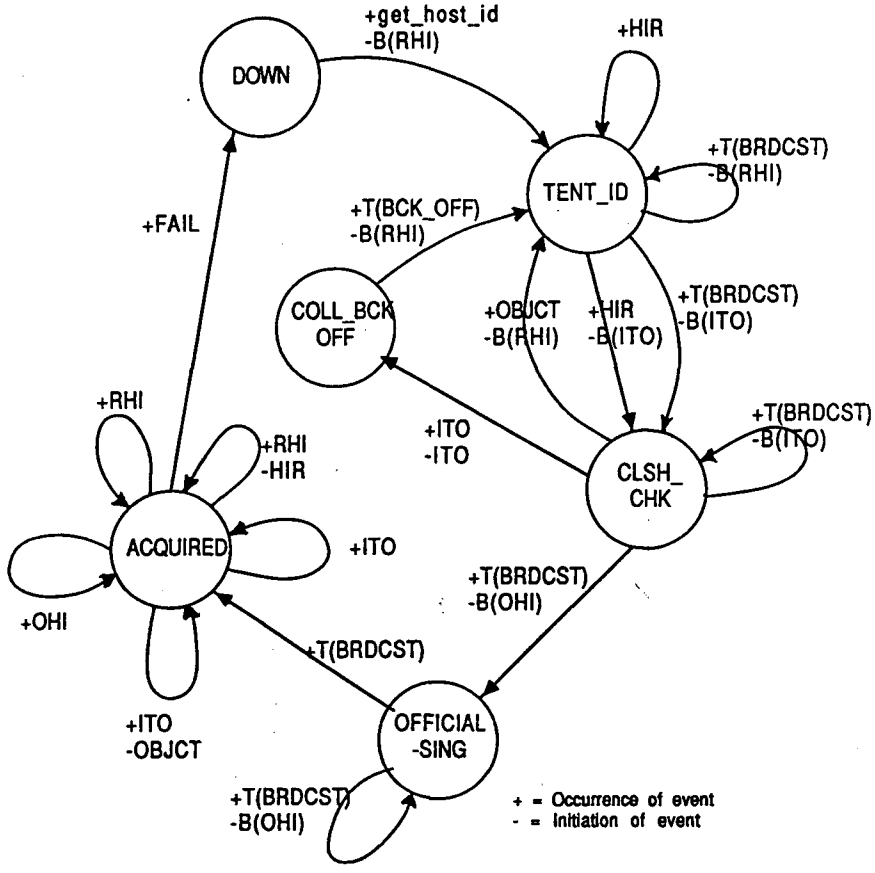


Figure 5.4: **FSM representation of the host identification protocol**

by broadcast messages from a joining machine. For any host,

$$(MIN\_ID \leq self\_id \leq highest\_host\_id \leq MAX\_ID).$$

Id generation is done in three stages (see Figure 5.4 for the FSM representation of the protocol):

Step1: Acquiring a tentative id

When a new machine wishes to join a system, it first acquires a tentative id from other hosts and bids to use it as its *self_id* in the system. To do so, it broadcasts one or more search messages looking for

the highest host id that has been assigned so far. During id acquisition, a machine uses its network address to allow other hosts to reach it. The machine specifies two integers Id_range1 and Id_range2 in the search message REQUEST_HOST_ID(Id_range1, Id_range2). These integers satisfy the following condition,

$$(0 \leq Id\_range1 \leq Id\_range2 \leq (MAX\_ID - MIN\_ID)).$$

The message requires all hosts with

$$(highest\_host\_id - Id\_range2) \leq self\_id \leq (highest\_host\_id - Id\_range1)$$

to send their respective local values of the state variable *highest_host_id* to the broadcasting machine. This allows for more than one response and increases the probability of receiving at least one correct reply. The joining machine filters the highest id from among the replies and uses it as the tentative host id. Id_range1 and Id_range2 specify a *polling window* that qualifies a selected set of hosts to respond to a particular search message. The size of this window is

$$[Id\_range2 - Id\_range1 + 1],$$

and the range of the host id space polled is

$$\{(highest\_host\_id - Id\_range2), (highest\_host\_id - Id\_range1)\}$$

In the simplest scheme, the joining machine initially starts with $Id\_range1 = 0$ and $Id\_range2 = 0$ implying a window of size 1. On receiving this message, the host whose $self\_id$ equals $highest\_host\_id$ responds with its value of $highest\_host\_id$. The initial search message may go unanswered if the machine that wishes to join is the first one in the system, or if none of the hosts has its $self\_id$ equal to the $highest\_host\_id$ (the latter is possible if the host whose $highest\_host\_id$ is equal to $self\_id$ has failed). Failure to get any response results in the machine rebroadcasting the search message with a different polling window until the entire host id space (MAX_ID - MIN_ID) is polled or sufficient number of

responses are received, whichever occurs earlier. If there is no response even after polling the entire host id space, the machine assumes that it is the first one joining the system, and takes on MIN_ID as the tentative id. The polling window is a control parameter of the protocol. A typical choice is to increase the window size logarithmically and slide it along the host id space. The range of id space covered by each window should be mutually exclusive to avoid receiving a reply more than once. (See [11] for details).

Having thus obtained the highest host id that has been assigned so far in the system (**Read** operation), the new machine then takes the next higher id as the tentative id.

Guarding against message losses

Message losses may result in inconsistencies among the instances of *highest_host_id*, or a machine choosing an id with insufficient information. This may lead to such error situations as i) more than one host think they are holding the highest host id in the system, and ii) the *highest_host_id* value received by a joining machine is not the highest host id that has been allocated. The effective outcome is that the joining machine will select an incorrect tentative id.

To guard against such inconsistencies, the new machine fields a certain number of replies N_h by polling over a certain range of the host id space before selection of the highest host id from among the replies received (see [11] for details). The machine may specify R = $(INTEGER, N_h)$ (refer to section 5.2) for the underlying broadcast communication used for polling, which suffices for operational consistency. The machine then resorts to a clash resolution protocol described below.

5.7.3 Step2: Resolving id clashes

After acquiring a tentative *self_id*, the machine specifies R = $(INTEGER, 1)$ and broadcasts an IS_THERE_OBJECTION message containing the tentative id. Any host whose id either clashes with or is higher than the broadcast tentative id raises an objection by replying with an OBJECTION message.

If an objection to the bid is received indicating that the id has already been assigned, the machine recompiles another tentative id and rechecks for objections. When there is no objection from other hosts after a certain number of broadcast-based probe messages, the machine acquires the id.

5.7.4 Step3: Officialization of the id

After affirming there is no objection to the id, the kernel initializes its local instance of *highest_host_id* to the acquired id. It then announces its entry into the system by broadcasting its *self_id* value in an OFFICIAL_HOST_ID message with R = $(INTEGER, 0)$, and thus becomes an official host. Each host which receives this message updates its local value of the state variable *highest_host_id*. The officialization thus constitutes the **Write** operation on the *highest_host_id* shared variable.

The host id's thus generated are, with a high degree of probability, unique and non-reusable. Note that a host does not maintain any information about the failure status of machines joining or already in the system. This results in better efficiency of the protocol though this may lead to unused id's; the latter is a non-problem since the id space is usually large.

5.7.5 Collision resolution

A collision occurs when two or more machines try to establish their host id's at the same time. This is the case for example when a machine sending the IS_THERE_OBJECTION message receives one from another machine. If a joining machine detects a collision during the acquisition of tentative id, it backs off for a random interval of time and tries again. If it detects a collision after it has acquired a tentative id but before it has officialized it, it sends a DEFER_HOST_ID message to the colliding machine advising the latter to back off as before.

Collisions arising due to more than one machine trying to join the system independently at the same time are rare and usually resolvable in a few (1 or 2) retries. However, a single external event may cause a large number of machines to scramble for a host id to join the system (e.g., an electrical power

bump). Such machines may collide with one another repeatedly causing a collision surge, which may lead to most of the machines being unable to join the system because of the congestion experienced in accessing the *highest_host_id* variable. Mechanisms to control this congestion may be based on the surge avoidance technique based on an initial wait period, as described in section 5.5.1.

5.7.6 Simulated behavior of the host id generation scheme

The protocol behavior has been studied under a simulated dynamic environment in which a varying number of machines join the system, operate for a while and exit [11]. The aim of the study was to assess how well the protocol enforces the uniqueness and the non-reusability of host id's. The quantitative indices were the probability with which the uniqueness and the non-reusability of the host id's were violated. The simulation parameters were the number of machines in the system, their failure rates and the message loss probability, mean and variance of the running and the down times of the machines. Even under strenuous conditions, the protocol was found to be satisfactory in terms of performance, and the probability of a host acquiring an id that has been used before was practically zero.

5.8 Summary

In this chapter, we have analyzed the consistency requirements on shared variables (associated with shared resources) such as information on name bindings and leadership within a server group. Since the consistency requirements of such variables can be relaxed, the access operations on the variables may be simpler than those used with file systems and databases. Along this direction, we have provided simple algorithms and protocols to realize the access operations. The algorithms enforce consistency of the variables to the extent required by the underlying resource. The procedural realization of the algorithms uses group communication among the various server members.

Though Cheriton's problem-oriented shared memory concept [13] is somewhat similar to our ADSV concept, Cheriton does not address the mutual exclusion requirements in access to the shared memory.

Also, it is not clear how the access operations on the shared memory are procedurally realized.

Chapter 6

Conclusions and future research

We have presented in this thesis models of communication abstractions for reliable client-server communication in distributed programs. The models use new techniques to mask failures during the communication, and are largely complete as far as failure transparency is concerned. In other words, the models are not just recovery techniques; when supported by other features (see sections 6.2.2 and 6.2.3 below), they may readily be transformed into full-fledged communication abstractions to support reliable distributed programming.

In this chapter, we summarize the contributions of our thesis and sketch directions in which the work might be extended for further research.

6.1 Summary of contributions

The contributions of our thesis take two forms: i) Identification of a new concept for failure recovery — the application-driven approach, and ii) Formulation of new techniques incorporating the concept to recover from failures. These contributions are pointed out below:

6.1.1 Application-driven approach to failure recovery

The thesis of the dissertation is that the use of application-level information by the failure masking algorithms of communication abstractions (RPC and ADSV in our case) may simplify the algorithms and

enhance their efficiency. This is based on the premise that many distributed applications can inherently tolerate failures under certain situations. Thus, the ordering and atomicity constraints on events in the communication layer need not be incorporated into this layer but may be specified by the application layer above it. This application-driven approach softens the boundaries of the communication layer (RPC and ADSV in our case), and allows the failure masking algorithms in the communication layer to exploit the characteristics of the application layer above it and the special features of the message layer below it. The approach allows relaxation of the constraints under which the algorithms may have to operate if such characteristics are not known. The relaxation significantly simplifies the algorithms and optimizes their performance. Typically, the underlying message transport layer need not attempt to mask failures rigorously (e.g., the V kernel). Instead, the failure masking algorithms in the communication layer use the information about when the application layer can tolerate failures, and use the information to tackle the effects of the unreliable message transport underlying the algorithms. If the information is not available, as in many systems, the algorithms require more functionalities of the underlying message transport layer such as atomic and ordered message delivery; these properties make the message transport layer complex and inefficient.

The above unifying concept forms the backbone of the various algorithms and protocols provided in the thesis to mask failures during client-server and intra-server communications. To our knowledge, no other work on failure transparency has systematically attempted to design failure recovery algorithms using the application-driven approach as presented in this thesis. In fact, as pointed out in the various chapters of the thesis, many existing works do not use this approach at all.

We now outline how the various recovery techniques presented in the thesis reflect the application-driven approach:

6.1.2 Orphan adoption in RPC's

Orphan handling is an important issue in RPCs, and has been dealt with to different extents in some earlier works. Nelson's [39] and Shrivastava's [53] works kill orphans on the ground that the orphans waste system resources. In ARGUS, orphans are killed for the reason that they introduce inconsistencies in the system state besides wasting system resources [56]; such an orphan killing is usually associated with a rollback of the environment.

We agree with the ARGUS view that orphans are more harmful than merely wasting resources. However, we take a new approach of adopting the orphans rather than killing them. The adoption approach is preferable because orphan killing and the underlying rollback is usually expensive, and sometimes not meaningful (c.f. 2.8.2). Additionally, the adoption approach saves any work already completed. We have incorporated orphan adoption into two replication schemes:

Primary-secondary scheme

In the primary-secondary scheme of replicating a server, at any time one of the replicas of the server acts as the primary and executes client calls while the other replicas stand by as secondaries. When the primary fails, failure recovery is initiated whereby one of the secondaries restarts as the primary to continue the server execution from the most recent checkpoint before the erstwhile primary failed. The run-time system uses event logs and call re-executions to realize adoption of the orphan generated by the failure (rollback is used only where essential). The re-executions of calls — both connection-less and connection-oriented — by the server arise when its client fails and recovers or when message orphans occur due to re-transmissions of call request messages. The re-executions are based on the idempotency properties of the calls (application level information).

Replicated execution of a server

In the replicated execution of a server, a call from a client on the server is executed by more than one replica of the server at the same time. When any of the executing replicas fails, the orphan generated by the failure is adopted by the surviving replicas and the call completed without explicit failure recovery initiated by the run-time system. The failure is transparent to the client because the replicated execution of the server is itself transparent. The scheme makes use of the idempotency properties of calls to relax the atomicity and the ordering constraints on the calls but still maintain the replicas in identical states. The relaxation of the constraints allows the use of a weak form of group communication in the underlying message transport layer (e.g., V kernel). Connection-less calls on a server are simply re-executed by the server irrespective of which replica requests the call. For connection-oriented calls, the scheme uses a combination of call re-executions, call replay from a server, and lateral coordination among the replicas to maintain them in identical states.

Call re-executions in both the schemes increase the level of failure tolerance in the program. We have also introduced quantitative indices to analyze the underlying algorithms and compare them with related works.

6.1.3 Access to shared variables

In distributed operating systems, the traditional view of resources as files and databases has to be extended to include a new class of resources such as information on name bindings, distributed load and leadership within a service group [13,28]. The consistency constraints on such operating system resources (or variables) need not be as strong as that on the information contents of a file or a database. Hence the access operations on the variables may be simpler than those used with file systems and databases. Along this direction, we have introduced a conceptual framework (or abstraction), which we refer to as application-driven shared variable, to govern access operations on the variables. The

abstraction is similar to the well-known shared variable concept, however the underlying algorithms and protocols used to realize the access operations on a variable enforce consistency of the variable to the extent required by the application. We have identified some simple algorithms and protocols useful for accessing the variables. The procedural realization of the algorithms uses intra-server group communication. Since the algorithms are not executed as atomic actions, a high degree of concurrency in access to the shared variables is possible without sacrificing the correctness of the access operations.

These new ideas have been incorporated into complete communication models and described in the thesis.

6.2 Future research issues

As a logical follow-up to the research presented in this thesis, there are many interesting issues for further investigation. We discuss some of them below:

6.2.1 Implementation issues

Prototype implementations to validate our approach to failure recovery and the underlying techniques have been made on the V kernel running on a network of SUN workstations interconnected by Ethernet. The implementations confirmed many of our ideas, and also gave insight into issues which were not clear at the abstraction level. For more details of the implementations, see [44,43]. The scope of the implementations may be broadened to obtain more insight into the approach and the underlying techniques; also, they may form a backbone for constructing fault-tolerant distributed operating systems. The impact of the recovery techniques on the normal case efficiency of the calls, efficiency of the actual recovery upon failures, and performance tuning of the implementations (including timeouts) are specific issues to be addressed in the future. It will be interesting to analyze the impact of such implementation issues on the high level communication models we presented in the thesis.

6.2.2 Incorporating communication abstractions into a language

We have generally ignored the question of incorporating the communication models — the failure semantics of the models and the underlying techniques — into a language because the models are language independent. However, to use the models in a system, they should be embedded into a system language. There are two ways to address the issue: One way is to incorporate the models into the underlying run-time support of existing IPC constructs in languages such as the rendezvous construct of Ada [21] and the module-based IPC construct of Modula [57]. Specific issues remain such as how to map unrecoverable failures (c.f. section 2.8.2) into exceptions at the language level, and the language level tools to deal with such exceptions. The exception handling constructs provided in Mesa [36] and CLU [32] may offer suggestions in this direction. The second way is to design new IPC constructs — along the lines of ARGUS at MIT [31] and SR at the University of Arizona [50] — in a distributed programming language to incorporate our communication models. The issue to be looked at in this context relates to the exception handling constructs of the language. It is unclear at this stage which is a better approach.

6.2.3 Automatic stub generation

Stubs are used in the run-time system for interfacing between a language level invocation of an IPC and the underlying procedural realisation of the IPC. The usual role of the stubs is to marshal and unmarshal arguments specified in the language level communication constructs, and to activate the underlying protocols. Though we have generated the stubs manually in our prototype implementations, a complete system would require automatic generation of the stubs. Techniques for stub generation have been studied elsewhere [39,18,22]. Since our IPC models use application-level information in the run-time system, some ways should be devised to systematically pass this information from the language level invocations into the stubs (c.f. section 2.10). These hooks in the application-level code should have a generic structure. Thus extensions may be required in the stub generation techniques.

6.2.4 Remote stable storage

The issue of a stable storage is fundamental to any recovery technique. Researchers have so far assumed some form of a local storage on machines upon which the stable storage abstraction is built [30,31]. However, this issue needs to be examined in the context of contemporary system architectures consisting of diskless machines serviced by a few file servers over a network (like the system we considered in the thesis). The majority of the machines in such systems do not have local disks but servers can still run on them and communicate with clients [14]. In some of our algorithms, we have used the buffer space in the client's run-time system instead of a stable storage to store recovery information. Though the method is workable, its generality should be studied in depth. A general solution might be to build the abstraction of a stable storage on the remote disks. However, we get into a bootstrap problem — since the physical storage is itself remote thereby susceptible to partial failures, the implementation of a remote stable storage itself requires a stable storage!! Thus, there are interesting problems in this direction.

6.3 Concluding remarks

We have shown in the thesis how the concept of application-driven approach to failure recovery can be useful in masking failures during client-server communications. New recovery techniques based on the concept have been presented. The application-driven approach serves to simplify the underlying protocols and increase the failure tolerance of the program. Contemporary system architectures with high inter-machine communication speeds and inexpensive broadcast facility allow natural realization of the proposed recovery schemes. We have conveyed the message in the thesis that the application-driven approach is viable for constructing fault-tolerant systems. The viability of the approach to provide other functionalities such as security and authentication warrants further research.

Bibliography

[1] **SUN Network Services — System Administration for the SUN Workstation.** Feb. '86.

[2] M. Ahamad and A.J.Bernstein. **Multicast communication in UNIX 4.2 BSD.** In *5-th International Conference on Distributed Computing Systems*, pages 80–87, IEEE CS, May '85.

[3] G. T. Almes. **The impact of language and system on remote procedure call design.** In *6-th International Conference on Distributed Computing Systems*, pages 414–421, IEEE CS, Cambridge, MA, May '86.

[4] S. Atkins. **Exception Handling Mechanisms in Distributed Operating Systems.** Technical Report, University of British Columbia, Jan. '86.

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

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

[7] 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.

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

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

[10] 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, New Orleans, Dec. '86.

[11] S. T. Chanson and K. Ravindran. **Host identification in reliable distributed kernels.** Technical Report 86-5, Dept. of Computer Science, Univ. of British Columbia, Feb. '86. (Submitted for publication).

[12] D. R. Cheriton. **Local Networking and internetworking in the V-System.** In *8-th Symposium on Data Communication*, pages 9–16, ACM SIGCOMM, Oct. '83.

[13] 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, Cambridge, MA, May '86.

[14] D. R. Cheriton. **V-Kernel: A software base for distributed systems.** *IEEE Software*, 1(2):19–42, April '84.

[15] 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.

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

[17] E. C. Cooper. **Replicated Distributed Programs.** In *10-th Symposium on Operating System Principles*, pages 63–78, ACM SIGOPS, Orcas Island, Dec. '85.

[18] E. C. Cooper. **Replicated Distributed Programs.** Technical Report UCB/CSD/85/231, University of California, Berkeley, May '85.

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

[20] H. Garcia-Molina. **Elections in a distributed computing system.** *IEEE Transactions on Computers*, C-31(1):48–59, Jan. '82.

[21] N. Gehani, editor. **Concurrent Programming in ADA.** Prentice-Hall, '84.

[22] P. B. Gibbons. **A Stub Generator for Multilanguage RPC in Heterogeneous Environments.** *IEEE Transactions on Software Engineering*, SE-13(1):77–87, Jan. '87.

[23] T. E. Gray. **Two years of Network Transparency: Lessons Learned from LOCUS.** Technical Report Vol.13,No.2, University of California, Los Angeles, Spring Quarter '85.

[24] M. Herilihy and B. Liskov. **A Value Transmission method for Abstract Data Types.** *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, Oct. '82.

[25] T. A. Joseph and K. P. Birman. **Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems.** *ACM Transactions on Computer Systems*, 4(1):55–70, Feb. '86.

[26] J.Sventek , et al. **Token ring Local Area Network – a comparison of experimental and theoretical performance.** In *Symposium on Computer Networking*, pages 51–56, IEEE CS, Dec. '83.

[27] B. W. Kernighan, editor. **The 'C' Programming Language.** Prentice-Hall, '78.

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

[29] 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, Los Angeles, Oct. '84.

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

[31] 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.

[32] B. Liskov and A. Snyder. **Exception Handling in CLU.** *IEEE Transactions on Software Engineering*, SE-5(6):546–558, Nov. '79.

[33] 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.

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

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

[36] J. G. Mitchell , et al. **MESA Language Manual.** Technical Report CSL79-3, Xerox Palo Alto Research Center, April '79.

[37] S. J. Mullender and P. M. B. Vitayani. **Distributed match-making for processes in Computer Networks.** *Operating Systems Review*, 20(2):54–64, April '86.

[38] N. Natarajan. **Communication and Synchronization primitives for Distributed Programs.** *IEEE Transactions on Software Engineering*, SE-11(4):396–416, April '85.

[39] B. J. Nelson. **Remote Procedure Call.** Technical Report CMU-CS-81-119A, Carnegie Mellon University, May '81.

[40] D. C. Oppen and Y. K. Dalal. **The Clearinghouse: A decentralised agent for Locating Named Objects in a Distributed Environment.** Technical Report OPD-T8103, Xerox Office Products Division, Oct. '81.

[41] Peterson and Silberschatz, editors. **Operating System Concepts.** Prentice-Hall, '85.

[42] 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.

[43] K. Ravindran and S. T. Chanson. **Handling Call Idempotency Issues in Replicated Distributed Programs.** Technical Report 86-21 (to be published), Dept. of Computer Science, Univ. of British Columbia, Nov. '86.

[44] 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.

[45] K. Ravindran and S. T. Chanson. **Process alias-based structuring techniques for Distributed Computing Systems.** In *6-th International Conference on Distributed Computing Systems*, pages 355–363, IEEE CS, Cambridge, May '86.

[46] 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, Los Angeles, Jan. '86.

[47] K. Ravindran and S. T. Chanson. **Structuring Reliable Interactions in distributed server architectures.** Technical Report 86-13, Dept. of Computer Science, Univ. of British Columbia, June '86. (Under review for publication in IEEE Transactions on Computers).

[48] K. Ravindran, S. T. Chanson, and K.K. Ramakrishnan. **Application-driven failure semantics of Interprocess Communication in Distributed Programs.** Technical Report 87-3A (to be published), Dept. of Computer Science, Univ. of British Columbia, Jan. '87.

[49] D. P. Reed. **Implementing atomic actions on decentralised data.** *ACM Transactions on Computer Systems*, 1(1):3–23, Feb. '83.

[50] R. D. Schlichting and T. D. M. Purdin. **Failure Handling in Distributed Programming Languages.** In *5-th Symposium on Reliability in Distributed Software and Database Systems*, pages 59–66, IEEE CS, Los Angeles, Jan. '86.

[51] R. D. Schlichting and F. B. Schneider. **Fail-stop processors: An approach to designing Fault-tolerant Computing Systems.** *ACM Transactions on Computer Systems*, 1(3):222–238, Aug. '83.

[52] J. F. Shoch , et al. **Evolution of the Ethernet local computer network.** *IEEE Computer*, 15(8):10–27, Aug. '82.

[53] 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.

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

[55] D. B. Terry. **Caching Hints in Distributed Systems.** *IEEE Transactions on Software Engineering*, SE-13(1):48–54, Jan. '87.

[56] E. F. Walker. **Orphan detection in the Argus system.** Technical Report MIT/LCS/TR-326, Laboratory for Computer Science, MIT, June '84.

[57] N. Wirth. **Modula: A language for modular multiprogramming.** *Software Practice and Experience*, 7(1), Jan. '77.

Appendix A

Death-will abstraction

A process is associated with an abstract property of *shout and fail*. The property allows the process to generate a message whenever it fails or observes a communication break-down. The message causes a state transition which enables all concerned processes of the program to detect the failure.

To realize the abstract property of shout and fail, we introduce a new element called a *process alias* [45] in the structure used for realizing the RPC. A process alias is an ancillary process created by a client and made to reside in the address space of a server. It does not have self-existence. When the client dies, its alias terminates after delivering a message to the server. When the server dies, the alias terminates after delivering a message to the client. The process alias is used in the RPC as follows [46] (see Figure A.1): The client prepares a death-will containing a list of processes that are to be notified upon its death. Typically, when the client opens a connection to a server, it includes the server in its death-will list. It also optionally specifies the message to be delivered to the processes on the list and registers the death-will with the run-time system. The system sets up an alias at the client site and one alias for each of the processes on the death-will list to be resident at the site of the associated process. The death-will message is deposited with these remote aliases. Each of the remote aliases engages in a protocol that exchanges keep-alive messages with the alias at the client site. When the client dies, the system destroys the local alias and dispatches an EXECUTE_DEATH_WILL message to each of the remote

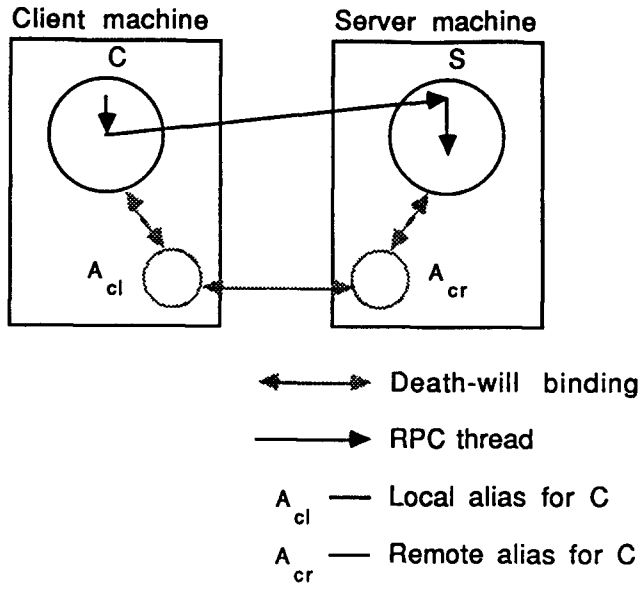


Figure A.1: Structural realization of RPC using an alias process

aliases. Upon detection of the client's death, the remote aliases deliver the death-will message to the respective processes and destroy themselves. A server handles the message of type DEATH_WILL_MSG by resorting to appropriate recovery such as reclaiming the internal resources committed for the client.

When the client site fails, the client alias at the server site detects this (absence of keep-alive messages), delivers the death-will message to the server and terminates. If the server site fails, the client alias at the client site detects this, excludes the server from its polling list, and deliver an ALIAS_DEAD message to the client. Network failures may also be detected and failure messages delivered in a similar way.

A.1 Extension of death-will scheme to RRPC

Suppose a caller makes an RRPC on a callee. The caller prepares a death-will containing a (error) message and registers it with the system to be delivered to the callee group should the caller fail. The system sets up an alias at the caller's site and one alias at each of the member sites by sending a group message. The death-will message is deposited with these remote aliases (see Figure A.2). The local alias engages in an asymmetric failure detection protocol by periodically sending a keep-alive message

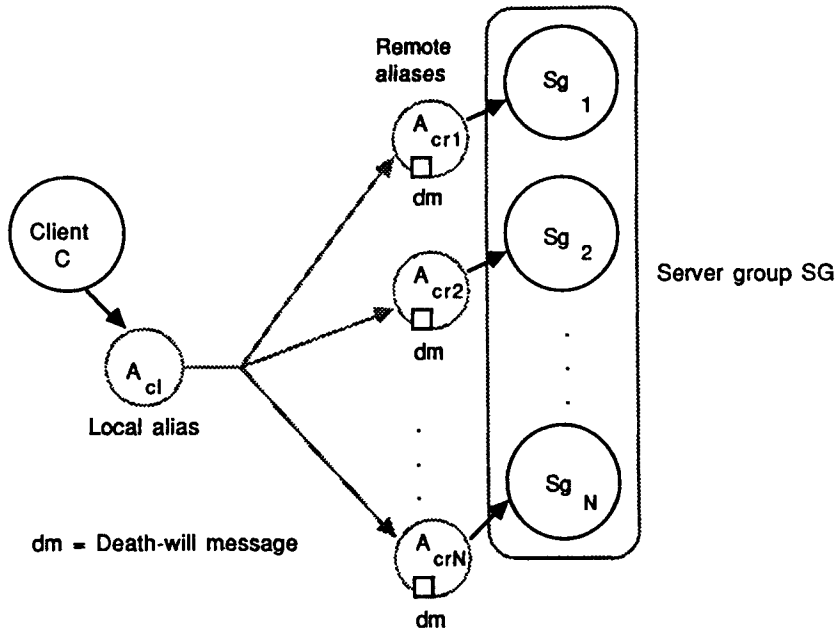


Figure A.2: Death-will abstraction applied to process groups

I_AM_HERE to the remote aliases. When the caller dies, the system destroys the local alias and sends a group message EXECUTE_DEATH_WILL to the callee group. If the caller site fails, the caller's alias at a member site detects this by the lack of any message from its peer. Network failures partitioning the members from the caller are similarly detected due to the breakdown in communication. On thus detecting failure, the caller's alias at a member site delivers the death-will message to the member and destroys itself. The member may then structure its recovery accordingly. The caller may cancel its death-will at any time in which case the aliases are simply destroyed.

List of publications/reports

1. K. Ravindran and S. T. Chanson, *Orphan adoption-based failure recovery in remote procedure calls*, Technical Report #87-3, Dept. of Computer Science, Univ. of British Columbia, Jan.'87.

2. K. Ravindran, S. T. Chanson and K. K. Ramakrishnan, *Application-driven failure semantics of interprocess communication for distributed programs*, Dept. of Computer Science, Univ. of British Columbia, Jan. '87.

3. S. T. Chanson and K.Ravindran, *A distributed kernel model for reliable group communication*, Proc. of the IEEE-CS Symposium on Real-Time Systems, New Orleans, Dec.'86, pp.138-146.

4. K.Ravindran and S. T. Chanson, *Structuring reliable interactions in Distributed Server Architectures*, Technical Report #86-13, Dept. of Computer Science, Univ. of British Columbia, June '86 (Under review for publication in IEEE Transactions on Computers).

5. K.Ravindran and S. T. Chanson, *Process alias-based structuring techniques for distributed computing systems,* Proc. of the 6th IEEE Computer Society (CS) Symposium on Distributed Computing Systems, Cambridge, Massachussettes, May '86, pp.355-363.

6. S. T. Chanson and K.Ravindran, *Host identification in reliable distributed kernels,* Technical Report 86-5, Dept. of Computer Science, Univ. of British Columbia, Feb.'86 (Under review for publication in the Computer Networks and ISDN Systems journal).

7. K.Ravindran and S. T. Chanson, *State inconsistency issues in local area network based distributed kernels,* Proc. of the 5th IEEE-CS Symposium on Reliability in Distributed Software and Database Systems, Los Angeles, Jan.86, pp.188-195.

8. S. T. Chanson, K. Ravindran, and S. Atkins, *A performance evaluation of the ARPANET Transmission Control Protocol in a Local Area Network environment,* Special issue of the Canadian Journal of Information Processing and Operations Research on Performance Evaluation of Computer Systems, Vol.23, No.3, Aug.'85, pp.294-329.

9. K. Ravindran, N. Rajan, G. S. Raman and V. K. Agarawal, *Some experiences on microcomputer development tools*, Proc. of the ISMM conf. on Mini- and Micro-computers and their applications, San Fransisco, May '83, pp.87-91.

10. N. Rajan, K. Ravindran, and P. S. Goel, *Design and analysis of a Pulse Width Pulse Frequency Modulator for satellite attitude control*, Proc. of the IFAC Symposium, IIT, New Delhi (India), Feb.'82, pp.36-39.

11. K. Ravindran and A. Krishnan, *A hybrid Simulation of on-orbit acquisition of APPLE satellite* Proc. of the Servo Systems Conference, Vikhram Sarabhai Space Centre, Trivandrum (India), Feb.'80.