INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame. If copyrighted materials were deleted you will find a target note listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in "sectioning" the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.

University
Microfilms
International

300 N. ZEEB RD., ANN ARBOR, MI 48106

8204168

Nelson, Bruce Jay

REMOTE PROCEDURE CALL

Carnegie-Mellon University

PH.D. 1981

University
Microfilms
International 300 N. Zeeb Road, Ann Arbor, MI 48106

Copyright 1982
by
Nelson, Bruce Jay
All Rights Reserved

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark __√__.

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy __✓__
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages __✓__
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print _____
11. Page(s) __________ lacking when material received, and not available from school or author.
12. Page(s) __________ seem to be missing in numbering only as text follows.
13. Two pages numbered __________. Text follows.
14. Curling and wrinkled pages _____
15. Other ______________________________

University Microfilms International

Remote Procedure Call

Bruce Jay Nelson

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania

May 3, 1981 11:59 PM

Carnegie-Mellon University

MELLON INSTITUTE OF SCIENCE

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Doctor of Philosophy

TITLE REMOTE PROCEDURE CALL

PRESENTED BY Bruce Jay Nelson

ACCEPTED BY THE DEPARTMENT OF Computer Science

[signature] MAJOR PROFESSOR 5/11/81 DATE

[signature] DEPARTMENT HEAD 5/13/81 DATE

APPROVED BY

[signature] DEAN 9/2/81 DATE

Abstract

Remote procedure call is the synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel. The thesis of this dissertation is that remote procedure call (RPC) is a satisfactory and efficient programming language primitive for constructing distributed systems.

A survey of existing remote procedure mechanisms shows that past RPC efforts are weak in addressing the five crucial issues: uniform call semantics, binding and configuration, strong typechecking, parameter functionality, and concurrency and exception control. The body of the dissertation elaborates these issues and defines a set of corresponding *essential properties* for RPC mechanisms. These properties must be satisfied by any RPC mechanism that is fully and uniformly integrated into a programming language for a homogeneous distributed system. Uniform integration is necessary to meet the dissertation's fundamental goal of syntactic and semantic *transparency* for local and remote procedures. Transparency is important so that programmers need not concern themselves with the physical distribution of their programs.

In addition to these essential language properties, a number of *pleasant properties* are introduced that ease the work of distributed programming. These pleasant properties are good performance, sound remote interface design, atomic transactions, respect for autonomy, type translation, and remote debugging.

With the essential and pleasant properties broadly explored, the detailed design of an RPC mechanism that satisfies all of the essential properties and the performance property is presented. Two design approaches are used: The first assumes full programming language support and involves changes to the language's compiler and binder. The second involves no language changes, but uses a separate translator—a source-to-source RPC compiler—to implement the same functionality.

Design decisions crucial to the efficiency of the mechanism are made using a set of RPC performance lessons. These lessons are based on the empirical performance evaluation of a sequence of five working RPC mechanisms, each one faster than its predecessor. Some expected results about the costs of parameter copying, process switching, and runtime type manipulation are confirmed; a surprising result about the price of protocol layering is presented as well. These performance lessons, applied in concert, reduce the roundtrip time for a remote procedure call by a remarkable factor of 35. For moderate speed personal computers communicating over an Ethernet, for example, a simple remote call takes 800 microseconds; on a higher speed personal computer, the same remote call takes 149 microseconds. In both cases the remote call takes about 20 times longer than the same local call. This represents a substantial performance improvement over other operational RPC mechanisms.

Key words and phrases: communication primitives, computer networks, distributed computing, interprocess communication, message passing, naming and binding, procedure call, programming languages, protocols, remote procedure call, software performance evaluation.

CR categories: 3.81, 4.12, 4.22, 4.6.

To the islands of the South Pacific

Remote Procedure Call

Remote Procedure Call

Report number CMU-CS-81-119.

© Copyright 1981 by Bruce Jay Nelson. All rights reserved.

This dissertation was submitted to Carnegie-Mellon University in partial fulfillment of the degree requirements for the Doctor of Philosophy.

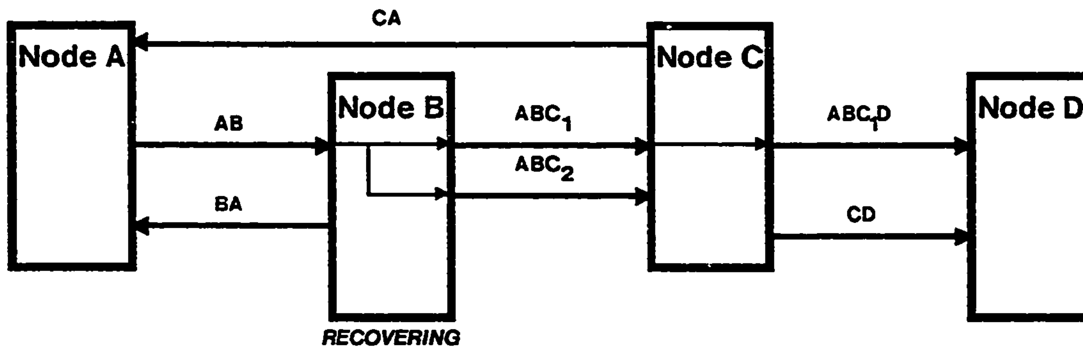
The research in this dissertation was supported by the following sources: the Carnegie-Mellon University Computer Science Department, the Hertz Foundation, the Xerox Corporation, and the Defense Advanced Research Projects Agency under contract F33615-78-C-1551.

The title illustration shows a four-node distributed computer system communicating with remote procedure calls.

This dissertation was produced with Bravo. Graphs and illustrations were prepared with Plot and Sil. Bribe and Scribe assisted with the reference list.

Remote Procedure Call

Bruce Jay Nelson



May 1981

Isla de Pascua—Chile
27° 06' S   109° 21' W
The Dutch discover huge monoliths on the eastern edge of Polynesia, Easter Day, 1722

Table of Contents

List of Figures, Tables, and Algorithms xiii

Acknowledgements xv

1 Introduction 1

1.1 An Informal Perspective 1

1.1.1 Concurrent Systems 1

1.1.2 Distributed Systems 2

1.1.3 Messages in Operating Systems 2

1.1.4 Procedures in Programming Languages 3

1.1.5 Remote Messages and Remote Procedures 4

1.1.6 Data Transfer 5

1.2 The Thesis 6

1.2.1 Scope and Goals 6

2 Remote Procedure Call 9

2.1 Definitions, Examples, Primitives, and Models 9

2.1.1 Definition of Remote Procedure Call 9

2.1.1.1 Autonomy 10

2.1.1.2 Unreliable Communication 10

2.1.1.3 Coroutines, Exceptions, and other Transfers 11

2.1.1.4 Synchrony and Concurrency 11

2.1.2 A Remote Procedure Example 12

2.1.2.1 A File Server Example 12

2.1.3 An Abstract Machine RPC Primitive 13

2.1.3.1 Remote *Transfer* 15

2.1.4 An Language Translator RPC Primitive 15

2.1.5 A Model and Example of Communication Systems 17

2.1.5.1 An Abstract Communication Model 17

2.1.5.2 The Internetwork, a Physical Communication System 18

2.1.5.3 Pup Internetwork Levels 19

2.1.5.4 Protocol Levels in Conventional Systems 21

2.1.6 A Crash and Failure Model 22

2.2 Overview of the Essential Issues 23

2.2.1 Fundamental and Nonfundamental Issues 23

2.2.2 Call Semantics 24

2.2.2.1 Exactly-once Semantics 25

2.2.2.2 Last-one Semantics 25

2.2.3 Binding and Configuration 26

2.2.3.1 Remote Interfaces 26

2.2.4 Typechecking 26

2.2.4.1 Type Translation 27

2.2.5 Parameter Functionality 27

2.2.5.1 Marshaling Parameters 27

2.2.6 Concurrency Control and Exception Handling 28

2.3 A Glance at Important Peripheral Issues 28

2.4 Benefits 29

2.4.1 Resource Sharing 29

2.4.2 Load Splitting 30

2.4.3 Conversation 31

2.4.4 My Motivation 31

3 Survey of Existing Mechanisms 33

3.1 Background 33

3.2 Mechanisms 34

3.2.1 Sail's Message Procedures 34

3.2.2 The Arpanet's Distributed Programming System 35

3.2.3 Hamlin's Cages 36

3.2.4 Rochester's Intelligent Gateway 36

3.2.5 CMU's Multi-Media Message System 37

3.2.6 Cook's StarMod 37

3.2.7 Clu's Guardians 38

3.2.7.1 Type Translation and Transmission 39

3.2.8 Spector's Remote Memory Operations 39

3.3 A Brief Evaluation 40

4 Ideal Properties of a Transparent Mechanism 41

4.1 The Essential Issues 41

4.1.1 Call Semantics 42

4.1.1.1 At-least-once Semantics 42

4.1.1.2 Last-of-many Semantics 43

4.1.1.3 Crash Semantics 45

4.1.1.4 Exterminating Orphans 47

4.1.1.5 Immediate-return Semantics 48

4.1.1.6 Sequencing Semantics 49

4.1.1.7 Invocation Schemes 50

4.1.2 Binding and Configurations 51

4.1.2.1 The Spectrum of Binding Times 52

4.1.2.2 Binding Interfaces and Components 52

4.1.2.3 Remote Variables 52

4.1.2.4 Authentication and Authorization 53

4.1.2.5 Binding Heterogeneous Configurations 54

4.1.2.6 Load Control 54

4.1.3 Typechecking 55

4.1.3.1 The Flexibility Spectrum 55
4.1.3.2 Type Authentication and Validation 56
4.1.3.3 Type Translation 57
4.1.3.4 Versions and Persistent Values 57
4.1.4 Parameter Functionality 58
4.1.4.1 VAR Parameters 58
4.1.4.2 Pointer Parameters 59
4.1.4.3 Procedure Parameters 61
4.1.5 Concurrency Control and Exception Handling 62
4.1.5.1 Concurrency 62
4.1.5.2 Exceptions 63
4.1.5.3 Aborts 63
4.1.5.4 Timeouts 63
4.2 The Pleasant Issues 64
4.2.1 Good Performance 64
4.2.2 Sound Remote Interface Design 64
4.2.3 Atomic Transactions 65
4.2.4 Respect for Autonomy 65
4.2.5 Type Translation 66
4.2.6 Remote Debugging 66
4.3 Summary of Ideal Properties 67
4.3.1 Essential Properties 67
4.3.2 Pleasant Properties 68

5 Design Approaches for a Transparent Mechanism 69
5.1 Emissary's Semantics 69
5.2 Design Overview 70
5.3 Orphan Algorithms 70
5.3.1 Algorithm Definitions 71
5.3.2 Orphan Definitions 71
5.3.3 Extermination 73
5.3.3.1 Mutually Interacting Crash Recoveries 74
5.3.3.2 Costs and Stable-storage Requirements 75
5.3.3.3 Incomplete Exterminations 79
5.3.4 Expiration 79
5.3.4.1 Costs and Clock Requirements 83
5.3.5 Epochs and Reincarnation 83
5.3.5.1 Weak Last-one Semantics 83
5.3.5.2 Regular Reincarnation 84
5.3.5.3 Gentle Reincarnation 85
5.3.5.4 Costs and Communication Requirements 86
5.3.6 A Comparison of Orphan Algorithms 86
5.3.7 Other Orphan Schemes 87
5.3.7.1 Reincarnation Only 88
5.3.7.2 Reliable Broadcasting 88
5.3.7.3 Deadlining with Postponement 88
5.3.8 Reflections 89
5.3.8.1 Transparency and the Essential Properties 90
5.4 Remote Call Mechanisms 90
5.4.1 Remote Call Machinery 90
5.4.1.1 Procedure Call Code Sequences 91
5.4.1.2 Local and Remote Transparency 92
5.4.1.3 Emissary's Runtime Mechanism 93

5.4.1.4 Some Fine Points of Emissary's Design 95
5.4.1.5 Omitted Details in Emissary's Algorithm 98
5.4.1.6 Performance Considerations in Emissary's Design 100
5.4.2 General *RTransfers* 101
5.4.3 Marshaling Parameters 103
5.4.3.1 Basic Operations 103
5.4.3.2 Marshaling Specific Types 103
5.4.3.3 Other Languages 107
5.4.4 Stubs 107
5.4.4.1 An Example 107
5.4.4.2 Stub Advantages 108
5.4.4.3 Marshaling Procedure and other Transfer Parameters 109
5.4.4.4 Stub Translator Design Issues 111
5.4.5 Reflections 112
5.4.5.1 Transparency and the Essential Properties 113
5.4.6 The Emissary RPC Algorithm 113
5.5 Distributed Binding 123
5.5.1 Background 124
5.5.2 Dynamic Loading of Configurations 124
5.5.2.1 A Local Loader Model 124
5.5.2.2 A Remote Loader Model 125
5.5.2.3 Dynamically Mustering a Set of Programs 125
5.5.3 Static Binding of Configurations 126
5.5.3.1 Using a Remote Server 128
5.5.3.2 Commanding a Distributed Company 129
5.5.4 Reflections 131
5.5.4.1 Transparency and the Essential Properties 132
5.6 Summary 133

6 Performance Evaluation of a Family of Mechanisms 135
6.1 Family History 135
6.1.1 Processors 136
6.1.2 Communication 136
6.1.3 RPC Mechanisms 136
6.1.3.1 Envoy 136
6.1.3.2 Diplomat 137
6.1.3.3 Stubs 138
6.1.3.4 Liaison and PktStream 138
6.1.3.5 EtherPkt 138
6.1.3.6 EtherPktMC 138
6.1.3.7 Profile of Characteristics 138
6.2 Benchmark Description 139
6.2.1 Benchmark Procedures 139
6.2.2 Timing Methods 140
6.2.3 PC Histograms 141
6.3 Performance Evaluation 141
6.3.1 Dolphin Remote Call Times 141
6.3.2 Dorado Remote Call Times 142
6.3.3 Network Characteristics 143
6.3.4 Process Utilization 144
6.3.5 PC Histograms 145
6.4 Performance Lessons 146
6.4.1 Bytestreams are bad. 146

6.4.2 Watch for hidden protocol costs. 147
6.4.3 Special-purpose protocols are good. 148
6.4.4 Use microcode for exceptional performance. 148
6.4.5 Caches are very important. 149
6.4.6 Marshal by compiling inline, not interpreting out-of-line. 150
6.4.7 Marshal large blocks, not small ones. 151
6.4.8 Select your data protocol carefully. 151
6.4.9 Avoid copying whenever possible. 152
6.4.10 Summary of Performance Lessons 153
6.5 Functionality Lessons 154
6.5.1 Named binding is important. 154
6.5.2 Clients want full parameter functionality. 155
6.5.3 Remote interfaces must be carefully designed. 155
6.5.4 Call-by-reference problems are tricky. 155
6.5.5 Summary of Functionality Lessons 157
6.6 Retrospective 158

7 Conclusion 159
7.1 Reviewing the Goals 159
7.2 A Critical Evaluation 160
7.2.1 The Value of Transparency 160
7.2.2 The Need for Orphan Algorithms 161
7.2.3 The Role of Performance 162
7.2.4 The Trials of Implementation 162
7.2.5 The Nature of Processes 162
7.3 Future Directions for RPC 163
7.4 Contribution to Computer Science 164

Appendix 1. Some Mesa Details 165

Appendix 2. Examples of Envoy-Diplomat, Liaison, and EtherPkt 169
A2.1 Envoy-Diplomat 170
A2.2 Liaison 178
A2.3 EtherPkt 185

References 191

Rapa—Austral Seamount
27°36′ S 144°20′ W
In this southern Polynesian enclave, women sometimes outnumber men six to one

List of Figures, Tables, and Algorithms

Figures

2.1 A simple remote procedure call from client *C* to server *S*. 12
2.2 The *Transfer* implementation of a procedure call from context *C* to procedure *P*. 14
2.3 A stub remote procedure call from client to server. 16
2.4 Formats of an Ethernet packet and of an encapsulated Pup internet datagram. 19
2.5 The abstract levels in the Pup internetwork protocol hierarchy. 20
4.1 A two-machine last-of-many remote procedure call. 43
4.2 A three-machine nontransitive last-of-many remote procedure call. 44
4.3 An orphaned remote procedure call that violates last-of-many semantics. 44
4.4 Two invocation policies: concurrent via processes and serial via call queueing. 50
5.1 A distributed system with orphans originating at recovering node *B*. 72
5.2 A distributed system with orphans originating at crashed node *G*. 87
5.3 The compiled code sequences for a remote call of *P* by *Q*. 92
5.4 An overview of an Emissary remote call from *Qnode* to *Pnode*. 94
5.5 A remote *FileOps.ReadPage* call from client to server. 107
6.1 Dolphin remote call times. 142
6.2 Dorado remote call times. 143
6.3 *TwentyArray* execution profiles. 147
6.4 Performance comparison of the evaluated mechanisms. 153

Tables

6.1 Summary of main RPC family characteristics. 139
6.2 Dolphin remote call times. 141
6.3 EtherPkt and EtherPktMC remote call times. 143
6.4 Network characteristics. 144
6.5 Process machinery utilization. 144
6.6 *TwentyArray* execution profiles. 145
6.7 *StringDescriptor* marshaling times. 150

Algorithms

4.1 Lampson's unabridged last-of-many remote procedure algorithm. 46
5.1 Emissary's orphan extermination algorithm. 76–78
5.2 Emissary's orphan expiration algorithm. 81–82
5.3 Emissary's high performance remote procedure mechanism. 114–22

Tofua—Tonga
19°46′ S   175°04′ W
Fletcher Christian and the mutineers cast Bligh adrift, 28 April 1789

Acknowledgements

Writing my dissertation was a voyage of discovery through a sea of detail. While the role of the crew was important—navigation through shoals, reassurance during storms, companionship in the doldrums—I had to be the captain throughout. The commission demanded more than I expected, and one of its rewards is the satisfaction I feel now that the voyage has ended. I call this success, although the ultimate worth of my cargo can only be valued long after it's been to market.

My main crew, a committee of four, was superb. Their strong individual contributions in the areas of call semantics, remote binding, and process failures are too numerous to mention. Bob Sproull tirelessly read through endless drafts, ever insistent on clarity and conciseness. Bob was an invaluable and uncompromising advisor who stuck to the helm in the heaviest of seas. Jim Mitchell was an extremely persistent reader, and his ample comments—from small to large—significantly increased the range, accuracy, and readability of the thesis. Jim was a patient coadvisor whose good ideas and spirits lightened my few passages through the doldrums. Anita Jones carefully considered all of my models, and her criticism and editing improved them manyfold. Rick Rashid suggested some splendid organizational changes and pinpointed many unwarranted (and now eliminated) generalizations.

In addition to my committee, Will Crowther, Craig Everhart, Paul Hilfinger, Andy Hisgen, Butler Lampson, Roy Levin, Roger Needham, and Mike Powell contributed ideas to the dissertation. Butler's continuing work on remote procedures kept me constantly on my toes: indeed, he was a effectively a silent coadvisor. Craig, Paul, and Andy read critical sections of the thesis; their insightful comments remedied several problems and clarified the presentation.

Other people contributed real work to the dissertation, actively helping me sail my ship. Jim White launched Envoy-Diplomat by christening a special version of Envoy. Amy Lansky and Paul Rovner wrote the first version of Stubs by building on top of Diplomat. Andrew Birrell wrote the

PktStream code and help me tune and debug it as component of Liaison. Gene McDaniel wrote all of the EtherPktMC microcode and, in addition, Gene's wonderful Spy gathered the histogram data appearing in chapter 6.

My voyage had two patrons. Carnegie-Mellon University's Computer Science Department and Xerox's Palo Alto Research Center. In my mind, a joint degree was awarded by both. In particular, CMU fostered the freedom that let me pursue my topic so single-mindedly, and Xerox's marvelous computing environment let me accumulate so much detail for chapters 5 and 6. These are two incomparable institutions. My highest compliment to them is that each was generous enough to let me spend half my time at the other.

The logistics of any long voyage are formidable, and I was lucky to enlist some of the best tacticians available. On the east coast, Sharon Burk's skillful ministrations allowed me the luxury of a May graduation—with nanos to spare. Sharon's love of students and mastery of CMU red tape make her a small treasure. On the west coast, Sara Dake coordinated the rapid passage of drafts between Jim Mitchell and myself and assisted Gloria Warner in dealing with Xerox's patent attorneys. Sara is also a treasure, and her role in my life is simply too large for words.

A chronicler on a voyage of discovery necessarily writes for others. I began my voyage with good writing skills, not realizing I'd spend many a dark night honing them further. My best English tutor, Cynthia Hibbard, gave excellent and learned advice on all writing matters. I also persuaded Cynth to critically review the final manuscript. My other tutors, fortunately, were able to be more constant companions: *A Handbook for Scholars* by Mary-Claire van Leunen (Knopf, 1978), *A Manual of Style* (University of Chicago Press, twelfth edition 1969), and *The Elements of Style* by William Strunk Jr. and E. B. White (Macmillan, third edition 1979).

Bon voyage.

Manihiki—Cooks
10° 24' S 161° 01' W
Natives free dive to a record thirty fathoms for mother-of-pearl

1

Introduction

Network pioneers explored many approaches to intermachine communication as they built the first distributed systems. This dissertation picks one promising approach—remote procedure call—and develops it into a flexible and powerful tool for communication in distributed systems.

1.1 An Informal Perspective

Mechanisms for communication between programs have developed steadily since Konrad Zuse's first high-level *Plankalkül* proposals in 1945 [3]. Early programmers had their hands full dealing with the complexities of exactly one process executing on exactly one processor. Today's programmer still deals with sequential processes, but he is also often concerned about the cooperative interactions of many such processes executing concurrently on one or more processors.

1.1.1 Concurrent Systems

The development of programming language mechanisms for communication and control in concurrent systems has traditionally taken two paths: procedure calling and message passing.

> *Procedures.* The procedural approach is characterized by independent single threads of control, one per process. Sharing and synchronization, sometimes accomplished with global variables and semaphores, are now more frequently (and abstractly) handled with Hoare-like monitors [10,37]. Interprocess communication occurs by procedure calls to monitors. The degree of concurrency is controlled by creating and destroying processes. This procedural approach tends to deemphasize concurrency and establishes a dominant theme of sequentiality that reduces apparent program complexity.
>
> *Messages.* The message approach is characterized by multiple threads of control. Interprocess communication is via messages explicitly sent between ports (mailboxes) that are accessed by one or more processes. In pure message-passing systems there is usually no

shared global data, and synchronization occurs automatically as messages queue at ports. Concurrency is controlled by using independent *Send* and *Receive* operations on messages so that a process is free to perform other computation—i.e., send more messages—rather than wait for a reply. The message approach emphasizes explicit concurrency.

1.1.2 Distributed Systems

In many computer systems, programs, processes, and processors are becoming increasingly more autonomous (logically isolated) and distributed (physically separated). This is causing a significant quantitative change in the problems faced by the programmers of these distributed systems. Application programs have traditionally dealt with monolithic operating systems that coordinate all services. But in network or multiprocessor environments, the same application programs often deal with a collection of autonomous, decentralized services that execute on distinct physical or virtual machines. This separation of resources has forced previously one-process applications to deal with multiple-process interactions that were once managed by a host operating system. This places an additional burden on the application programmer: Not only must his program be correct, but it must also be reliable in the presence of failures that were often previously handled by a monolithic operating system.

In a sense distributed systems have created no *new* burden at all. They have just exacerbated an old one by shifting some of the responsibility for robust coordination of critical resources like file systems from carefully armored operating systems to generally unprepared applications. Alternatively, this shift can also be attributed to increased *user expectations* about the reliability of a distributed system.

One way to achieve high reliability is to extend existing (robust) operating system facilities over the distributed environment. This approach provides applications with a uniform high-level *network operating system* [94]. While network operating systems have been built, most distributed systems emphasize the independent and decentralized nature of their environments. Lower-level and more loosely coupled communication is considered appropriate in these explicitly decentralized systems.

In practice, of course, applications will see a spectrum of resource functionality and reliability, ranging from distinct autonomous services to fully integrated conventional systems. What distributed systems have done is to widen an application's simultaneous view to include this entire spectrum of functionality and reliability—not just a narrow point or two. This widening of the spectrum, especially at the distributed end of the scale, has created a demand for new language-level primitives to deal with the communication needs of programming distributed systems.

1.1.3 Messages in Operating Systems

A frequent operating-system response to this demand has been to provide message-passing primitives. For example, Demos, the RC4000, RIG, StarOS, and Thoth are all message-oriented operating systems [2,10,51,41,15].

The message-passing approach is a natural reflection of underlying implementations that usually copy messages between processes, or transmit messages between processors in a network. Message-based interprocess communication has the advantages of simple and efficient implementation, direct introduction of parallelism, and explicit notions of *Send* and *Receive* failure. The latter is especially important in distributed environments where the partial failure of one, but not all, autonomous services is commonplace.

On the other hand, this operating system approach can have several disadvantages from a language design standpoint. The first is that messages introduce a control primitive that is quite different from procedure-oriented mechanisms. This can be a problem for procedural (Algol-like) languages where a message-passing operation is a new communication primitive. The second is that messages are sometimes introduced into languages with an inconsistently typechecked structure. This can cause type violations in strongly typed languages. Unfortunately, these two disadvantages often combine to give message passing a different, confusing level of semantic support from procedure call. (This is not to slight exclusively message-oriented environments like Xerox's Smalltalk and Hewitt's Actors [26,35], which have provided elegant message semantics in single-machine systems for some time.) Fortunately, recent language-level message-passing distributed system designs have overcome this problem (e.g., Hoare's Communicating Sequential Processes (CSP), Feldman's Plits, and Liskov's Guardians [38,23,57]).

1.1.4 Procedures in Programming Languages

In programming languages, another response to the demand for distributed-system communication primitives has been the remote procedure call, that is, one machine (or virtual machine) "calling" another. Both Brinch Hansen's Distributed Processes and Cook's original StarMod exemplify this approach [12,17].

The development of remote procedure call (RPC) is a natural outgrowth of work on the abstraction of data, control, and concurrency in procedural languages. For RPC purposes this evolution can be briefly summarized as follows: The basic procedure notions of Algol60 were supplemented with type specifications in Pascal. Brinch Hansen extended Pascal with monitors and processes in Concurrent Pascal [11]. Modules were added as an abstraction method in Mesa and Modula [62,97]. Finally, separate compilation of strongly typed modules and monitors appeared in Mesa and Ada [91].

A crucial characteristic of all these languages is that their processes are *tightly coupled* and can share memory. Because of this, interprocess communication in these languages usually occurs by local procedure calls to monitors. When monitors are located in different address spaces from the programs that want to to call them, however, local procedures no longer work and *remote procedures* (with identical semantics) can be used. This remote environment of isolated processes and monitors corresponds to distributed systems of distinct (virtual) machines.

An important property of remote procedures is that they can uniformly extend Algol-like naming and binding into a distributed system: If the naming and binding of remote monitors and their procedures conform to the same standards as their local counterparts, then programmers will see a consistent structure in distributed systems. This approach represents one logical evolution of the single-machine languages described above into a multiple-machine distributed system.

While remote procedures may seem like an obvious communication primitive for use in distributed systems, their language-level implementation has been hampered by the very strong semantics of local procedures. For example, local procedure call has well-defined semantics for parameters, concurrency, naming, and binding. In many languages procedures are also subject to the scrutiny of strong typechecking. Consistently extending even a subset of these procedure semantics into a loosely coupled distributed environment has been much harder than specifying an orthogonal message scheme with different semantics.

A more fundamental semantic problem with remote procedures has been the "always returns" property of local procedures. Guaranteeing that a remote procedure call always returns is extremely difficult in a distributed environment where both computers and communication can—and do—fail arbitrarily. Compounding these reliability problems, the lack of powerful concurrency and exception-handling mechanisms in procedural languages has made difficult the extension of local calls into remote ones without some combination of clumsy timeouts, error codes, or call-failure traps. Fortunately, languages with sufficient concurrency and exception facilities are now becoming available (e.g., Ada and Mesa).

1.1.5 Remote Messages and Remote Procedures

The preceding discussion of messages and procedures uncovered an interesting trend: operating systems usually provide message-passing interprocess communication primitives, and programming languages usually offer procedure-based primitives. The reason for this difference lies in the nature of operating systems and programming languages themselves. Operating systems are typically designed to support a variety of different programming languages and applications, and to support them all reliably. Given the proliferation of message-based interprocess communication (IPC) mechanisms in uniprocessors and multiprocessors, most operating system designers must feel that message passing is a more reliable and adaptable IPC primitive than procedure call. (Of course, many operating systems are "impure" and have *procedural* interfaces to themselves. But this discussion is about application-to-application IPC, even if it is invoked with procedures.) An operating system that adopts RPC as its application-level IPC primitive—for instance, Pilot [73]—must apparently assume that all applications will be using the same (or related) high-level languages. The ostensible conclusion here is that RPC is frequently easy to integrate into a language—where uniform high-level primitives are wanted—but often hard to integrate into an operating system—where universal low-level primitives are desired. But this conclusion is not absolute: as noted above, full support for either message- or procedure-based communication is found in at least one programming language or operating system.

This discussion has tried to show that neither message passing nor procedure calling is a panacea for the problems of communication and control in distributed systems. Whether messages, procedures, or entirely different mechanisms are the best language-level primitives for communication between independent asynchronous programs is still an open question. What is certain, however, is that the fundamental semantics of any communication primitive must cope with the failure-prone nature of distributed systems.

Guarino's thesis [29] addresses the message-passing and procedure-calling question in depth and concludes that either mechanism can be used to implement a very general model of control. In the common case of intraprocessor communication, each scheme is represented well in present programming environments—for example, CSP and Smalltalk use messages, and Mesa and Modula use procedures. Lauer and Needham [52] discuss the same question in the much more concrete setting of operating system structures and conclude that, under some mild restrictions, messages and procedures have the same power. Although Guarino's review of Lauer and Needham's paper exposes some fundamental flaws in their arguments, she does not dispute their following high-level result: The decision to provide message-based or procedure-based control primitives should be founded on considerations of machine architecture and programming environment rather than on intrinsic properties of messages or procedures themselves. Again, each scheme is present in existing single-machine operating systems (e.g., Demos uses messages and Pilot uses procedures).

The extension of Lauer and Needham's argument to distributed environments is not always readily accepted. In a 1978 distributed computing workshop [69], Howard Sturgis's suggestion that distributed systems be built with remote procedures provoked heated argument. In a later 1980 workshop [34], however, the benefit of several years of actual experience was clear: Workshop participants responded favorably to reports about a number of operational RPC-based and message-based systems. This acceptance is to be expected, for the problems uncovered in the comparison above share a common semantic thread in both schemes. Issues of call semantics, naming and binding, system configuration, strong typing, parameter functionality, concurrency control, exception handling, and error recovery are still being actively investigated for both mechanisms. Because remote procedure call and message passing are both rubric for a convenient, powerful, and semantically coherent communication primitive, we can fully expect that work on one mechanism will directly benefit work on the other.

1.1.6 Data Transfer

Of course, there are situations in which neither messages nor remote procedures are the best *communication* mechanism. These situations are characterized by extensive communication that requires little or no programmer control. For example, bulk data transfers between machines are supported quite handily by existing network bytestream mechanisms [8]. These network streams are very similar to the untyped file and stream I/O that most programming environments supply through the courtesy of their underlying operating systems. Digital voice transmission is another situation where rapid but somewhat unreliable response is more important than slow but reliable

delivery. While messages and remote procedures are poor models for this bulk communication, they can still be fruitfully used to control the initiation, mediation, and termination of these communications.

1.2 The Thesis

The thesis of this dissertation is that remote procedure call (RPC) is a satisfactory and efficient programming language primitive for constructing distributed systems. I studied remote procedures for two reasons:

> I believe in casting design problems in a concrete setting. My experience with the Alto computers, Mesa programming language, and Ethernet local network lead quite naturally to my studying remote procedures in the context of this strongly procedure-oriented distributed environment.
>
> As distributed computing comes of age, creating a suitable paradigm for programming distributed systems is important. The system designers and programmers of most conventional systems are already familiar with procedural languages—e.g., Ada, Algol, Bliss, C, Interlisp, Maclisp, Mesa, Pascal, and PL1. Giving these programmers a primitive for distributed communication that preserves the important notions of abstract datatypes can make their transition to distributed systems reasonably straightforward. RPC is one such primitive because it can have all of the desirable properties—and also the familiarity—of local procedure call.

1.2.1 Scope and Goals

This dissertation investigates remote procedure mechanisms for homogeneously programmed distributed systems using procedure-oriented programming languages. It explores the general semantic issues of RPC and presents the detailed design of *Emissary*, a transparent remote procedure mechanism. The concept of *transparency* is vital to the thesis. It is defined as follows:

> *Transparency.* Two programming language mechanisms are *transparent* if they have identical syntax and semantics. In particular, a transparent language-level RPC mechanism is one in which local procedures and remote procedures are (effectively) indistinguishable to the programmer.

The Emissary design, while unimplemented, is based on significant implementation experience and performance evaluation of prototype mechanisms. As a result, Emissary is one of the most complete language-level RPC mechanisms known at this time.

The target setting for Emissary is based on the Alto-Mesa programming environment and Ethernet local network [61,62,89]. This particular choice of a concrete environment is good for two reasons. First, Mesa is a state-of-the-art language that supports concurrency directly with its process and monitor abstractions. Second, the Ethernet is an exemplary high-bandwidth communication channel for local-area (say, one square kilometer) distributed systems. Because the Alto-Mesa environment is so general, the Emissary-specific results of the thesis are readily applicable to similar

distributed systems (for instance, to CMU's Spice system [13]). Some design decisions will certainly change for environments with different attributes, notably communication characteristics, programming philosophies, and process structures. But the basic design principles and performance lessons of this dissertation apply to most distributed systems that have loosely coupled processors programmed with procedural languages.

This dissertation focusses primarily, but not totally, on homogeneous language systems. Because real distributed systems are not homogeneous, many RPC-related issues of heterogeneous languages and environments are discussed as well. The underlying philosophy of the dissertation is to survey the entire area of remote procedures, even where demonstrating the thesis does not require detailed exploration.

To prove its thesis the dissertation addresses three major goals.

Desirability of remote procedure call. Chapters 2 and 3 show that remote procedure call is a desirable and satisfactory primitive for communication and control in distributed programs.

Transparency of remote procedure call. Chapter 2 introduces the important RPC issues and emphasizes the need for a clear statement of remote procedure semantics in the presence of machine crashes. Chapter 4 addresses these issues in detail and extracts a set of properties that must be satisfied by any RPC mechanism that has syntactic and semantic transparency. Chapter 5 presents Emissary—which has these properties—showing that remote procedures are a realistic primitive for distributed programming.

Efficiency of remote procedure call. Chapter 6 contains a performance evaluation of a large family of operational RPC mechanisms. The empirical evaluation yields a series of general performance lessons about communication in distributed systems. These lessons, which increase the performance of the prototype mechanisms by a factor of 35, are incorporated into Emissary's design.

Tarawa—Gilberts
1°25′ N 173°00′ E
With just a knife, lagoon swimmers kill tiger sharks for sport

2

Remote Procedure Call

Communication in distributed systems is characterized by transfers of information and control between autonomous programs executing in distinct (virtual) machines. This study of remote procedure call is the result of a search for a suitable *language-level* communication primitive for use in distributed systems. One approach to language-level communication is simply to make remote procedure calls between distributed programs look and behave exactly the same as local procedure calls in traditional programs. The thesis adopts this approach wholeheartedly, and the resulting quest for syntactic and semantic transparency raises a number of fundamental RPC issues. This chapter introduces these issues briefly; chapter 4 considers them in depth.

The desirability of RPC can be measured by its benefits. The end of this chapter characterizes the benefits of remote procedures in three ways and then gives a few examples of each. The examples are not comprehensive, nor are they suitable only for RPC, but they do demonstrate the widespread utility of remote procedures in distributed systems.

2.1 Definitions, Examples, Primitives, and Models

2.1.1 Definition of Remote Procedure Call

> Remote procedure call is the synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel.

The terms *remote procedure* and *remote procedure call* have been used by members of Arpa's Network Working Group to describe calling procedures through the Arpanet [32] for many years [70].

2.1.1.1 Autonomy

A key notion of "remoteness" is that the narrow channel is the primary medium used by programs to communicate. This is why the definition explicitly excludes traditional communication channels such as shared memory. Programs communicating with remote procedures are best viewed as isolated, autonomous entities whose preferred communication is along narrow and clearly defined pathways—communication shortcuts are neither allowed nor desirable. This autonomous node environment of distinct virtual memories and channeled communication is one instance of what Saltzer [75] and others [53] have called a *distributed system.*

Because remote procedure calls take place between autonomous programs, a failure in one program does not usually result in the failure of another and is usually detectable by another (e.g., when a remote call does not return). This isolation and detectability of failures is a fundamental property of distributed programs, and it distinguishes distributed programs from local programs (using local calls) where a failure usually stops everything (e.g., both caller and callee).

2.1.1.2 Unreliable Communication

The notion of inherently *unreliable* communication is essential in distributed systems where communication is often through error-prone channels. For example, one communication medium frequently used in distributed systems is what Cerf and Kahn [14] and Postel [71] call an *internetwork* (or internet), a highly but not totally reliable medium whose transmissions are subject with nonzero probabilities to losses, duplications, errors, and delays. This unreliability can distinguish remote procedure implementations from local procedure implementations. For instance, procedure call in uniprocessors and shared-memory multiprocessors is almost always supported by the hardware and is implicitly assumed to be totally reliable. In loosely coupled distributed systems, on the other hand, there is usually no hardware support for intermachine procedure calls. The software or firmware RPC implementation must cope with unreliable communication, not error-free shared memory (which the hardware makes reliable). But while overcoming unreliable communication is often an important consideration for an RPC implementation, it need not be a factor when communication *is* reliable. For example, programs conversing with RPC can be executing in autonomous virtual processors that are multiprogrammed on the same uniprocessor (or multiprocessed on a multiprocessor). If an underlying kernel uses message-passing to provide interprocessor communication, then not only is a narrow channel being used, but the channel is probably totally reliable if the kernel uses memory-to-memory message copying.

It is important to observe that the reliability of communication usually has no qualitative impact on RPC semantics. Whether the underlying remote procedure *implementation* uses a reliable or unreliable channel, the autonomous programs at the ends of the channel can still fail independently. Thus the key aspects of distributed programs remain autonomy and isolation of failure, although communication characteristics—specifically, speed and reliability—are likely to have a marked *quantitative* impact on RPC performance.

In summary, the sole requirement of remoteness is that the primary communication medium used by autonomous programs be a narrow channel. Furthermore, because the unreliability of communication is a central theme in distributed systems, the remainder of the thesis uses the unreliable internetwork as the standard channel model.

2.1.1.3 Coroutines, Exceptions, and other Control Transfers

A third important notion in the RPC definition is that the control transfer implied by RPC is not limited to procedure call itself. While procedure call is the dominant control transfer discipline in procedural languages, there are usually less frequently used language-level primitives as well. These can include coroutine transfers, exceptions, and module initialization and finalization. Thus a general RPC scheme must support arbitrary synchronous transfers such as those proposed by Lampson, Mitchell, and Satterthwaite in their paper "On the Transfer of Control between Contexts" [44]. The transfers described in the paper include all of those mentioned above, but specifically exclude GOTO; this exclusion is adopted here as well. Always remember that while remote procedure call is a popular and intuitive name, RPC is actually a misnomer that covers the spectrum of remote transfers in procedural languages.

2.1.1.4 Synchrony and Concurrency

The last essential notion in the definition is that *synchronous* transfers are required because high-level languages usually have only synchronous local communication primitives. For example, calling a procedure, transferring to a coroutine, and raising an exception normally do not start a concurrent activity that executes the caller and callee in parallel. The limitation to synchronous remote transfers does not hinder overall concurrency. A language's synchronous transfer primitives—including remote procedures, remote coroutines, and so forth—can always be composed with the language's *independent* concurrency operators—COBEGIN, FORK, and so on—into parallel activities. Of course, if a language does not have concurrency operators, then the synchronous restriction prevents concurrency—just as the original language does.

If Guarino's general communication model [29] is used to evaluate the constraint of synchronous transfers, the definition of RPC appears weak until the independent concurrency operations are included. This apparent weakness occurs because Guarino demands that her model have the power to express *any* control discipline whatsoever: the fundamental operations in her model create and destroy program components dynamically, and these components communicate both synchronously and asynchronously. Concurrency and dynamic instantiation are therefore central to her model so that asynchronous, dynamic control primitives can be defined at the lowest level. As a result, Guarino's model is wonderfully appropriate for specifying the abstract behavior of high-level language primitives, but it is not appropriate for evaluating this dissertation's definition of remote procedure call. This is because the definition of synchronous RPC is based on the high-level language behavior of synchronous local procedure call, that incorporates parallelism with independent concurrency operations.

2.1.2 A Remote Procedure Example

The prevalence of procedure call as a control primitive in most programming languages makes RPC a very intuitive concept once one understands that the remote call is simply going through a narrow channel such as a network. This is illustrated in figure 2.1 with a suggested Mesa syntax. In the example, procedure P_C in module M_C on machine C is calling procedure P_S in module M_S on server machine S. C transmits P_S's arguments to S, where S computes P_S and transmits P_S's results back to C.

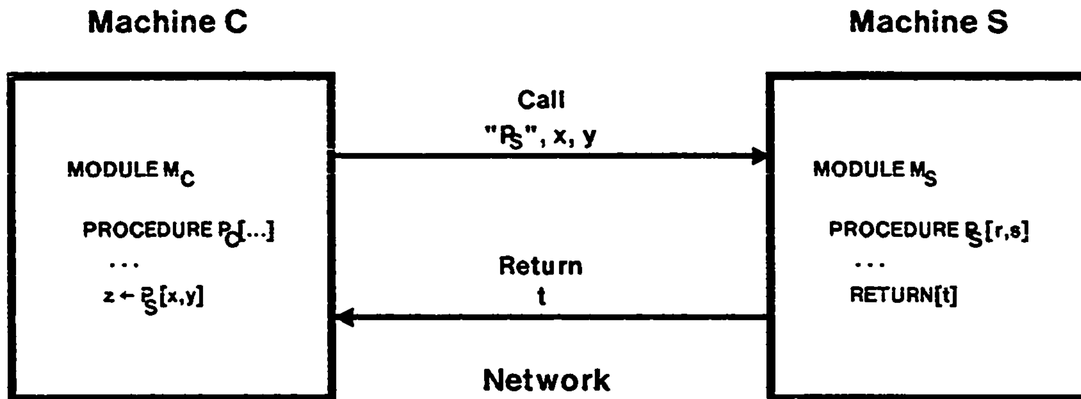


Figure 2.1: A simple remote procedure call from client C to server S.

The beauty of this example is that the programmers of modules M_C and M_S do not know there is a network between them. RPC, when done in the style proposed in this thesis, is transparent to the programmers of both the client *and* server modules. The programmer does not have to leave his usual programming environment to deal with the details of designing and implementing an application-specific protocol each time his program is distributed over more than one machine.

2.1.2.1 A File Server Example

A *file server* [5,85,87] is a node that provides information storage services to clients. (Users are often called *clients* when they can be either people or programs.) File servers usually store their information in files, and clients access data through an abstract interface of operations such as *CreateFile*, *WritePage*, *CloseFile*, and so forth. Because file servers typically execute in isolated machines that are accessed remotely through a network, their operations are ideal candidates for remote procedure call.

To illustrate this further, consider a concrete instance of figure 2.1 where C is a client reading files and S is a file server offering its operations through remote interface M_S. Client procedure P_C now calls service procedure P_S, where P_S is the server's remote *Read* procedure, e.g.,

Read: PROCEDURE [*file*: *File*, *start*, *stop*: *BytePosition*] RETURNS [*data*: *Buffer*].

Of course, if P_C processes only one byte at a time, his call of *Read*[*file*, *position*, *position*+1] will have rather large overhead. In this case it is likely that the server, S, will provide the client with

some *local* procedures that execute in *C* and access the server as necessary. For example, the following local *ReadByte* routine could maintain a buffer in *C* and call *Read* remotely in *S* as necessary to refresh it:

ReadByte: PROCEDURE [*file*: *File*, *position*: *BytePosition*] RETURNS [*byte*: BYTE].

Observe that the client application on *C* need not know where *Read* and *ReadByte* reside. The implementors of the file server declare these operations to be local or remote (relative to the client) depending on their own policies for providing reasonable service to the client. Implementors can change these policies at any time without programming impact on the client because the client's abstract file *interface* is the same whether all, some, or none of the interface procedures are remote.

2.1.3 An Abstract Machine RPC Primitive

The Transfer paper's description [44] of local program control is important because it defines a low-level *Transfer* primitive, where *Transfer* is an abstract machine operation that is general enough to express a rich collection of synchronous control disciplines. At the programming language level—as the paper discusses at length—this collection usually includes at least procedure calls, coroutine transfers, and exception handlers. In searching for a *remote* transfer primitive, then, the desire for language-level uniformity requires looking beyond the intuitive but slightly misnamed notion of remote *procedure* call for an abstract machine operation at least as powerful as *Transfer*. Fortunately, *Transfer* is flexible enough to extend naturally into the distributed domain.

Define a *context* to be a program with some local storage (including PC) and a binding rule mapping the program's names into storage addresses. Let a *port* be the name of a context. Then as defined in the Transfer paper, the *Transfer* operation

Transfer (*destinationInport*, *returnOutport*, *argumentPtr*)

specifies that execution of the current context is suspended and that control, *returnOutport*, and *argumentPtr* are delivered to the context named by *destinationInport*. This new context begins executing at its previously suspended PC and retrieves arguments using *argumentPtr*. When the new context finishes its work, it (usually) initiates another *Transfer* of control to the *returnOutport* context with a new *argumentPtr*.

As an example of how *Transfer* is used to implement language-level primitives, consider procedure call. The context of a procedure is usually called its *activation record*. In Algol-like languages, a new activation record is created for each execution of a procedure and is destroyed when the procedure returns. The *Transfer* procedure call implementation creates and destroys these activation records with special, predefined contexts. Performing a procedure call involves *Transferring* to these special contexts as intermediate steps of the call. Figure 2.2 demonstrates this for a call from context *C* to procedure *P*. The *Transfers*, abbreviated *Xfer1–4*, are explained below.

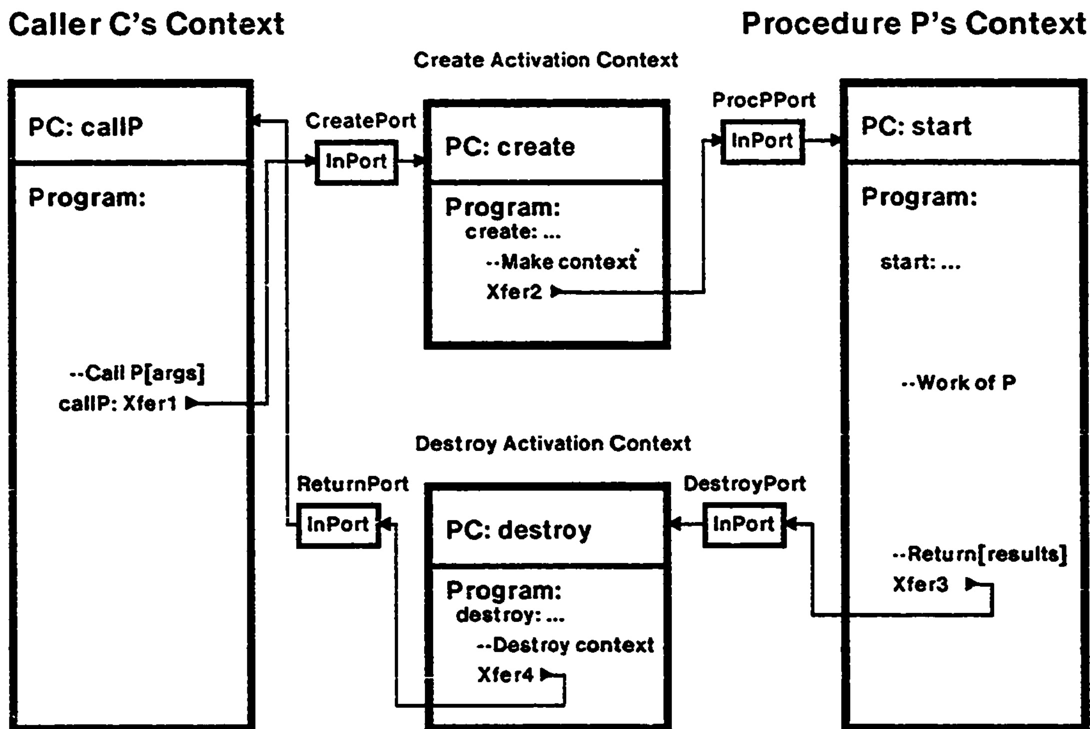


Figure 2.2: The *Transfer* implementation of a procedure call from context *C* to procedure *P*.

The call of *P* from context *C* uses four *Transfers*.

Xfer1: *Transfer*(*CreatePort*, *ReturnPort*, (*ProcPPort*,*args*)). *C* starts the call by *Transferring* to the port of the create-activation-record context, *CreatePort*. The creator uses the *ProcPPort* argument to construct the correct activation record.

Xfer2: *Transfer*(*ProcPPort*, *ReturnPort*, *args*). The creator *Transfers* to *P*'s context, passing along the *ReturnPort* and *args*. *P* then executes the procedure body.

Xfer3: *Transfer*(*DestroyPort*, *ProcPPort*, (*ReturnPort*,*results*)). *P* begins returning to *C* by *Transferring* to the *DestroyPort* with the eventual *ReturnPort* and *results*. The destroyer deallocates *P*'s activation record, which it identifies by the second *ProcPPort* argument.

Xfer4: *Transfer*(*ReturnPort*, NIL, *results*). Finally, the destroyer *Transfers* to *C*'s *ReturnPort* with *results*, completing *C*'s call to *P*. Since the destroyer is never resumed, the return link is NIL.

Executing a procedure call with *Transfer* appears like a lot of work. In practice, extensive optimizations streamline the four *Transfers*. For example, creating and destroying an activation record can be done in just a few instructions: languages that use a stack need only adjust a stack pointer; languages that use a heap (like Mesa) can use a microcoded frame allocator to obtain similar efficiency. Hence, while creating and destroying activation records complicates the model in figure 2.2, the *Transfer* implementation is nonetheless small and cheap for procedure calls.

Optimizing the common cases of the Transfer operation—and procedure call is certainly the most common—is crucial for extracting the best performance from a general mechanism like *Transfer*. For less common cases, the efficiency of a straightforward *Transfer* implementation can be quite adequate. The utility of *Transfer* lies in its versatility as a powerful descriptive tool: it can be used to model procedure call, coroutine transfer, and other language primitives. Optimization is necessary, but secondary. This vital theme is revisited throughout the thesis, especially in chapters 5 and 6.

2.1.3.1 Remote Transfer

Extending *Transfer* for remote use is straightforward. *RTransfer* is defined by making these simple changes to *Transfer*'s arguments.

> *Inport* and *outport* must now identify contexts in the distributed environment rather than just within one node. This identification is extended by prefixing the name of each existing context with the name of its host node. This presumes that contexts do not migrate, which is a continuing assumption here. (If contexts do migrate, adding a level of indirection to port names remedies the problem.)
>
> *ArgumentPtr* receives the same treatment. Because it is a pointer, however, dereferencing it in the destination address space does not work. This problem is overcome by insisting that remote *Transfer* handle *argumentPtr* by passing *argumentPtr*↑ as a value parameter, therefore copying it. This copy makes remote *Transfer* use the actual argument record, not a pointer to it. The Transfer paper suggests using this technique on short argument records; remote *Transfer* must always use it.

This definition of *RTransfer* is readily implemented with messages. First, denote the node and context parts of a *port* by *port.node* and *port.context* and rename *argumentPtr*↑ to *argumentRec* for clarity. Then

RTransfer (destinationInport, returnOutport, argumentRec)

sends the message (*destinationInport, returnOutport, argumentRec*) to *destinationInport.node*, where *destinationInport.node* delivers control to *destinationInport.context* with *argumentRec*. The *returnOutport* always remains a fully qualified name because it can specify a return to any node at all, not necessarily to the *RTransfer's* origin.

Observe that the four *Transfers* implementing local procedure calls above still serve—with *RTransfer*—to implement remote calls as well. *Xfer1* and *Xfer4* are now *RTransfers* between machines; *Xfer2* and *Xfer3* are still local *Transfers* because they switch control between contexts that are in the same machine.

2.1.4 A Language Translator RPC Primitive

The *RTransfer* primitive is a concise and appealing abstract operation for implementing remote transfers. The Mesa programming system actually uses the local version—*Transfer*—to implement control transfers between programs, procedures, coroutines, and exceptions. But there are other

languages where *Transfer* is not available as an abstract machine operation, and indeed where the compiler generates a different sequence of instructions for each control transfer—typically, for only procedure call. Integrating an RPC primitive into these *Transferless* languages is not as easy as it is for languages that use *Transfer*. Consider two possibilities:

> *Compiled RTransfers.* Implement *RTransfer* directly with compiled remote transfer instruction sequences. This has the advantage of efficiency and transparency, but may not be feasible if the compiler cannot be changed.
>
> *Source-level stubs.* A separate source-level RPC *translator* can augment the compiler by generating *stub* routines that implement *RTransfer* in the language itself. This solution can be inconvenient for application programmers, but it requires no compiler changes.

In the stub approach, the translator generates two source-level stubs for each remote procedure—one for the client and one for the server. This is illustrated in the setting of the previous file server example in figure 2.3. In the figure, the client's remote call to *FileOps.ReadPage* is automatically intercepted by the local *FileOps* stub interface, which has a declaration of *ReadPage* identical to the file server's real *ReadPage*.

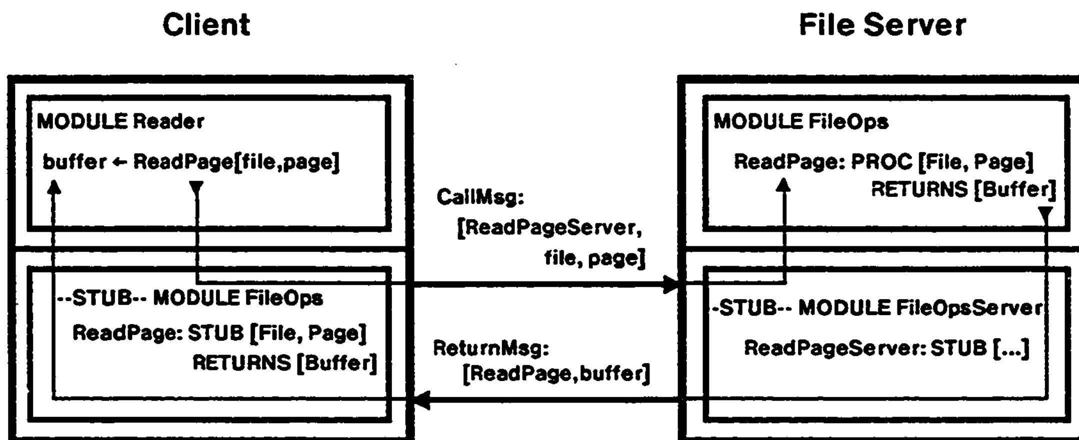


Figure 2.3: A stub remote procedure call from client to server.

The stub routines operate as follows: The client *ReadPage* stub packages the call into a *CallMsg* and sends it to the server's *ReadPageServer* stub. *ReadPageServer* unpackages the message and calls the server's actual implementation of *ReadPage*, which does not know whether the call is local or remote. When the real *ReadPage* procedure finally returns its *Buffer*, the server stub packages the buffer in a *ReturnMsg* and sends it back to the client. After the client stub receives *ReturnMsg*, it unpackages the message and returns the buffer to the original invocation of *ReadPage*. Because the call semantics are uniform, the client then proceeds as though the call had been local all along.

Notice a crucial property: The client and server programs—i.e., client module *Reader* and server module *FileOps*—do not change for remote use. The code in these modules is the same whether it is invoked locally or remotely because the physical location of the procedure implementations is hidden from programmers behind the *FileOps* interface. All the RPC details are in the stubs, which the stub translator creates automatically from a scan of the procedure declarations.

The stub approach to language-level RPC is typically less efficient than the *RTransfer* or compiler approaches because it is not as well integrated into the language runtime environment. But, in trading the efficiency of the latter methods for the translator's stub approach, procedure-calling uniformity is still preserved: users write the same code for local and remote use, and no program changes are necessary. Stubs are discussed again in chapter 5.

2.1.5 A Model and Example of Communication Systems

The descriptions of both of the previous RPC primitives were deliberately vague about "sending and receiving messages." Indeed, the reader may have wondered how *call* and *return* messages are meaningfully exchanged with the unreliable communication media of section 2.1.1.2: What if the *call* and *return* messages are lost; what if they are delivered many times; what if they have errors? In a distributed system, understanding the reasons for these questions is as important as knowing the answers because unreliable behavior is characteristic of intermachine communication primitives. While any degree of robustness can be built on top of unreliable primitives, the cost of robust operation can be extremely high. Achieving a cost-effective RPC design requires consideration of all communication characteristics.

This section uses a two-step approach to study communication characteristics. It first presents an abstract communication model. The purpose of the model is to capture the unreliable behavior of physical communication media, where messages can be lost, duplicated, delayed, and delivered with errors. Second, it presents a typical internetwork. Internets are an excellent physical realization of the communication model and are an appropriate example because they are so frequently used in real distributed systems.

2.1.5.1 An Abstract Communication Model

The communication model consists of two functions, *Send* and *Receive*, that operate on a set of messages, M, called the *medium* (M represents messages in transit). Each message, m, is self-contained. If necessary, the identities of the sources that call *Send* and of the destinations that call *Receive* are encoded in messages themselves (this is usually desired). Restating this formally,

> *Message*: TYPE = ...;
> *Send*: PROCEDURE [*m*: *Message*];
> *Receive*: PROCEDURE RETURNS [*status*: {*good*, *bad*}, *m*: *Message*];
> *M*: SET OF *Message*.

The formal semantics of *Send* and *Receive* are stated below. The notation is conventional [36]: If P and Q are logical expressions and S is a statement, then $\{P\}\ S\ \{Q\}$ means that if P is true, then after executing S, Q is true. In a concurrent program, the interpretation is that P and Q are true immediately before and after S, respectively, treating S as indivisible.

$$\{M=M'\} \quad Send[m] \quad \{ M=M' \vee M=M' \cup \{m\} \};$$

$$\{M=M'\} \quad [status, m] \leftarrow Receive[] \quad \{ status=good \wedge (M=M' \vee M=M'-\{m\}) \wedge m \in M' \\ \vee\ status=bad \wedge (M=M' \vee \exists m_{bad}(M=M'-\{m_{bad}\})) \}.$$

Send always terminates (strongly satisfies the semantics) while *Receive* might not (weakly satisfies the semantics).

This particular model is for *unreliable* communication. It has the following properties. When *Send* completes, it has not necessarily added m to M (messages can be lost). When *Receive* completes, m can be any message in M (M's messages are unordered). Furthermore, m is not necessarily removed from M (duplicates can exist). If the *status* returned by *Receive* is *bad*, then m is corrupted and should be ignored. If *status* is *good*, then m is identical to some message that was sent (no corruption).

Observe that corrupted messages in this model always have *bad* status. In physical systems this can never be absolutely true, although the probability of errors can be made arbitrarily small. Catastrophes caused by the remaining nonzero probability are ignored by the model.

2.1.5.2 The Internetwork, a Physical Communication System

At the lowest level of physical communication, an internetwork [14] has all of the properties of the preceding abstract model. To guarantee more attractive properties, however, an internetwork usually includes extra information in messages so that sources, destinations, and errors are readily identified. This section discusses these basic internet notions and also some useful higher-level abstractions. The descriptions presented here are fairly comprehensive because internets are used extensively later in the thesis.

A *packet* is the fundamental message exchanged between computers on a network. Packets are usually several hundred to several thousand bytes long; they have identifying *source* and *destination* addresses and usually contain error control information for status checking. (The format of an Ethernet packet [61] is illustrated in figure 2.4.) A *network* is a *packet transport mechanism* that connects together a number of physically distinct computers (also called hosts, nodes, and machines). The hosts of a network use *Send* and *Receive* to store and forward packets among themselves as the packets are moved from source to destination: this is where packet switching gets its name. An *internetwork* is a heterogeneous collection of networks connected by *gateways*, which are hosts with two or more networks connected to them. Gateways contain *routing algorithms* that *route* packets between networks as necessary.

A *datagram* is the fundamental message in an internetwork. It is a universal packet with internet-wide source and destination addresses that allow the datagram to be sent between any two nodes in the internet. To send a datagram between two nodes, it is first *encapsulated* in one or more network-specific packets that are transmitted to the proper destination (more than one packet is required when a datagram is larger than the maximum packet size). If a datagram must traverse two or more networks to get to its destination, then the gateways along the path will *deencapsulate* the datagram as it arrives from one network, and *reencapuslate* it before routing it out on the next. (The format of a Pup datagram [8] and its encapsulation in a single Ethernet packet are shown in figure 2.4.)

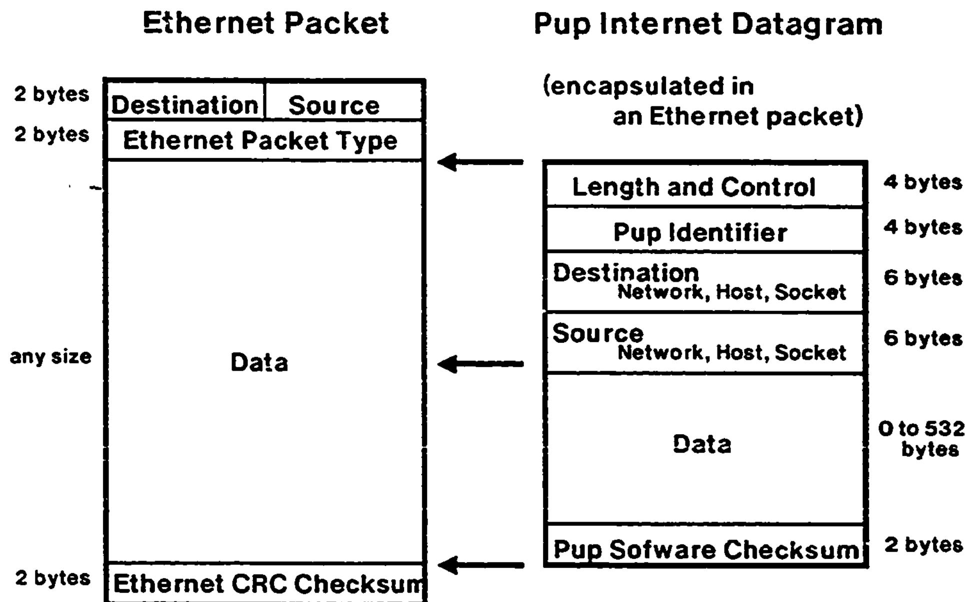


Figure 2.4: The formats of an Ethernet packet and of an encapsulated Pup internet datagram.

In this dissertation, the transmission of datagrams—sometimes simply called packets—is assumed to be *unreliable:* they are delivered only with high probability, they can contain bit errors because of corruption during transit, they can be duplicated (delivered many times), and they can be delayed (delivered long after they are sent). Cooperating hosts that want to communicate reliably must use *reliable transmission* to achieve properly sequenced, duplicate-free delivery of errorless datagrams. Reliable transmission of datagrams is performed by application-level processes that use end-to-end state information—such as sequence numbers and checksums—to keep track of their transmitted datagrams. This type of reliable two-party *connection* is often called a *virtual circuit.*

2.1.5.3 Pup Internetwork Levels

Packet transport mechanisms, internet datagrams, and virtual circuits are just three levels in a layered hierarchy of protocols. One simple system is the Pup protocol hierarchy, which is a part of the Pup internetwork architecture described by Boggs et al. [8]. The Pup protocol levels, illustrated in figure 2.5, are similar to the Arpanet TCP internet design and to the Open Systems Architecture [71,98]. The level numbers, however, are different from one architecture to the next.

Each level in the Pup protocol hierarchy implements a different abstraction and has its own characteristic properties and performance. It is worthwhile to review the layers carefully. This will be especially useful for chapters 5 and 6.

Figure 2.5: The abstract levels in the Pup internetwork protocol hierarchy.

Level 0: Network driver interface. The level 0 abstraction consists of packet transport mechanisms such as the Ethernet, the Arpanet, and packet radio. This level is used to encapsulate unreliable datagrams as they transit the internet. It is also used to send private, noninternet traffic on the local network. The software interface to level 0 is usually called the *network driver interface* because it drives individual (unreliable) packets on the local network.

Level 1: Socket or datagram interface. The level 1 abstraction is the unreliable internet datagram. Level 1 uses fixed datagram formatting, hierarchical addressing, and internet routing to provide a common layer to *all* internet hosts. An internet address is the triplet (*network, host, socket*). The *socket* is used for more specific addressing than just a host; it usually identifies a particular process, or even a mailbox within a process, as the destination of an internet packet. Because level 1 provides this end-to-end socket routing for datagrams, its software interface is often called the *socket interface* or *datagram interface.*

Level 2: Interprocess communication or stream interface. The level 2 abstraction is interprocess communication (IPC): protocols for exchanging structureless data between processes. These protocols have different combinations of reliability and throughput. They can be divided into two rough classes by the amount and duration of state information kept by the end processes: *Connectionless* protocols are used to exchange a small and perhaps even fixed number of unreliable packets that require little or no state—e.g., requesting the date from a time server. *Connection-based* protocols are used to exchange an indefinite number of packets that require significant state information to coordinate properly—e.g., reliably transferring a transparent stream of data between two processes.

In figure 2.5, RTP is a connectionless rendezvous and termination protocol that is used to establish connections, i.e., to create unique *socket* numbers that unambiguously identify the two parties who wish to communicate. BSP is a bytestream protocol that blurs packet boundaries by creating a totally reliable stream between processes (not shown is a similar SPP—sequenced packet protocol—that gives a reliable stream of packets without blurring the boundaries). RPC is a remote procedure protocol that maps reliable *call* and *return* packets between interacting processes. RPC uses more state than connectionless RTP, but less than sophisticated BSP. (RPC is shown here at level 2; it can higher in the hierarchy—or, as we will see later, lower as well.) Because level 2 has traditionally been used for bytestream traffic, its software interface is usually called the *stream interface* or *virtual circuit interface.*

Level 3 and above: Application interfaces. The level 3 abstraction adds structure and assigns meaning to the data of level 2. For instance, FTP is a file transfer program that uses RTP to create connections and BSP to move data over them reliably. The Printing and File services use their own application-oriented protocols to implement their printer and file abstractions. At levels 4 and above, application interfaces build on the lower levels and the hierarchy often becomes indistinct. The high-level protocols used at these levels can be quite complex [82]. Software interfaces at level 3 and above and above are usually application- and function-specific.

2.1.5.4 Protocol Levels in Conventional Systems

The hierarchy of protocol layers described above has an excellent analog in uniprocessor and multiprocessor systems. Level 0 represents the movement of raw data over bit-serial and bit-parallel processor and memory busses. Level 1 represents the primitive structuring on raw data that allows messages—datagrams—to be passed between processes. These level 1 messages also resemble unreliable datagrams because messages are not *always* delivered reliably in systems with buffer allocation quotas; messages usually are, however, delivered error free. Level 2 in a uniprocessor or multiprocessor message system represents traditional IPC: connection and connectionless protocols, reliable and unreliable behavior, and streams and other process coordination mechanisms. Levels 3 and above are the same in both conventional and distributed systems.

It is interesting that at level 2, RPC is usually absent from operating system protocol hierarchies. The reason for this absence, as discussed in section 1.1.5, is that programming languages and operating systems usually have different founding philosophies.

2.1.6 A Crash and Failure Model

The words *crash* and *failure* are used interchangeably in this dissertation. Their meaning is complicated in distributed systems by the presence of multiple machines and the unreliable communication between them—links of networks and gateways. After the preceding review of unreliable internet communication and its analog in conventional systems, it is useful to look at a simple model for crashes and failures in distributed systems as well.

Node crashes can be classified in two ways.

> *Machine failure* is the complete halt of a machine and all of its processes because of a hardware failure or because of a software error that affects all processes (e.g., operating system "crash"). Furthermore, in some environments the degraded response of a crippled machine can be tantamount to machine failure.
>
> *Process failure* is the halt of one process because of a process-specific hardware problem (e.g., parity error) or because of a process-specific software error.

These definitions of machine and process crash—or failure—apply to both conventional and distributed systems. Serious communication failures, on the other hand, are usually a problem only in distributed systems. For instance, if procedure-calling primitives break on a single machine it is hard to imagine that the processor has not broken, crashing all processes. But in a distributed system a broken link does not imply a broken processor, nor does the isolation of a process imply that the process has failed.

Before taking a deeper look into this, three definitions of communication-related failure are needed.

> *Communication loss* is the destruction of information because of unreliable transmission. The duration of a communication loss can span many orders of magnitude, as the next two definitions discuss.
>
> *Communication outage* is the loss of a small amount of information—say, a few packets—because of network or gateway errors. Communication outages are transient errors analogous to correctable memory parity errors and are from milliseconds to minutes in length. If the end-to-end communication mechanism is providing reliable transmission as described in section 2.1.5.2, then short outages (milliseconds to seconds) are usually masked from clients and no information is lost. As outages increase in length or frequency, however, they become more serious.
>
> *Communication breakdown* or *network partitioning* is when no information at all can be routed to the destination because of network breaks or gateway failures. Communication breakdowns are hard failures analogous to hardware and software crashes and are from seconds to weeks in length. A reliable transport mechanism generally will not mask breakdowns from clients and will inform them when the duration of a partitioning is minutes or longer. Deciding when an outage should be declared a breakdown is usually system or application dependent. The choice is roughly analogous to the machine failure decision that is made when a crippled processor has substandard performance.

Handling communication breakdown is conceptually straightforward: the isolated nodes simply wait until communication is reestablished and then retransmit their waiting information. There are two difficulties with this ideal approach. The first is timing considerations. Real programs are often not willing to wait even minutes, let alone hours or days, to complete their tasks. Rather than wait for a particular remote service these programs will often try to take their business elsewhere. This is, after all, touted as an advantage of decentralized computing. The second difficulty is the nature of the failure. When communication fails it is often impossible to tell whether the failure is occurring because of a partitioning or because of a remote machine or process crash. If the remote machine has crashed then waiting will not usually aid recovery. Because of this, most distributed programs elect to treat communication breakdown exactly like machine failure.

2.2 Overview of the Essential Issues

Behind the conceptual simplicity of the RPC examples in the previous section are a number of thorny issues. The alert reader may have had an inkling of these issues when considering the ramifications of the extended *RTransfer* definition. For instance, increasing the naming scope of *inports* and *outports* raises two questions: What happens when *RTransfer* is unable to deliver control to an isolated or unresponsive remote context? How are *RTransfer*'s previously local context names bound in the new global name space? In addition, the addressing problem with *argumentPtr* is recursive—what happens when *argumentRec* contains pointers?

Behind these questions there are five *essential issues*. They are defined very briefly here and are discussed in more detail below.

> *Call semantics* define the abstract invocation behavior of a remote procedure mechanism.
>
> *Binding and configuration* establish the naming and configuration (interconnection) of programs communicating with RPC.
>
> *Typechecking* enforces the type compatibility of interprogram bindings.
>
> *Parameter functionality* determines the restrictions, if any, on the parameters passed by an RPC mechanism.
>
> *Concurrency control and exception handling* define the interactions between the RPC mechanism and any independent parallel-processing and exception-handling mechanisms.

2.2.1 Fundamental and Nonfundamental Issues

The five issues listed above must be addressed by remote procedure mechanisms whose goal is language-level local and remote transparency in a homogeneously programmed distributed system. If the requirement of language-level transparency is relaxed, the issues can be split into two groups:

> *Fundamental invocation issues.* Call semantics and concurrency control are fundamental to any remote invocation mechanism, whether it is based on procedures, messages, or some other communication discipline. These two issues are crucial because they determine the low-level semantic behavior of an invocation primitive.

> *Language-level transparency issues.* Binding and configuration, typechecking, and parameter functionality are vital for the *language-level* transparency of local and remote procedures. In this dissertation, these transparency issues merit the same attention as the fundamental invocation issues. This is because the goal of overall local and remote transparency requires defining more than the low-level semantic behavior addressed by the fundamental issues. To obtain transparent high-level semantic behavior, the language-level issues must be considered along with the more fundamental ones. Thus there are five essential issues in all.

The importance of the division into fundamental and nonfundamental issues is quite apparent in the definitions of the *Transfer* and *RTransfer* primitives. In particular, the spirit of the *Transfer* primitive is to be as fundamental (primitive) as possible and still get the job done. Thus *Transfer* specifies only the basic semantics of intercontext control transfer, parameter passing, and concurrency (none). Nonfundamental issues of port binding and typing, parameter typing, and parameter restrictions are explicitly left to the encompassing programming language. As Lampson, Mitchell, and Satterthwaite point out in the Transfer paper [44], however, the deferral of these issues must stop at the language level because all the issues must be resolved in the language's specification. This same principle is true in languages using the *RTransfer* primitive.

The thesis adopts this principle in the following form: the language-level issues of remote procedures are resolved in precisely the same fashion as the language-level issues of local procedures. Thus, in figure 2.1, the example remote call is intentionally written to *appear* like a local call even though the two implementations are worlds—machines—apart. Again, making these local and remote calls *transparent* to the programmer is a fundamental requirement of this thesis' RPC mechanisms.

Chapter three's study of existing RPC mechanisms will confirm this requirement, for one of the strongest lessons to emerge from existing schemes is that, to be successful, they must be fully integrated into their host language environments. For example, if clients must use different mechanisms for local and remote communication, then many locality decisions will be made early in the design rather than when the resulting system is tested and tuned. In addition, statically changing the location of an application's modules (before execution) will require client reprogramming unnecessary in the presence of a transparent mechanism.

The five essential issues are now discussed in more detail. Chapter 4 considers them completely.

2.2.2 Call Semantics

Call semantics are the most critical behavioral issue of remote procedure call. There are two cases to consider: RPC in the absence of crashes (the normal situation) and RPC in the presence of crashes (a rare situation caused by a local or remote process failure or a communication breakdown). Meeting the goal of local and remote transparency requires that local calls and remote calls have the same semantics: otherwise, programmers have to adapt their code to both. This goal is clarified by separately considering normal call semantics (no crashes) and abnormal call semantics (crashes).

2.2.2.1 Exactly-once Semantics

In the absence of crashes, meeting the transparency goal is straightforward. Local procedure call is characterized by *exactly-once* semantics: the caller transfers arguments and control to the callee, waits for the procedure computation to occur exactly once, and resumes execution when the callee returns results. Remote procedure call easily achieves these exactly-once semantics by demanding that *RTransfer* send and receive its control-passing messages reliably. Reliable transmission ensures that each *RTransfer* message is exchanged exactly once, as desired.

2.2.2.2 Last-one Semantics

In the presence of crashes, obtaining transparency is complicated. First of all, local calls no longer have exactly-once semantics. If procedure *P* performs a local call to procedure *Q*, their host machine can crash any time during the transfer to *Q*, during the execution of *Q*, or during the transfer back to *P*. A crash at any of these times violates exactly-once semantics because the call does not complete. If the machine is booted and its crashed programs are restarted, either from checkpoints or from scratch, then the call from *P* to *Q* will be repeated. In the face of crashes, this repetition will continue until the call finally completes and the program either terminates or reaches another checkpoint. In the presence of crashes, then, the call results and side effects that the program actually uses are those of the very *last call* that executes. The results of intermediate executions—partial or total—are abandoned, although the side effects of intermediate calls can potentially influence the last one. We will say that local calls have *last-one* semantics in the presence of machine crashes.

Crashes in distributed systems create new semantic difficulties because a remote procedure call takes place between two independent parties, and a crash of one party usually leaves the other one running. In addition, this qualitative semantic difference is often quantitatively magnified by failure-prone networks that make loosely coupled remote calls more crash-prone than tightly coupled local calls. For example, a remote call can fail when the machine executing the callee crashes (leaving the caller running), when the machine executing the caller crashes (leaving the callee running), or when the transport mechanism partitions sometime after the *call* message is sent but before the *return* is received. The caller and callee of a local call almost never have this problem when their *machine* crashes because both caller and callee are swept away together. It can happen, however, during a local interprocess call when one of the two *processes* fails, and this is especially true in multiprocessors. This type of crash, when one of the communicating parties remains running, is considered further in chapter 4.

It is important to observe that the problems of failure, violated exactly-once semantics, and crash recovery are *not* unique to distributed systems. They are just exacerbated in autonomous, loosely coupled distributed environments.

2.2.3 Binding and Configuration

A language-level RPC scheme must give programmers convenient mechanisms for declaring distributed programs, or for specifying remote procedure bindings and module configurations. In particular, the assignment of program modules to the nodes in a distributed environment should *not* require programming. There must be a higher-order scheme, such as Mesa's configuration language (for single machines) [52], which handles the details of module assignment and intermodule procedure binding, linking, and loading. Reconfiguration of the modules and machines of a distributed program should be easily specified in this configuration language and require no lower-level changes to the programs themselves.

One important function of a binder—a program that performs bindings—is to act as a typechecking agent. Again considering the example of figure 2.1, the binder must ensure that the procedure P_S actually being called on machine S is the one that procedure P_C on machine C really wants. The binder's responsibility now extends between machines, and the programmer is asking the (remote) binder to typecheck these *procedure types* as well as link them together between whatever machines are hosting them.

2.2.3.1 Remote Interfaces

This notion of procedure types can be extended a step further by requiring, following chapter 7 of the Mesa Manual [62], that the complete *interface* of procedures *exported* by module M_S be typechecked and correctly bound to every module (such as M_C) that *imports* the interface of module M_S. This must happen for all machines that import or export M_S's interface. Mesa's notion of an interface as a collection of related procedures and definitions incorporates remote procedures nicely. Rather than deal with individual remote procedures, it is often much more effective to talk about importing and exporting entire *remote interfaces* of remote procedures (or coroutines or exceptions). The notion of binding a remote interface is crucial, and remote interfaces are used from here on.

2.2.4 Typechecking

For remote procedures to be as typesafe as local procedures, the underlying programming environment must guarantee that whatever type calculus is enforced on local machines is extended completely into the distributed environment as well. In figure 2.1, this means that the types of x and y must conform to those of r and s, and similarly for t and z.

For weakly typed languages this guarantee will often simplify into (for example) a requirement that t and z are both integers. This simplification may even extend to type translation between heterogeneous environments of weakly typed languages like Bcpl and Lisp, where Bcpl integers may inherently conform to Lisp integers. For strongly typed languages, on the other hand, the guarantee may be much more strict. One can imagine, for example, languages where if t is a RED INTEGER then z must be a RED INTEGER also.

Whatever typechecking is necessary, efficiency or convenience may dictate that much of it be done as early as possible (e.g., at compile time). But the only actual requirement is that remote calls violating the type calculus be reported as errors before they execute.

2.2.4.1 Type Translation

Type translation is a fairly separate issue of typechecking, for example, deciding when to convert a 32-bit ones-complement integer into a 36-bit twos-complement integer. This is a problem for both heterogeneous language environments *and* heterogeneous processor environments. For instance, in the Bcpl/Lisp example, both programs may run on identical machines, yet Bcpl may use 16-bit integers and Lisp 32. On the other hand, there could be a distributed Pascal program whose host environments support 36-bit integers on one machine and 32-bit integers on another. These translation issues transcend remote procedure call although they are vital to its widespread success.

2.2.5 Parameter Functionality

Permitting the argument and result parameters of remote procedures to have a full range of types (i.e., be as general as possible) is as important as typechecking. For languages with only simple scalar and static array types this is easy. For variable length structures like strings, dynamic arrays, and variant records the implementation is still straightforward, typically requiring extra dynamic allocation—in the callee for arguments, and in the caller for results. The real problems occur when parameters have implicit or explicit pointers: because the caller and callee have distinct address spaces, pointers valid in one machine are not usually meaningful in the other. These problems, including those of procedure parameters, are addressed further in chapter 4.

2.2.5.1 Marshaling Parameters

The process of packaging a parameter record into *a call* or *return* message is called *marshaling*. Similarly, the inverse process of unpackaging the message is called *unmarshaling*. The neutral term *marshal* has been chosen because it connotes a loose but ordered group. The density connotation often associated with *pack* is too strong, as chapters 5 and 6 will show.

As noted above, marshaling is complicated by parameters with pointers because a pointer's referent, not the pointer itself, is usually what must be transmitted. A parameter record containing pointers is therefore a tree with the pointers as branches and the referents as leaves. The marshaling procedure must traverse this tree and flatten it, placing all of the leaves into the parameter portion of *call* messages (for arguments) or *return* messages (for results). Unmarshaling performs the inverse operation, reconstructing the tree from the flattened representation.

As an example of marshaling, consider the argument record of the following *OpenFile* procedure.

OpenFile: PROCEDURE [*name*: STRING, *access*: {*read*, *write*, *update*}]

In Mesa, strings are implemented with pointers:

```
STRING: TYPE = POINTER TO RECORD [
    length, maxlength: CARDINAL,
    text: PACKED ARRAY OF CHARACTER ].
```

To marshal *OpenFile's* argument record, then, simply sending (*name, access*) in a *call* message is not sufficient. Instead, ((*name.length, name.text*), *access*) must be transmitted. The order and manner in which parameters are flattened and marshaled define a *data protocol* for the transmission of parameters. Marshaling and data protocols are considered again in chapters 5 and 6.

2.2.6 Concurrency Control and Exception Handling

The goal of transparency demands that remote calls execute synchronously, just as local calls do. Because a remote call (usually) involves a process switch, this implies that the caller automatically blocks and waits until the remote callee returns—just as local calls do without the process switch. For parallel execution of any activities—local or remote—the programmer must therefore use whatever concurrency tools his language provides and not expect the RPC implementation to offer hidden concurrency.

The desire for transparency also requires that remote procedure calls report errors in the same fashion as local calls. This is straightforward as long as we remember that remote calls raise some exceptions that local calls never do, for instance, *NetworkPartitioned*, *CommunicationTimeout*, and *RemoteCrash*. The handling of these exceptions will of course depend on the particular call, but their *reporting* must use the same mechanisms available to local calls—not any new ones.

2.3 A Glance at Important Peripheral Issues

Designing a flexible remote procedure mechanism and incorporating it into a rich programming language will not ensure that RPC is properly or enthusiastically used. To promote the full spirit of RPC programming a number of allied issues must be considered. These are issues of the programming environment rather than of the language. These issues are introduced briefly here and addressed fully in chapter 4.

Good performance of RPC is vital, for a cumbersome mechanism will be sidestepped whenever efficiency is important, and this is quite often.

Sound remote interface design is critical because poor remote interface designs, paying for the higher costs of distributed communication, will suffer performance bankruptcies unmatched in monolithic systems.

Atomic transactions are necessary for robust programming in the presence of crashes, but are generally outside the scope of this dissertation.

Autonomy of individual nodes must be considered when the desires of transparency conflict with the demands of decentralization.

Type translation of data representations between different machines and languages must be resolved for heterogeneous environments.

Remote debugging is essential so that programmers do not have to travel between physically distinct machines to debug distributed programs.

2.4 Benefits

Remote procedure call has many desirable properties. One way to characterize these properties divides remote procedure applications into three classes: resource sharing, or accessing a common resource like a file server; load splitting, or partitioning work for space or time advantages; and conversation, or distributed interaction among closely-coupled processes. The boundaries between these classes are elastic and many RPC applications fit naturally into more than one. The examples below are intended to demonstrate the wide utility of remote procedures as a distributed system-building tool, not to categorize a specific system.

All of the following examples can also be programmed with message-passing primitives. But because the examples are chosen from procedure-based languages and systems, RPC is a very natural programming context. This is especially true when existing programs must be changed, not rewritten from scratch. A system originally built from modules and procedures can be distributed with virtually no changes to the program text if language-level remote procedures are used—the distribution is largely transparent.

2.4.1 Resource sharing

In a distributed environment common resources are often made available to clients of the environment by installing the resources in stand-alone machines (or perpetual processes) which Crocker and his coworkers [19] have called *servers*. Typical servers include printers, special purpose processors, file systems, and mail transport and delivery services. As White points out in his pioneering RPC proposal [95], server clients have traditionally communicated with servers through individual, application-specific protocols. While Haverty [31] and others have formulated a general request-response protocol for most server interactions, a server communicating via remote procedures entirely avoids the special-protocol problem by providing clients, at the outset, with a procedural interface. If the interface is well defined, clients need not even know whether the resource's server is actually on their host machine or in fact on another machine of the internet. This implies that a modularly written service for a one processor system can be easily distributed if the service's local and remote interfaces are equivalent.

Of course, these procedural server interfaces are exactly what many traditional uniprocessor operating systems supply with their supervisor and monitor calls. Because they are monolithic, operating system services are usually viewed differently from the distinct services available in a distributed environment.

Distributed file services are a fertile area for resource sharing RPC. Most file servers present a procedural interface that is actually supported with a manually programmed protocol underneath; the WFS file server is a lucid example of this [87]. Gifford's thesis [25], on the other hand, describes a transactional file service that builds upon language-level remote procedures for its implementation. More importantly, Gifford also requires that his RPC have nearly transparent semantics, especially in the areas of typechecking, parameter functionality, and exception handling.

Another example of resource sharing RPC is provided by the Cambridge file server [5]. Needham [64] describes an unusual plan whereby personal computers on a local ring network will transparently use remote procedures to access a universal file server rather than their own local file stores. This conversion will be implemented by writing an *interface translation server*. Executing on its own machine, the translation server will, on one side, use RPC to give local machines exactly the same interface as their local stores. On the other side, the server will send translated commands to the central file store.

2.4.2 Load splitting

A common characteristic of many distributed computing environments is the modest processor speed and memory size of individual host computers. Demanding applications that exceed the capabilities of an individual machine are often restructured to run on several machines concurrently. In most cases this means designing an application-specific protocol and then reprogramming to use it. With remote procedures, however, just reconfiguring the modules of a system for the various machines is often sufficient. Considerably less reprogramming is needed because these modules present the same procedural interfaces as they did before. The difference is that they now execute—let us hope transparently—on a different machine.

One example of memory-reducing load splitting is where an application program's host machine has sufficient memory for the program's code, or for its display bitmap, but not for both. A description of this appears in section 2.4.4.

An example of time-reducing load splitting is given by Will Crowther's integrated circuit design program [20]. This program uses hill climbing to optimize the layout of circuit elements. An easy way to speed up the program is to split the search into several pieces and conduct each one on a separate machine. Crowther's program is written with this optimization in mind. He claims that with language-level remote procedures he could write a simple control program that partitions the search space and uses RPC to farm out the layout work, which is already written as procedures. Thus, by dividing the search space among n machines, the distributed version of the layout program would complete n times faster than the current program. But Crowther also says that without transparent RPC he would not seriously consider using multiple machines because of the protocol programming details.

It is also easy to imagine system designs that exploit load splitting with remote procedures from the beginning. When VLSI technology has advanced to the point where hundreds of processors exist on a single chip, the ability to perform vast parallel computations will demand a flexible, high-level communication strategy such as RPC. Such general high-level approaches have already been advocated by Sutherland et al. [86]; the first implementations are even appearing in advanced VLSI processors such as the Intel 432 [39]. In this setting, the complexity of VLSI is so great that application-level communication primitives must themselves have the power of remote procedures.

2.4.3 Conversation

Perhaps the largest application for RPC in distributed environments is straightforward conversational interchange—machine-to-machine, machine-to-person, and person-to-person. Examples of the first two have been presented above and are readily found in distributed system literature. The use of distributed environments for intraprogram communication, on the other hand, is not as widely explored.

The Worm programs of Shoch and Hupp [78] make a whole class of conversation programs possible. A *worm* is a distributed program composed of independent segments, each running a program in a different machine. Worms communicate over an internetwork and can *self-replicate*, or automatically spawn new segments each of which boots an idle machine and executes a program specified by the parent segment. The worm itself keeps a small amount of distributed control in each segment so that it can manage the birth, replication, and death of segments. A user of a worm simply supplies a program to be run in each segment. While worms carefully handle all of their own intersegment activities, they do not supply user applications with any high-level intersegment communication primitives. Shoch and Hupp report that the toy worm applications they tested all handled their own communication directly by sending packets. For serious worm programming, remote procedures are a suitable, higher-level primitive. RPC could also be used by the worm to coordinate its own intersegment activities.

An example of conversation in a file server utility appears below in section 2.4.4.

Other examples of internetwork communication are given by Sproull and Cohen in their discussion of high-level protocols [82]. They place particular emphasis on the application-specific design and implementation of good protocols. They also conclude that for most applications an RPC approach would be superior because remote procedures eliminate the tedious, error-prone programming between the protocol specification and its procedural interface to clients.

2.4.4 My Motivation

My motivation to study RPC comes from my personal experience in distributed computing. I have participated in the implementation of two major distributed programs, and both of these efforts would have been significantly easier and less frustrating if a viable RPC scheme had been available. Instead, I was forced in each instance to invent a specific protocol and mechanically write code mapping the desired procedural interface into the internetwork bitstream implementing the protocol.

The first application I attacked was a software performance monitor for the Juniper distributed file system [85]. Clients of the monitor insert *probes* into their programs. These probes generate events that are collected by a *monitor* and processed and displayed to clients by a *reporter*. When I joined the project, all client, reporter, and monitor programs executed on the same machine. Because the memory and processor demands of the reporter were becoming significant, one of my tasks was to make the reporter run on an independent machine that received events over a network

from the smaller and less-demanding client-resident monitor. This task, which falls into the load splitting characterization defined above, would have been easy with remote procedures because the reporter and monitor already had separate interfaces. Simply redefining the existing reporter interface to be a *remote* interface consisting of the same (remote) procedures would have solved 80% of the conversion problems. Instead, a great deal of time was spent on "obvious" protocols, mappings, and debugging.

The second distributed application I implemented was a representation translation service for Juniper. This program is used whenever Juniper changes its internal representation for files; such changes happen only when the file server software is updated at a new release. Because each server runs on a different machine a very straightforward scheme for performing the translation suggests itself: Use the existing server, running *old* software, to read its old format files and transmit them through the internet to the new server, running *new* software and using the new representation. (Conventional FTP programs are not suitable for this task because Juniper's representation of a file and its related properties are quite different from the file abstraction it presents to FTP.) This potential RPC application, which falls into the conversation class, was again implemented by inventing, programming, and debugging a specific protocol whose only purpose was to translate a *Read* procedure call on the old server into a *Write* call on the new one, e.g., in Mesa,

FOR *file* IN *Files* DO FOR *page* IN *file.pages* DO *new.Write*[*old.Read*[*Disk*[*page*]]] ENDLOOP ENDLOOP.

Puluwat—Carolines
7° 17' N 149° 13' E
Traditional navigation by wind, wave, star, and bird still survives in Micronesia

3

Survey of Existing Mechanisms

Remote procedure call is about ten years old. One of the earliest visions of remote procedure calling in a network was by Larry Roberts, considered by many to be the father of the Arpanet. In a 1970 paper now regarded as a classic, Roberts and Wessler [74] asserted:

> Any program should be able to call on the resources of other computers much as it would call on a subroutine. The resources which can be shared in this way include software and data as well as hardware.

The passage of these ten years, however, has seen message passing dominate RPC both in the Arpanet and elsewhere. Furthermore, most of these messages were not exchanged between high-level communication primitives, but between a small number of programs performing file, mail, and terminal-traffic transfers. Because of the bulk transfer dominance, this chapter discusses not only remote procedure mechanisms but allied language-level message-passing schemes as well. This is fruitful because, as we have seen, the issues behind remote procedures and messages are quite similar.

3.1 Background

The RPC mechanisms described in this chapter are restricted to implemented, operational, language-level mechanisms that give the programmer the ability to define and call his own remote procedures. This restriction excludes some elegant but unimplemented RPC designs:

> Brinch Hansen's *Distributed Processes* [12] are a vertsatile distributed programming tool. To call procedure *Q* in process *P* a programmer writes CALL *P.Q(argumentExpressions, resultVariables)*. The scheme prohibits shared variables so all interprocess communication takes place via CALL.

Lampson's concise remote procedure model [49] is proposed for use in data storage systems with atomic transactions. The scheme is unusual because it does not have exactly-once semantics even in the normal, no-crash case. This RPC model is considered further in chapter 4.

Schuman, Clarke, and Nikolaou's language-level RPC proposal for Ada [77] addresses most of the essential issues. A primary objective of the scheme is making no changes to Ada and, as a result, the RPC mechanism is somewhat unwieldy. In addition, it does not address call semantics in the presence of crashes or parameter functionality.

The restriction to operational RPC mechanisms also excludes systems that, while advertising remote procedure calls, actually implement them manually rather than with a general language-level facility. The Juniper file system [85] and the intercommunication portion of IBM's Customer Information and Control System [27,40] are examples of such systems.

In addition to the previous procedure-based mechanisms, a number of message-based operating systems offer invocation primitives that are useful for request-reply interactions. These primitives are usually not integrated into the underlying programming language. For example, Demos, RIG, Roscoe, and Simos have *Call* operations, Thoth has *SendRequest*, StarOS has *Invoke*, and Medusa has *UCall* [2,51,79,84,15,41,68]. The semantics of parameter passing and typechecking are different for each of these mechanisms. In addition, the callee in these systems is usually not a procedure but rather a *process* that explicitly receives *call* messages and sends *return* messages. For this reason, these message-based invocation primitives are excluded from the following discussion of RPC mechanisms. RIG's *Call*, however, is included as a typical example of invocation in a message-based operating system.

3.2 Mechanisms

The program samples used to illustrate each mechanism in this chapter are written in the mechanism's own host programming language.

3.2.1 Sail's Message Procedures

An early system developed at Stanford by Feldman and Sproull in 1971 provides *message procedures* [22]. The general form of a remote message procedure call is:

handle ← ISSUE (*directive, source, destination,* MESSAGE *Procedure(arg1,arg2,...)*)

Because users of message procedures explicitly send messages (the ISSUE), this scheme is not really RPC in the strong sense of the definition. It does, however, have several novel features.

Message procedures are integrated directly into the Sail language and have full compiler support.

The syntax of the remote MESSAGE call is exactly local procedure syntax, i.e., *Procedure(arg1,arg2,...)*.

> The arguments of remote calls are typechecked and passed by value. Functions are illegal because results are not supported.
>
> Message procedures are used for communication between processes in (almost) disjoint virtual memories.

The message procedure mechanism, while primitive, was an accurate harbinger of systems to come.

3.2.2 The Arpanet's Procedure Call Protocol and Distributed Programming System

Early work on program communication in the Arpanet environment was generally described in terms of message-oriented interprocess communication (IPC). Walden's pioneering article [92] on an extended IPC for the Arpanet raised various issues of remote procedures versus messages in 1972. Discussion about the "correct" network mechanisms for interprogram and interprocess control ensued in Arpa's Network Working Group. Two opposing positions emerged: At SRI, Postel and White [70] set forth a strong case for using a (remote) *procedure call protocol* as the fundamental implementation tool of the National Software Works (NSW). The other camp, lead by Schantz [76], attacked this ambitious RPC proposal as inappropriate for distributed computing. The counter proposal by Schantz and Thomas was MSG [90], a simpler message-based IPC designed by a group at BBN.

The message advocates prevailed in this discussion, and MSG became the underlying communication mechanism of the NSW. The remote procedure group continued in its course, however, and in 1976 various advanced proposals by White eventually found their way into a distributed version of SRI's NLS text editor. While the *Distributed Programming System* (DPS) [96] never included sufficient language support to make local and remote calls syntactically transparent, much of the required functionality was present. Some relevant DPS primitives are illustrated by the following system routines.

> *OpenPackage*: PROCEDURE (*package*: STRING) RETURNS (*handle*: PACKAGE);
>
> *CallProcedure*: PROCEDURE (*handle*: PACKAGE, *procedure*: STRING, *arguments*: LIST OF ANY,
> *argumentMask*, *resultMask*: LIST OF BOOLEAN)
> RETURNS (*outcome*: BOOLEAN, *results*: LIST OF ANY);
>
> *ReadVariable*: PROCEDURE (*handle*: PACKAGE, *variable*: STRING) RETURNS (*value*: ANY).

Notable here is the PACKAGE concept, which allows DPS clients to construct simple remote interfaces including both procedures and variables. Concern with the binding issue is shown by the package concept and by the fact that packages, procedures, and variables are identified by a string name. Clients concerned with efficiency can explicitly ask for certain arguments and results to be ignored (not transmitted) via the corresponding *parameterMasks*. Exceptions are handled with independent *NoteProcedures*, and DPS included a means of aborting remote procedures (processes).

3.2.3 Hamlin's Cages

Outstanding work in remote procedure call was done by Griffith Hamlin at the University of North Carolina in 1975. Hamlin's thesis [30] discusses *Cages*, a system for configuring graphics application programs between a host computer and a satellite graphics processor. In this context he designed and built a preprocessor that scans PL/1 (graphics) programs with special configuration declarations and produces two PL/1 output programs—one for the host, an IBM 360, and one for the satellite, a DEC PDP11. These two programs execute in Hamlin's augmented runtime environment which provides RPC support for both machines.

The style of the configuration declarations is indicated in the following example. The SATELLITE and HOST keywords indicate which processor executes *procedure*. The optional USES and SETS clauses define which, if any, of the arguments are passed only for value or result.

> DECLARE *procedure* ENTRY (*argumentList*) RETURNS (*resultList*)
> SATELLITE *or* HOST SETS (*argumentListSubset*) USES (*argumentListSubset*)
> CALLS (*procedureList*) ONS (*onConditionList*) EXTERNALS (*globalVariableList*) ...
>
> DECLARE *globalVariable* EXTERNAL ...

It is clear that these declarations require information that is certainly known to the compiler, or could be determined by a more elaborate preprocessor. In spite of this lack of transparency, Cages is a significant achievement. In particular:

> Programmers use *only* procedures, and local and remote syntax are identical. The *call* and *return* messages of the implementation are completely hidden.
>
> In addition to procedures, Cages handles both remote global variables (which are migrated back and forth on demand) and remote PL/1 ON-conditions (a flexible exception mechanism).
>
> Translation between data representations on the 360 and PDP11 is handled automatically.

Hamlin's thesis attacks the problem of how to write a single graphics application program that is automatically compiled for two machines. In solving the basic graphics problem he creates an outstanding RPC mechanism as well. By addressing a well-defined two-processor graphics application, Hamlin achieves significant remote functionality and sidesteps the problems of multimachine binding and crash recovery.

3.2.4 Rochester's Intelligent Gateway

Rochester's Intelligent Gateway (RIG) [51] is one of the first systems to supply a standard remote procedure facility for internetwork clients. RIG's communication medium is a simple and elegant message-passing mechanism that provides interprocess communication for all processes (and thus machines) in the Rochester internet. As an adjunct to this message-oriented system RPC primitives are supplied. A remote call looks like:

Call (*msg*, *procId*, *arg1*, *arg2*, ..., *timeout*) —> *result*.

Here *msg* is a message template, *procId* is the procedure name, *arg1* and *arg2* are its arguments, and *result* is its result. A *Call* blocks until either the results come back or the specified *timeout* interval expires. For calls with no *result* the similar *Instruct* operation starts a remote call and resumes the caller without waiting. The RIG scheme, while attractive, has these disadvantages:

> Remote "calls" are message operations with opaque message syntax and not transparent procedure syntax.
>
> No typechecking is done, as the typeless Bcpl language is used throughout the RIG system.
>
> No assistance is provided with binding or configuration. The process implementing remote procedures must be running at the time of the call and be known to the caller.

On the positive side, RIG does have a stack-following *Error-ErrorSet* exception mechanism that can be used by remote procedures to pass signals between machines. In addition, *arg1* and *arg2* can be used in conjunction with a type template mechanism to pass buffers containing string and array parameters.

3.2.5 CMU's Multi-Media Message System

Carnegie-Mellon's *MMMS* project is developing an electronic mail service that combines speech, text, and graphics with a display-oriented user interface. The prototype system uses a DEC Vax for processing and Altos as workstations. These machines communicate with a remote procedure mechanism designed by Gene Ball [1]. The scheme, similar to Hamlin's, provides only for unilateral calling from the Vax to the Altos.

Remote procedures are written in Bcpl on the Altos. The definitions file for a remote interface is slightly annotated so that a postprocessor can read the definitions file and generate a file of stub procedures (section 2.1.4). These stub routines, written in the Unix C language, implement the remote interface on the Vax. Ball provides a suitable runtime environment on both machines. One of the novel features of the mechanism is that intermachine communication does not build on expensive level 2 streams. The reliable two-packet protocol used in their stead is very efficient for the slower Altos.

3.2.6 Cook's StarMod

StarMod (also called *MOD) is a language for distributed programming developed by Robert Cook at the University of Wisconsin [17]. Starting with Modula [97], Cook enhanced the basic language using concepts from Distributed Processes [12]. StarMod introduces two new module types for distributed programming—*network modules* and *processor modules*. Here is a simplified example:

```
NETWORK MODULE ExampleDistributedSystem =
                    (Client, FileServer), (FileServer, AuthenticationServer);
    PROCESSOR MODULE Client;
        Processes, procedures, and data.
        END Client;
    PROCESSOR MODULE FileServer;
        Processes, procedures, and data.
        END FileServer;
    PROCESSOR MODULE AuthenticationServer;
        Processes, procedures, and data.
        END AuthenticationServer;
    END ExampleDistributedSystem.
```

In this example there are three autonomous entities: a *Client*, a *FileServer*, and an *AuthenticationServer*. Network modules define the topology of the communication system, and in the example the *FileServer* can talk to both the *Client* and the *AuthenticationServer*, but the *Client* and *AuthenticationServer* are unable to talk to each other directly. Processor modules define the activities of a set of (virtual) machines with a common address space. Each *process* in a processor module shares memory with all other processes in the same processor. Intraprocessor communication is by local procedure and process calls. (A process call is basically the execution of a procedure in a newly created process.) Interprocessor communication takes place by remote calls (written in the same fashion as local calls) to other processors. Synchronization is accomplished with *interface modules* and *signals* (Modula's monitors and condition variables).

StarMod is a significant achievement: Cook has built an operational (uniprocessor) system for distributed programming with a compiler and binder that permit separate compilation. StarMod addresses the important language-levels issues of uniform call semantics, syntactic transparency, intermodule binding, and strong typechecking. On the other hand, it does not deal with call semantics in the presence of crashes or parameter functionality. In addition, the close coupling of module binding specifications and network topology information can lead to unnecessary rebinding in realistic systems.

(In a later paper [18], Cook revises StarMod substantially, moving away from the procedure orientation described above. The revised language includes explicit interprocess communication *ports* that queue messages for program *regions*.)

3.2.7 Clu's Guardians

Guardians are an extension to Clu [56] proposed by Liskov [57]. Although not implemented, Guardians are of special interest and merit inclusion in this discussion.

Communication between Guardians is by strongly typed messages. While separate SEND and RECEIVE primitives are available to Guardian programmers, of particular interest is the CALL primitive. Liskov has proposed the following syntax for CALL [34,59]:

```
CALL Operation(args) ON port
        WHEN Response1(formalArgs): S1
        ...
        WHEN ResponseN(formalArgs): SN
        WHEN FAILURE(s: STRING): Sfailure
        WHEN TIMEOUT(time): Stimeout
        END.
```

Each *ResponseN* is a different reply to the called *Operation*, and the FAILURE and TIMEOUT responses are for error handling. A *port* is a unique global name attached to the receiving Guardian. (Observe that ports, when completely specified with formal parameters, have many of the characteristics of a procedure interface.) The current semantics of CALL, which have changed markedly from early proposals, is what Liskov calls *at-most-once* [58]. This is exactly-once semantics with atomicity and indivisibility, which means that the computation of a CALL completes totally or not at all: the intermediate states that can happen in the presence of crashes (section 2.2.2) are eliminated by integrating an atomic transaction mechanism [48] directly into the RPC scheme.

Guardians, a topic of current research, are of interest because they address the critical semantic, binding, and error recovery issues of distributed program communication. At-most-once semantics are discussed again in chapter 4.

3.2.7.1 Type Translation and Transmission

Methods to transmit Clu's strongly typed objects between nodes are a vital part of the Guardian work. Herlihy's enlightening thesis, "Communicating Abstract Values in Messages" [33], discusses an operational scheme. Herlihy's method invokes user-written *encode* and *decode* operations to convert a user's internal, concrete representation for a type back and forth between an external, standard representation, or *xrep*. A highly desirable property of the scheme is that different Guardians can use different concrete representations for the same abstract type; this concreteness is hidden behind *encode* and *decode* and the fact that only *xreps* are communicated. The Clu runtime implementation takes responsibility for transmitting *xreps* between Guardians, even those executing on processors with different built-in type representations. Herlihy also discusses methods for dealing with cyclic and acyclic sharing.

3.2.8 Spector's Remote Memory Operations

Intrigued by the megabit bandwidth of local networks, Alfred Spector [80,81] has added synchronous high-speed remote memory operations to a network of personal computers. This work, which uses message passing to implement a shared memory architecture similar to Cm* [24,41], is important because it uses a general purpose internetwork as its transport mechanism. Spector microcoded three new instructions for the Alto:

RLDA *address, machine*. Remote Load Address returns the contents of *address* on *machine*.

RSTA *address, machine, value*. Remote Store Address stores *value* in *address* on *machine*.

RCS *address, machine, value1, value2*. Remote Compare and Swap; this instruction is used as a low-level semaphore.

Each instruction performs its operation synchronously and atomically. The implementation sends two Ethernet packets in the usual case, more if errors occur. Each of these instructions executes in total of 155 microseconds when *machine* is on the same Ethernet.

Spector's work is extremely valuable because it establishes that synchronous remote memory operations are viable in a fast, low latency, *loosely coupled* network. Cm* [24] demonstrated this result first, but the Cm* architecture is tightly coupled in the overall spectrum of distributed systems. In one light, Spector's remote memory operations can be viewed as highly optimized remote procedure calls that indicate an upper bound on the performance of RPC mechanisms in a similar network.

3.3 A Brief Evaluation

Each of the remote procedure schemes just surveyed attacks and solves some particular problems of remote calls. But none of these solutions—with the possible exceptions of StarMod and Guardians—yields a uniform or transparent mechanism for remote procedures. Cages and MMMS all have identical local and remote point-of-call syntax, but because of their two-machine nature none addresses the multimachine binding problem. DPS has a general scheme for both binding and calling remote procedures, but the local and remote syntax is different. Message procedures, RIG, and Guardian CALLS all support remote invocation, but the calls are visibly embedded in messages and do not necessarily invoke procedures. StarMod addresses many issues of local and remote transparency, but does not deal with multimachine crashes. Spector's machine-level work cannot be evaluated against language-level criteria.

Kauai—Hawaii
21° 57' N   159° 40' W
Cook discovers the "Sandwich Islands" near Waimea. 20 January 1778

4

Ideal Properties of a Transparent Mechanism

The design of transparent remote procedure mechanisms requires paying attention to all of the issues outlined in sections 2.2 and 2.3. This chapter divides these issues into two groups: *essential issues* that must be addressed by RPC schemes that provide transparent local and remote semantics in a homogeneous programming language, and *pleasant issues* that make remote procedures a feasible and comfortable tool for programming distributed systems.

The issues are explored to varying depths. The first reason for this variation is that the dissertation focusses primarily on the essential issues; the second is that some of the issues are more complicated and less well understood than others. All of the issues, however, are addressed in enough detail that a reasonable resolution for each one emerges. These resolutions are expressed as a set of ideal *essential* and *pleasant properties* for transparent RPC mechanisms.

4.1 The Essential Issues

The first four-fifths of this chapter delve into the five essential issues: call semantics, binding and configuration, typechecking, parameter functionality, and concurrency control and exception handling. Following the philosophy set forth in the introduction, each issue is surveyed until the fundamental problems and tradeoffs are exposed, although not all aspects of each issue are pursued in the remainder of the thesis. The first topic, call semantics, is easily the most complicated. This is because semantics are at the heart of any communication mechanism, and there is a wide spectrum of choices for RPC.

Throughout this section, the reader who wants to regain his high-level perspective on the issues can always turn to the end of the chapter, where a summary of essential properties captures the salient points of each issue.

4.1.1 Call Semantics

In section 2.2.2 the semantics of local procedure call were characterized as exactly-once in the absence of crashes and last-one in the presence of crashes. While transparency requires that remote procedures have precisely these semantics, a number of nontransparent semantics have been proposed for distributed communication primitives. Examining these alternative schemes is worthwhile for two reasons:

> They explore different assumptions about "good" primitives. This aids in understanding the assumptions made for transparent semantics.
>
> They deal with crashes in different ways. Considering them in order of robustness gives a useful and gentle introduction to providing last-one semantics in the presence of crashes.

Before considering other call semantics it is useful to review two important policies of exactly-once semantics for noncrashing remote calls. The first policy is reliable transmission. In chapter 2, reliable transmission was suggested for each remote call to ensure that the *call* and *return* messages that send arguments and return results are received exactly once at each end. The second policy is blocking the caller. A local caller blocks for a synchronous remote call just as it waits for a synchronous local call. These constraints are now relaxed so that we can investigate some resulting call models.

4.1.1.1 At-least-once Semantics

The *call* and *return* messages of the RPC implementation can be sent and received multiple times if reliable connections are no longer required. Thus, if the caller's communication policy is to retransmit the *call* periodically until a *return* is received, it is possible that the callee will repeat the execution. This is especially true in the presence of crashes: if the callee crashes after completing the call's execution but before the caller receives the *return*, then the caller will again send the *call* and the now-restarted callee will repeat its execution. Variations of this scenario apply when messages are lost or delayed in the internet. With this mechanism, then, the receipt of a *return* by the caller guarantees that the call happened *at least once*, but the caller does not know how many times it happened or even which *call* his results are from.

The utility of at-least-once semantics to the distributed system designer is not clear. They are certainly not equivalent to local call semantics, and the programmer must remember and use two distinct methodologies for local and remote calls. In fact, at-least-once semantics are those of unreliable messages—the "procedure call" is a disguised *Send* followed by a *Receive* for the very first reply that is successfully received. Because at-least-once calls have the semantics of the level 1 datagram mechanism, they can be used to implement other semantics—such as exactly-once—in exactly the same fashion that protocol levels 2 and above are built on level 1. Thus at-least-once calls are a very flexible primitive, but only if transparency is sacrificed.

On the positive side, Liskov [58] points out that at-least-once calls have the enjoyable property that the caller need not know whether the callee has crashed or not. The possibility of crashes is inherent in the mechanism because the at-least-once semantics guarantee is satisfied (trivially) in

both the presence and absence of crashes. Liskov at one time proposed at-least-once semantics for Guardian communication, but she now favors much stronger, atomic, at-most-once semantics [59]. This is done by including a transaction mechanism within the RPC scheme. Some intermediate RPC semantics are described before we consider the implications of transaction schemes.

4.1.1.2 Last-of-many Semantics

An interesting variation on the at-least-once model is one where the call can repeat any number of times, but where the caller is guaranteed to receive the results of the very last one of the many calls; this is called *last-of-many* semantics. Basically, this scheme matches the sequence numbers in *call* messages with those of returning *return* messages to discard the results of all but the last call. Each *call* message is assigned a new sequence number, even when it is a retransmission of an existing call. Figure 4.1 demonstrates this. Here three *calls* are sent from *A* to *B* before the right *return* is received in time.

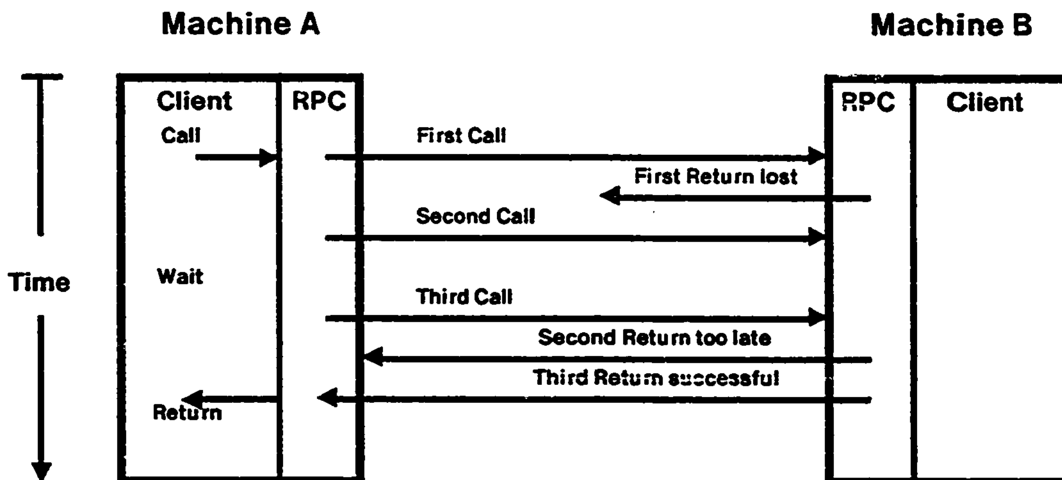


Figure 4.1: A two-machine last-of-many remote procedure call.

The simple last-of-many scheme works well in the two-machine case. Its most important property is that it guarantees last-of-many semantics even in the presence of crashes on either machine. The addition of a third machine, however, introduces situations that violate this property whether there are crashes or not. The basic model does not guarantee transitive last-of-many semantics across machines. The problem is that each client call on *A* can generate any number of independently executing calls on *B*. If each of these calls on *B* must call machine *C*, the executing calls on *C* can complete in any order. Thus *B* can return to *A* without using the last-of-many result from *C*. This is illustrated in figure 4.2.

Transitivity is violated because A_1's call to *C* completes after A_2's, yet it is A_2's results that *A* returns to the client. Fortunately, a slightly more elaborate sequence numbering scheme that attaches the numbers to the client *calls* rather than to the repeating *call messages* fixes this problem in the absence of crashes. Lampson, who first proposed last-of-many semantics [46], has made this

revision to the basic model [49]. His algorithm, written in a cross of Pascal and Mesa, appears in algorithm 4.1 for the stout-hearted reader. In the algorithm, individual call invocations (not *call* messages) are uniquely identified by *request* numbers. Last-of-many semantics are achieved for each request by using separate *id* sequence numbers for each *call* and *return* message. The *Send* and *Receive* operations are unreliable.

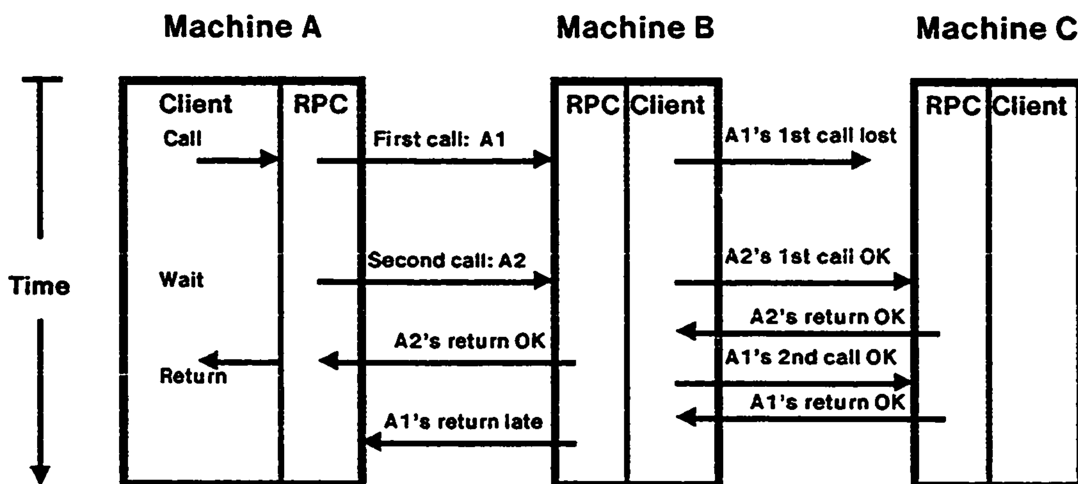


Figure 4.2: A three-machine nontransitive last-of-many remote procedure call.

Lampson's algorithm guarantees last-of-many semantics between two machines, even in the presence of crashes. When there are more than two machines, however, the guarantee is void if there are crashes. The desired transitivity is unobtainable when intermediate machines in the call chain fail. These intermediate failures can leave outstanding calls that continue to execute even though they were initiated by now-crashed machines. Lampson denotes such calls *orphans*. For example, in figure 4.3, assume B_1 crashes and leaves outstanding calls executing on *C*.

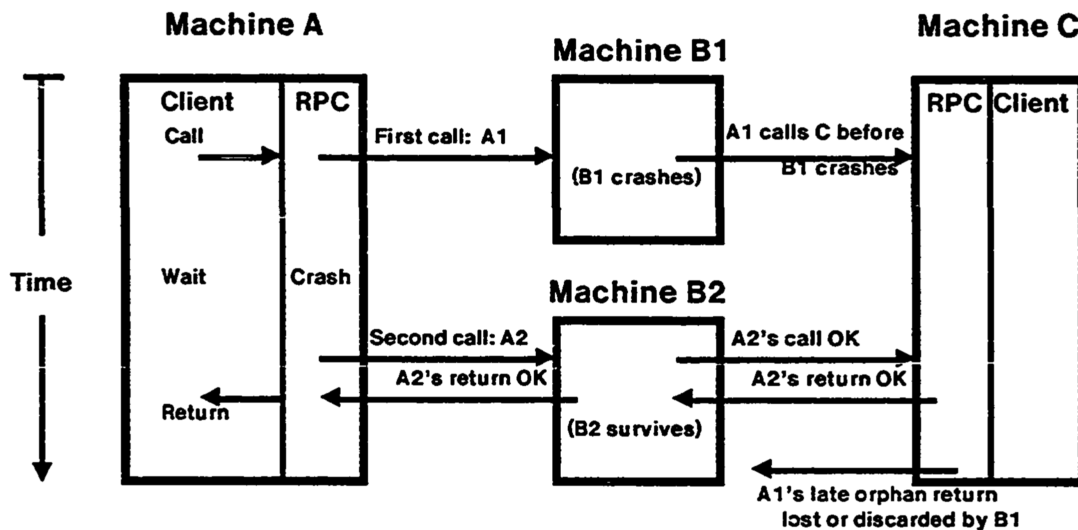


Figure 4.3: An orphaned remote procedure call that violates last-of-many semantics.

If A decides to retry its call through another machine, B_2, and B_2 calls C, C can return results to B_2 and thus to A *before* B_1*'s* outstanding calls on C die out. The orphan, B_1, causes a violation of transitive last-of-many semantics. Furthermore, B_2 need not be a different machine: it could just as well be B_1 after B_1 completes crash recovery. Orphans are discussed further below.

One crucial question to ask about last-of-many semantics is their utility to the application programmer. Even this revised model does not have exactly-once semantics in the normal, no-crash case. This is because communication between caller and callee is not reliable for the *return* message. In the algorithm, after the *return* message is sent, the callee's RPC mechanism erases its record of the call. If this message is lost (which can happen without a crash), then the caller must repeat its *call* message and the procedure will be reexecuted and a new *return* message sent. This process will repeat until a *return* is received; the semantics are not exactly-once.

To be fair, in Lampson and Sturgis's transaction-oriented programming paradigm [46,48], last-of-many semantics are acceptable because all remote operations must be inherently *restartable*, that is, capable of delivering the same results after repeated executions. When operating outside their specific paradigm, however, exactly-once semantics are desirable in the absence of crashes. Fortunately, achieving exactly-once semantics from the last-of-many scheme is straightforward if the communication uses a reliable connection. One simple connection method is to have the callee hang onto a *return* message until the next *call* message acknowledges that the *return* was received—or, if it is a repeat *call* message, the waiting *return* is simply resent. The details of this scheme, which gives exactly-once semantics in the absence of crashes, are considered in chapter 5.

4.1.1.3 Crash Semantics

As we have seen, crash handling is exacerbated in distributed systems. An internetwork environment intensifies the problem because apparent process failure can occur for two reasons. The first is communication failure: This can happen when the internetwork breaks and the remote process appears unresponsive. The second is bona fide process failure: This can happen because of a specific mortal error in the process or because the process's machine crashes.

In the absence of crashes accomplishing exactly-once semantics is conceptually straightforward. Now consider a crash.

In a single-machine environment, if a given procedure call is repeated after crash recovery we know that *all outstanding activity of the call has ceased.* This is true because all processes are recreated (either from a checkpoint or by a boot) and therefore all outstanding activity in any old processes is terminated. In fact, operating systems usually extend this all-activity-ceased property throughout their domain by *resetting* I/O channels, disk controllers, and other semi-autonomous devices which can influence the processor's memory and state. Thus, in the presence of crashes, single processor exactly-once local procedure semantics degenerate into the last-one semantics discussed in section 2.2.2.2.

```
{ Remote procedures, using Send and Receive for messages, and UniqueId for unique identifiers }
type ID = 0..2^64; const timeout=...;
type Message = record
  state: (call, return); source, dest: Processor, id, request: ID; action: procedure; val: Value end;

{ The one process which receives and distributes messages }
var m: Message; var s: Status; while true do begin (s, m) := Receive(); if s=good then
      if m.state=call and OKtoAccept(m) then StartCall(m)
      else if m.state=return then DoReturn(m) end;

{ Make calls }
monitor RemoteCall = begin
type CallOut = record m: Message; received: Condition end; var callsOut: set of ↑CallOut := ( );
entry function DoCall(d: Processor, a: procedure, args: Value): Value = var c: ↑CallOut; begin
      New(c); with c↑ do with m do begin
            source := ThisMachine(); request := UniqueID(); dest := d; action := a; val := args;
            state := call; callsOut := callsOut + c { add c to the callsOut set };
            repeat id := UniqueID(); Send(dest, m); Wait(received, timeout) until state=return;
            DoCall := val; Free(c) end end;
entry procedure DoReturn(m: Message) =
   var c: ↑CallOut; for c in callsOut do if c↑.m.id=m.id then begin
      c↑.m := m; callsOut := callsOut - c { Remove c from callsOut }; Signal(c↑.received) end;
end RemoteCall

{ Serialize calls from each process, and assign work to worker processes }
type CallIn = record m: Message; work: Condition end
monitor CallServer = begin var callsIn, pool: set of ↑CallIn := ( );
entry procedure StartCall(m: Message) = var w, c: ↑CallIn; begin
      w := ChooseOne(pool) {waits if the pool is empty};
      for c in callsIn do if c↑.m.request=m.request then begin c↑.m.id := id; return; end
      pool := pool - w; callsIn := callsIn + w; w↑.m := m; Signal(w↑.work) end;
entry procedure EndCall(w: ↑CallIn) = begin
      Send(w↑.m.source, w↑.m); callsIn := callsIn - w; pool := pool + w; Wait(w↑.work) end;
end CallServer

{ The worker processes which execute remotely called procedures }
var c: ↑CallIn; New(c); c↑.m.source := nil; EndCall(c); with c↑.m do
while true do begin val := action(val); state := return; EndCall(c) end;

{ Suppress duplicate messages. Needn't be a monitor, since it's called only from the receive loop. }
type Connection=record from: Processor; lastID: ID end; var connections: set of Connection: =( );
function OKtoAccept(m: Message): Boolean = var c: ↑Connection; with m do begin
      for c in connections do if c↑.from=source then begin
            if id≤ c↑.lastID then return false; c↑.lastID := id; return true end;
      { No record of this processor. Establish connection. }
      if action= UniqueID then return true { Avoid an infinite loop; OK to duplicate this call. };
      { For good performance the next two lines should be done in a separate process. }
      New(c); c↑.from := source; c↑.lastID := DoCall(source, UniqueID, nil);
      connections := connections + c; return false { Suppress the first message seen. } end
```

Algorithm 4.1: Lampson's unabridged last-of-many remote procedure algorithm [49].

Now consider a machine crash in a distributed environment. To provide last-one local procedure semantics, all outstanding remote activity must cease before restarting. Conceptually, a distributed *reset* must be issued to all the threads of control that the crashed machine has left outstanding on other machines. But these threads are precisely the orphans discussed before. Therefore, to achieve last-one semantics in a distributed system, crash recovery must ensure that *all orphaned calls to other nodes are exterminated.* This guarantee must be met while the crashed machine is in recovery and before it can repeat any of its calls that would violate last-once semantics. The process of killing all of a node's orphans is called *extermination.*

These last-one semantics for a distributed system are quite similar to last-of-many semantics discussed above. The main problem with last-of-many semantics was that they could leave orphans. If an orphan extermination scheme is added to the last-of-many model, then the new algorithm will have last-one semantics in the face of crashes.

Meeting the goal of identical local and remote call semantics is now possible because of the orphan extermination requirement. Section 4.1.1.2 showed how to get exactly-once semantics from the last-of-many algorithm in the absence of crashes; exterminating the algorithm's orphans gives last-one semantics in the presence of crashes. The vital ingredient is orphan extermination.

4.1.1.4 Exterminating Orphans

One obvious and unacceptable method of exterminating orphans is to issue a master *reset* that brings down the entire distributed system when any machine crashes. This is, of course, unacceptable behavior, but it is just what many conventional systems do when a vital process fails—restart everything.

A second solution is to crash just those machines containing orphans. If this improvement still appears extreme, consider a further refinement that exterminates just those processes that are executing orphaned procedure calls. This seems ideal and very similar to the single-machine case. The difficulty with either of these two schemes, however, is tracking down all of the orphaned processes—in effect, determining a crashed machine's domain of influence by following the call stack from machine to machine. This becomes hard when these orphaned calls are themselves executing on crashed processors, making stack-following impossible.

An additional problem with these selective schemes is that the set of outstanding remote calls at any time must be kept in stable storage so that the set will survive crashes and be available for the recovery phase that exterminates orphans. (*Stable storage* [48] is a storage medium that has an extremely high probability of surviving machine crashes and media failures; it must also have an atomic *write* operation. Some examples of stable storage include pair-redundant disk pages and nonvolatile memory units.)

An entirely different approach from extermination is to let the orphaned calls continue executing until completion. If these orphaned calls retain their results, fully expecting the crashed parent machine to repeat its calls and *adopt* its orphans, then all is well and the semantics are last-one. An

orphan is *adopted* when a postcrash instance of its parent node retransmits all of its orphaned *call* messages in the hope of finding and reuniting with all its orphans. The usual method of ensuring that orphaned calls repeat is by checkpointing the caller before each call.

Whether crashed calls repeat is a decision usually made above the remote procedure mechanism, that is, it is a decision of crash recovery methodology. Building this decision into the RPC mechanism involves expensive transaction mechanisms—such as doing a checkpoint for each remote call—whose cost appears prohibitive. Furthermore, permitting the caller to change his decision and *abandon* orphans could violate uniform semantics: if the caller does not repeat his call and the orphan is not adopted (or exterminated), then even last-one semantics cannot be guaranteed.

The choice of appropriate orphan schemes is open. Either of the two presented here offers exactly-once semantics in the normal case. Both, however, require special treatment of orphans after crashes—the extermination scheme because it must wait until all orphaned activity has ceased before restarting, and the adoption scheme because it must rendezvous with orphaned activities before continuing. Extermination is the least expensive scheme giving transparent local and remote semantics, and I select it for this reason. The details of some orphan extermination algorithms are presented in chapter 5.

4.1.1.5 Immediate-return Semantics

Procedure call is not a good analog to message passing for unilateral communication. Consider an example where plotting commands are sent to a remote plotter: Each command is asynchronous and can be sent independently, generating no response. If the commands are implemented as remote procedures, then each command procedure can give an *immediate return* as soon as the underlying *call* message is constructed and queued for delivery. There are no *return* messages.

These immediate-return semantics—easily and well-modelled with unilateral message passing—are useful whenever communication is to a concurrent "output only" device or resource. Here are the conditions and problems of immediate-return procedures.

> Immediate procedures cannot return results—that is, there are no immediate-return functions. Similarly, they can have no VAR arguments; all parameters must be called by value.
>
> Whether or not procedure parameters are permitted is unclear. In Algol-like execution environments, there can be dangling reference problems with the procedure parameter's environment.
>
> Callers of immediate procedures can receive no synchronous exceptions. Handling the remotely generated exceptions of immediate procedures is uncertain. Because the compiler can distinguish immediate procedures (say, by the appearance of IMMEDIATE), it could enforce a you-must-handle-'em policy on immediate procedures.

These problems are difficult. Apparently, immediate procedures must have totally inconsistent semantics to meet their goal of increased efficiency. We now look at a consistent but less elegant solution where clients program immediate-return semantics themselves.

In a simplistic approach the programmer simply forks the remote call as a separate process. This process performs the call, waits for the null return, and dies. The problems with this method are twofold:

> Creating a separate process for each operation corrupts the correct sequencing of the operations. The concurrency causes this.
>
> Spawning a separate process for each operation causes performance to suffer. Creating a process is usually significantly more costly than performing a simple plotter call with two integer arguments.

If the client is willing to do a bit more work then both of these problems can be circumvented: The programmer creates a single, asynchronous, auxiliary process that receives plotter commands from the main process. This second process performs normal, non-immediate remote calls to the plotter—perhaps even batching them together—while the main process continues to "plot" with abandon. The main process is, of course, scheduled to run while the second is blocked on a remote call. (Notice that even this revised scheme is just a complicated restatement of the unilateral message-passing model.)

There is a vital observation here: The client-programmed implementation outlined above is exactly how most operating systems implement I/O resources. For example, consider file service on a timesharing system. The client *GetByte* and *PutByte* routines almost always access a buffer in the client's own address space. Even those operating systems that keep the buffer in the executive's address space avoid an expensive context switch for these routines—the calls merely cross a protection boundary. On the other hand, when the buffer is empty (or full), the operating system usually communicates with a file service process to have the buffer handled asynchronously. Thus short, inexpensive calls are performed locally and synchronously; long, expensive I/O is performed remotely (in another process) and asynchronously.

There are two advantages of immediate procedures over normal, non-immediate ones: The first and obvious one is their ease of use by clients—quick programming with good performance. The second and less obvious one is the typechecking that the simpler but efficient I/O stream approaches do not perform. For example, operating systems typically assume no structure in the data they handle; a *PutByte* of a character on one day can be retrieved with a *ReadByte* of an array of eight Booleans on the next. Immediate procedures, however, give good efficiency and a guarantee that a plotter command sent by one host on the internet will be received only as a plotter command on the next.

In the absence of immediate procedures, clients can always program their own variants of immediate-return semantics as outlined above. This gives typesafety, efficiency, and semantic consistency at the cost of some additional programming. This course is recommended.

4.1.1.6 Sequencing Semantics

As mentioned above, an allied issue of immediate-return calls is the proper *sequencing* of those calls. For example, if a program is writing vectors to a remote graphics display, the order in which

the vectors are drawn usually does not matter. Each vector is independently specified (e.g., with a pair of endpoint coordinates) so that the immediate *call* messages can arrive at the display in any order. On the other hand, most local and remote calls—including the immediate calls on sequenced resources like the plotter above—need to be properly ordered. This is occasionally true even in the hypothetical vector example above: Clearing the display, for instance, is an operation that demands proper sequencing with respect to the vectors that are drawn before and after it.

If the lack of sequencing is important to an application, it can always bracket appropriate operations with COBEGIN and COEND, or it can resort to sending messages directly. Since unsequenced language-level procedure calls introduce chaotic semantics, they cannot be permitted when transparent semantics are the goal.

4.1.1.7 Invocation Schemes

Until now, "executing the remote call" has been an informal notion. The actual mechanism that performs *RTransfers* between remote contexts has been unclear. Two possible schemes are now presented in figure 4.4. In each diagram three machines—*A*, *B*, and *C*—are all transferring to procedure (context) *P* in machine *M*.

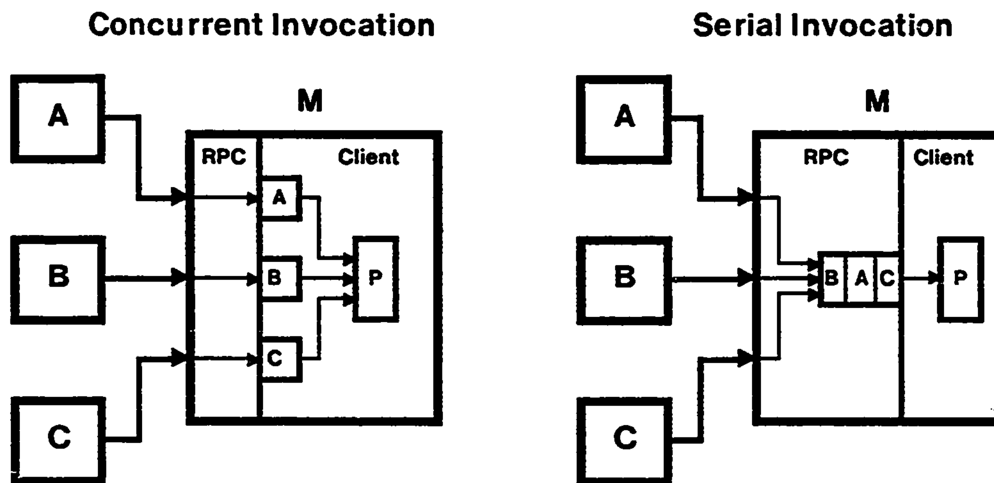


Figure 4.4: Two remote invocation policies: concurrent via processes and serial via call queueing.

In the concurrent invocation diagram, each incoming *RTransfer* is immediately dispatched by the RPC environment to a separate process that concurrently performs the transfer. In the serial invocation diagram, each transfer is queued by the RPC environment and *P* is invoked serially (without overlap) by *C*, *A*, and finally *B*. Here are some observations on the two methods:

> *Concurrent invocation.* The concurrent scheme models *Transfer* semantics accurately. For example, if *A*, *B*, and *C* were parallel processes executing on one machine, then procedure *P* could be invoked by any of *A*, *B*, and *C* at the whim of the scheduler. Of course, if *P* handles shared data then *M* should be a monitor that synchronizes invocations of *P* (and other procedures in *M*). The concurrent scheme extends this behavior to the multimachine case by creating a separate process in machine *M* for each *RTransfer*. These remote

processes correspond exactly to the calling processes and compete for P just as in the single-machine case. The efficiency of these distinguished RPC-spawned processes may benefit greatly from special treatment.

Serial invocation. The serial scheme, on the other hand, actually introduces strong message-passing semantics into the procedure-monitor world. Incoming *RTransfer* messages are queued and their destination process—the one containing M—"receives" them one at a time by virtue of the mediating RPC environment. This has the effect of serializing access to M and P. An advantage of this serialization is that P is automatically synchronized, just as P is when it is a monitor entry procedure. A disadvantage is that deadlocks can occur that will not happen in the concurrent invocation scheme. For example, a deadlock can result if C waits on a condition variable that A or B would normally signal if they were concurrent and not waiting calls. Furthermore, deadlock can occur even without synchronization problems: imagine that C has called P, and P calls back to C recursively—P's call to C will be queued awaiting completion of the one from C to P!

This deadlock problem is an indication that a hybrid invocation scheme is the wrong general approach. Programs that use *Transfer* on a single machine should continue to work in exactly the same fashion when their modules use *RTransfer* among multiple machines. Uniform local and remote semantics—either procedure or message based—are desired because they present the same concurrency model to the programmer. In the *RTransfer* (procedure) model the programmer must always protect shared data with monitors; in the message model he is guaranteed serialization because messages are explicitly received.

The reader may well ask why nontransparent invocation schemes are considered. The primary reason is applications. For example, Ball's remote procedure mechanism [1] serializes calls because processes are not cheap and because the Alto application is stream-oriented and will neither benefit from the extra concurrency nor deadlock in its absence. Other invocation schemes between those offering full concurrency and complete serialization are easily imaginable. It is possible to serialize (or parallelize) a procedure, module, resource, or entire machine by choosing an invocation scheme and applying it to the corresponding unit—procedure, module, resource, or machine. Each of these schemes may have a place depending on the application programmer's needs and on the programming environment's semantics for transfers and concurrency. While alternative invocation schemes may be attractive in special situations, the thesis uses *only* concurrent invocation.

4.1.2 Binding and Configurations

In a distributed system, *binding* is the process of naming and connecting all of the modules of a distributed program. A *configuration* is a set of modules that cooperate to implement a program. The main goal of remote binding is making the specification of remote configurations syntactically and semantically convenient, if not transparent. This convenience can be measured by watching RPC users at work: It should be possible for them to configure, bind, load, and start the modules of a distributed system just as easily as they can perform the same steps within their local environment.

4.1.2.1 The Spectrum of Binding Times

There is a broad spectrum of binding times in both conventional and distributed systems. Early or *static* binding is usually very efficient, but not very flexible; late or *dynamic* binding is extremely flexible, but often causes a loss of space or time performance. There are four common binding times, listed here in order of increasing flexibility:

> *Compile time.* Binding decisions are made when modules are compiled.
>
> *Link time.* Binding decisions are made when modules are combined into predefined configurations but before they have begun executing. Compile time and link time binding are often called *declarative* binding because the binding relationships are written down, or declared, in advance.
>
> *Static runtime.* Binding decisions are made just before the modules (or configurations) first execute and communicate.
>
> *Dynamic runtime.* Binding decisions are made at runtime and can be changed at will. Dynamic runtime binding is often just called *dynamic* binding.

4.1.2.2 Binding Interfaces and Components

While most of the preceding semantic discussion concentrated on procedures, real programs communicate with more than procedure calls. Fortunately, the notion of an interface—a compilable, abstract specification of a module's behavior—embraces other control structures quite easily.

Interface *components* are instances of certain types, typically programs, procedures, coroutines, exceptions, variables, and types. The interface components exported by a module are those abstract operations and data that the module makes available to users of the module's abstraction [45,62,91]. Other modules can implement the same abstraction differently: these modules export different instances of the same interface components. The local binder's job is to link together the users of an interface (its importing clients) with its implementors (its exporters). The remote binder operates similarly: binding control transfer components—programs, procedures, coroutines, exceptions, and so forth—is straightforward. Since these components are all implemented by the *RTransfer* primitive (section 2.1.3.1), the binder logically just assigns *inports* and *outports* according to the programmer's static or dynamic configuration specification. Variables and types, which are outside the definition of RPC, cannot be handled with *RTransfer*. Global variables, however, are so frequently used for intermodule communication that it is worthwhile considering their problems.

4.1.2.3 Remote Variables

Whether or not RPC schemes should furnish transparent access to remote data is an open question. The operational DPS and Cages mechanisms discussed in chapter 3 support remote data, but Guardians and the Downloader do not. One of the arguments advanced in favor of global variables is efficiency. Presumably, the cost of a memory reference is much less than the cost of a procedure call to encapsulate a read or write of the same value. For remote global variables, however, this cost argument is largely vacuous because of communication expenses.

In light of current abstraction practice, it is *strongly* recommended that neither local nor remote global variables be used. Instead, they should be encapsulated in a (remote) procedure interface, or inside an object with a (remote) handle (section 4.1.4.2). Replication and sharing of global variables become problems that are best solved with (remote) procedure calls to monitors anyway.

In systems where remote variables *do* have priority over methodology, however, a uniform address space is the best approach. In local network environments, Spector's remote read/write memory work demonstrates good performance for remote variables. In shared memory multiprocessor environments, even very loosely coupled systems such as Cm* offer rapid and uniform addressing that inherently provides "remote" variables.

In systems with remote variables, caution must be exercised with special datatypes such as condition variables and monitor locks (semaphores). Making remote instances of these variables can lead to situations where monitored (synchronized) abstract objects are implemented or controlled on multiple nodes. Whether or not such *language-level* resources should be spread across machines is a topic of controversy [69].

4.1.2.4 Authentication and Authorization

Transparent RPC intentionally blurs the distinction between local and remote calls. In a real distributed system, however, remote calls between autonomous nodes will often be subject to access controls. For example, many resources provided by servers are controlled by accounting and protection policies. Enforcement of these policies involves two procedures which can affect the remote binding process. The first is exactly identifying the party requesting service: Authentication procedures are used for this, and Needham and Schroeder [63] have successfully attacked this problem for internetwork environments. The second procedure is deciding if the requesting party is authorized to access the service: This access problem is solved by most multiuser operating systems; the complexity of the authorization mechanism generally depends on the nature of the resource. A file server is a good example of a controlled service requiring both these techniques to institute its protection policies.

The role of the remote binder in these policy procedures is as a language-level agent connecting two potentially suspicious parties. Here are two opposing models for this role:

> *Democracy Model.* The binder should ignore access problems and let the two connected parties negotiate after they have been properly bound. This approach is sound when a host machine's overhead for binding is negligible. If the overhead is too great, the binder may blindly commit too many of the server's resources to clients that will eventually be turned away. This degrades the service of legitimate clients and violates the server's autonomy.
>
> *Autonomy Model.* The binder should handle first-level authorization checking. This approach is useful when a server wishes to completely exclude large classes of clients. It also requires nonbinding machinery in the binder. Of course, the binder's typechecking facilities can perform this authorization checking in the extreme case where types are carefully distributed capabilities and not publicly available definitions.

The anarchy of the binder itself is the critical issue here. In the democratic model, the binder is an extremely independent part of the system—like the judicial branch of government. It has the power to create and connect processes on any machines whether those machines want them or not. Clients are free to break unions only after they have been established. In the autonomous model, the binder is a subservient element in the environment—like the press in many countries. Nodes can throttle all outside communication simply by squeezing the binder.

The choice between these models and others will depend on characteristics of the actual environment. Communication speeds, binding complexity, and mutual cooperation will be strong factors. Authorization policies will change as distributed systems evolve. At this point in RPC development, having remote binders provide at least the bare-bones mechanisms needed to implement these policies is more important than enforcing any given policy. A breachless typesafe environment supplies this skeleton in the democratic model.

4.1.2.5 Binding Heterogeneous Configurations

Binding together modules from different languages or for different machines requires that the issue of type translation be resolved. The desire for easy reconfigurability in the presence of crashes postpones translation decisions until this time; earlier, static decisions often restrict binding choices so severely that uniformity is lost.

It is likely that a mixed language environment will have not one but multiple binders. Because each binder and language will have different power, a common denominator of facilities will evolve. Rather than restrict two powerful systems to the potentially weak common facilities, *negotiation* is suggested. In this scheme, the binders communicate between themselves to determine the highest level of support each can mutually offer the other. Actual binding then uses these less primitive mechanisms. A conceptually straightforward approach is for the binder to insert appropriate type translation filters between the heterogeneous calls that it binds together. Other approaches are discussed in section 4.1.3.3. Negotiation in the general context of internet communication is well covered by Sproull and Cohen [82].

4.1.2.6 Load Control

Load control is a problem similar to authorization control: Using a declarative configuration language to write binding requests for distributed services can be unrealistic when the eventual (dynamic) binding is to heavily loaded servers. Even the most cooperative servers may have to turn away potential clients—because the server is full, because it is broken, or because it has a high-priority task to perform. Denial of service because of load control, however, is rarely as absolute as denial due to improper authorization. For instance, a server may well respond to a request for service by asking the requestor to retry in a few minutes. An important goal here is that a service never appear dead due to overloading. It should always have the small reserve needed to inform clients that it is (almost) saturated.

Because service denial is likely in realistic distributed systems, remote binding semantics are different from local binding semantics when the local host's full resources are assumed to be continuously available. This nontransparent behavior can be overcome only by repeated dynamic binding attempts with overloaded servers. A sophisticated remote binder could, however, perform these attempts automatically.

It is interesting to observe that load control in remote procedure schemes is analogous to flow control in message-passing systems.

4.1.3 Typechecking

Extending single-machine typechecking into an internet is conceptually straightforward. The real challenge is presented by the related efficiency and validity problems.

4.1.3.1 The Flexibility Spectrum

There is a flexibility versus efficiency spectrum for typechecking. At the top end, carrying around the symbol table of each exported interface is possible. This allows full checking to occur at runtime at the cost of both space for the table and time to do the check. At the other end of the spectrum is a unique ID approach like Mesa's. Here each interface has a unique identifier indicating when it was compiled. Mesa defines the typesafe use of an interface to require that all of its importers be compiled with exactly the same interface—that is, the one with exactly the same ID.

This spectrum of tradeoffs is just that of Lisp's EQUAL test versus EQ test [88]. Performing arbitrary checking at runtime requires an EQUAL test because the checker must walk the trees of all user-defined types and compare the base types attached to each terminal node for equality. Flattening these trees ahead of time provides some improvement, but does not approach Mesa's EQ method where a single comparison of timestamps validates an entire interface of components—types, procedures, and data.

This spectrum also reflects the delay of binding decisions as discussed in section 4.1.2.1. Delayed binding gives increased generality, typically at the price of runtime checking. Early binding, on the other hand, greatly reduces generality as well as the cost of the binding. Mesa performs intermodule typechecking during binding simply by checking that the interface IDs match. This scheme, however, has the bad property that if the definition of an interface is recompiled just to change an obscure component, then every program using that interface must be recompiled and rebound. This recompilation problem is severe in a single-machine environment; it would be untenable in a distributed system if the often-changing interface were that of a public file server. One solution to this difficulty is through an intermediate binding approach where each *component*—rather than each entire interface—has a unique ID. At the cost of carrying these additional IDs around, the RPC environment can still perform rapid EQ checks at runtime. Recompilation is necessary only for those modules that actually use the changed interface component, rather than for all modules that use the interface.

The style of remote typechecking adopted for a given environment can be expected to influence the communication style. For example, supporting dynamically typed languages like some Lisp and Algol68 dialects requires the full EQUAL typecheck. The cost of this checking is great enough that the overhead of connections might be justified for remote calls. After opening a typesafe stream and validating the procedure types once, argument and result records can be transmitted through the stream without further checking. In untyped languages such as Bcpl, on the other hand, the state information of connections may not be required: it may be cheaper just to send the procedure name along with the parameters each time. The same might apply to Mesa, where the unique ID would be sent along with the procedure name for fast EQ typechecking.

Communication decisions will also be influenced by the frequency of remote calls. A single call to a time server, for example, does not have the same performance impact as calls to a file server in an inner loop of a data base program. The underlying remote procedure implementation may want to negotiate such decisions on the fly. For example, it might perform all initial calls in a reliable but connectionless manner and then switch to connections if the same call repeats sufficiently often. Such decisions should always be hidden from RPC clients because they are internal to the RPC implementation. Transparent remote procedure mechanisms give a uniform view of the distributed world that makes these efficiency considerations invisible.

4.1.3.2 Type Authentication and Validation

Whatever typechecking method is used, it is easily breached if malicious clients have access to the communications medium. This is certainly the case with most internetworks. A knowledgeable intruder can always forge the correct unique IDs, symbol table entries, or whatever type information accompanies remote procedure calls to guarantee consistency of access. To be certain about type safety in an internetwork stronger *type authentication* is needed. Here the type information is encrypted and authenticated by the typechecking machinery. This authentication layer is of course independent of the typechecking itself, and Needham and Schroeder [63] have discussed some appropriate internet techniques. If the type information is sealed along with corresponding instances of values, the resulting objects begin to resemble capabilities.

Another approach to type validation is supplying a *legality procedure* to each client of a type. A legality procedure takes as its argument a bitstring purporting to be a valid value of the type and returns an indication of whether it is valid or not. Of course, this is *not* the same as type authentication, nor is it an exclusive problem of remote procedure call: a RED INTEGER's bitstring is probably the same as a BLUE INTEGER's, and a legality procedure will doubtlessly tell us that both are INTEGERS. Legality procedures can do is check that a client has not damaged—accidentally or otherwise—a value of a type to the extent that it will compromise the implementor of the type. Viewed in this way, a legality procedure guarantees the *consistency* of an object whereas type authentication guarantees the *structure* of an object. These are independent notions with separate strengths and weaknesses. The consistency problem is especially troublesome in heterogeneous language environments where strong typechecking is impossible.

Because validating the consistency of a type requires executing a legality procedure to check each value, the scheme is not usually practical for performance reasons. In spite of this, the availability of legality procedures is often useful because they can be called when unknown errors corrupt the system. Legality procedures—even when explicitly written by users—can be real timesavers during memory smashes and other hard-to-diagnose situations.

4.1.3.3 Type Translation

Type translation for heterogeneous machines and languages is a problem beyond remote procedure call. This is not a quick dismissal of the importance of bridging dissimilar environments. These translation issues must be on the mind of any remote procedure designer; otherwise, retrofitting an existing RPC scheme into a heterogeneous environment could be impossibly difficult. (See Cohen's enlightening discussion [16] of these principles.)

Fortunately, there is a body of existing work on translation schemes. Levine's thesis [55] evaluates a number of interprocess communication (IPC) schemes that translate built-in types such as integers, characters, and strings. His conclusion is that a standard intermediate representation is best. Rashid's IPC mechanism [72] uses this approach to implement interlanguage type translation. Rashid also includes extensions that handle user-defined types and record structures. Interlanguage record translation work in the setting of network operating systems is described by Kimbleton, Wood, and Fitzgerald [42]. Wallis [93] discusses translation between user-defined and external representations in a portable programming language. The Clu-oriented type translation schemes of Herlihy [33] were mentioned in section 3.2.7. By going a step further and translating abstract types as well as built-in ones, Herlihy's scheme permits *intra*language translation of different concrete representations of the same abstract type.

4.1.3.4 Versions and Persistent Values

An important problem related to typechecking and type translation is the treatment of persistent values. A *persistent value* is a long-lived typesafe object that resides on secondary or tertiary storage (e.g., disk or tape). The difficulties with persistent values are twofold:

> Guaranteeing typesafety when the object is outside the domain of the implementing environment, e.g., outside primary memory because of programmer I/O;
>
> Successfully retrieving values whose representations are obsolete, e.g., have been superseded by new representations of the object's abstract type.

Persistent values are mentioned in the context of remote procedures because they bear on the problem of software version changes. Since different versions of a system can have different interfaces, version changes are a potential typechecking problem for remote procedure call.

Consider, for example, a distributed transport system for electronic mail where each user has a personal computer running a distinct copy of the mail program [6]. These mail processing programs communicate to mail servers—where mailboxes, distribution lists, and so forth are stored—via RPC. New versions of the mail program are released periodically, and in a user community numbering in

the hundreds or thousands it is highly likely that many versions of the program will be in use simultaneously. Imagine that some release adopts a new message format. If messages are persistent values passed as parameters, then either the RPC mechanism or the mail servers must cope with translating old messages (values) into new ones. Here are the three cases:

> The old mail programs cannot do this because they have no knowledge of the new representation.
>
> The RPC mechanism can do it if each version of the mail program translates its messages into a standard representation such as Herlihy's *xrep*. This works until the standard representation changes.
>
> The mail server can do it as long as it is programmed to handle all versions of messages. Of course, it must be *persistently* prepared to do this.

The problems of persistent values and version changes are not further addressed by the thesis, but real systems—and their RPC mechanisms—must consider them.

4.1.4 Parameter Functionality

The earlier discussion of parameter functionality sidestepped several important problems. Foremost among these were the handling of address-containing parameters, that is, reference, pointer, and procedure parameters.

4.1.4.1 VAR Parameters

Semantically, passing an argument by reference is equivalent to replacing the formal parameter with a (contextually dereferenced) pointer to the actual argument. (This actual argument must of course be a variable and not a constant or expression.) For a remote call to a disjoint address space, passing an address valid in the local address space does not usually work and is often catastrophic. Fortunately, for calls with no aliasing, call-by-reference is equivalent to call-by-value-result [28]. (Actually, there can be differences during exception handling if an abstraction raising an exception is not careful to restore any invariants that apply to its parameters.) For example, axiomatic Pascal permits its VAR parameters to be implemented by either method because it leaves aliasing behavior undefined. Remote procedure call can easily support value-result VAR parameters by transmitting back into the callee's VAR arguments the final values of the corresponding formal arguments from the site of the remote invocation.

This value-result scheme works equally well when a chain of VAR parameters extends across machines, even back to the original one. The inherent nesting of procedure calls guarantees that chains have their results copied back properly, even for recursive calls. There are potential synchronization problems if a VAR parameter is shared between processes, but this is just another case of aliasing. These same synchronization difficulties exist for sharing among local procedures, too, so the programmer must exercise identical cautions for both local and remote calls. Another potential problem is with datatypes that do not admit full assignment, e.g., semaphores. In the absence of aliasing—including unsynchronized sharing among processes—variables of these types

still work correctly when called by value-result as long as only the assignable parts of the type are copied back. Of course, these special types and their operations are probably better cast in the object model discussed below.

The copy-back scheme provides uniform local and remote semantics for value-result VAR parameters. The lack of call-by-reference will not be missed in view of the current trend toward eliminating aliasing.

4.1.4.2 Pointer Parameters

Pointers offer harder problems than VAR parameters. Fortunately, these subproblems can be attacked independently by examining the three ways that pointer parameters are used:

> *Efficiency.* Pointers are used for efficiency in passing records and other bulky values by reference. This is especially true in languages like Mesa where all parameters are passed by value.
>
> *Object handles.* Pointers are used as handles (capabilities) to objects (abstract aggregates of operations and data).
>
> *List structures.* Pointers are used to pass the roots of honest list structures, i.e., graph structures containing other pointers. Included in this case are parameters that have embedded pointers, e.g., arrays of pointers, records with pointers, and so forth.

Efficiency. The solution to the efficiency problem is easy: Just make the programmer tell the truth and use VAR parameters instead of pointers. Of course, this *will* be inefficient when only one component of a record or a few elements of an array are being changed. By leaving the implementation of VAR in the hands of the language instead of the programmer, however, using call-by-reference locally and call-by-value-result remotely is possible.

This use of VAR assumes that the argument will always be modified. Since VAR is being used to replace this whole class of pointer usage, we must also consider the case of readonly references, i.e., where a pointer is used just to prevent the inefficiency of call-by-value. To handle this situation consider a VAR-like extension to formal parameters called VAL (it could also be called READONLY VAR). The semantics of VAL is call-by-value, but a smart compiler can use a reference to implement VAL in local procedures if the programmer does not modify the formal argument. An even smarter compiler can make VAL unnecessary by deducing which instances of VAR never modify their referent and by substituting an implicit VAL. Of course, this is not a decidable problem, but some cases can be handled quite well this way.

The introduction of VAR and VAL to solve pointer efficiency problems gives the programmer sufficient expressive power to state his desires rather than force a realization of them. These language features prevent him from being so "efficient" that he will find himself drastically rewriting his programs for remote use rather than watching them distribute gracefully. This is precisely what a transparent RPC mechanism tries to avoid, even at the cost of some additional overhead.

Object Handles. The solution to the handle problem is more difficult. A handle usually names some abstract object which has an associated set of abstract operations. If a handle is used as a remote argument, then the handle must not be dereferenced except in the context of its associated object. This is somewhat analogous to call-by-name in Algol60, where every operation on a call-by-name formal parameter (e.g., the handle) causes the actual parameter to be reevaluated in the environment of the caller (e.g., the object's context). For handles in remote address spaces, this means that all attempts to access the object through the handle must be converted into remote calls to the object's context. Fortunately, most languages already cause this trapping by requiring all object references to occur as operations (procedure calls) qualified by the handle. These object procedure calls can be easily converted into the proper remote calls by storing a unique identification of the implementing context (say, (*host, process, module*)) along with the object pointer itself.

Some high-level languages, for example Clu [56], are completely object oriented and have no language-level notion of pointer. This elegant approach is good for the programmer and—when objects are not immutable—gives implementors latitude to find the implementations best suited for local and remote use.

Other high-level languages, however, do not provide direct object support but allow programmers to program them with idioms. Mesa is one such language, and a possible Mesa implementation of objects is:

```
Handle: TYPE = PRIVATE RECORD [
        object: POINTER TO Object,
        context: ContextIdentifier ];

Object: TYPE = PRIVATE RECORD [
        operations: PUBLIC READONLY POINTER TO Operations,
        data: RECORD [ ... ] ];

Operations: TYPE = RECORD [
        Read: PROCEDURE [...] RETURNS [...],
        ...
        Delete: PROCEDURE [...] ].
```

An entirely different approach to handling objects is to actually transmit the object to the remote client. This is especially feasible when the actual code of the operations' procedures is sent along too. If the abstract operations are called frequently and their execution time is short compared with the time of a remote call then large gains in performance are possible. Of course, objects moved in this way must be transmitted in their entirety so that they contain no invalid pointers. Dynamic migration of code between machines is not really within the definition of remote procedure call, and there are many unstated problems here, but such migration may be attractive for some homogeneous language systems.

List Structures. Solutions to the general list structure problem are hard because of the computation costs. The transmission of graph structures usually requires extensive pointer chasing and storage allocation unless compact list encodings such as those proposed by Bobrow and Clark

[7] are used in the user's or system's implementation. Furthermore, if parts of a graph are shared, sharing in subgraphs can disappear when a remotely modified subgraph is copied back and reallocated in the local address space—that is, when the parameter is a VAR list. Handling shared list parameters in this way violates graph structure semantics and is not transparent.

Some uses of list and pointer parameters, on the other hand, are for the convenience of implementation and not for sharing. Two trivial examples are Mesa's STRING and ARRAY DESCRIPTOR types, both of which use pointers in their underlying representations (specific examples appear in chapter 5). Handling acyclic call-by-value lists in this nonshared context is reasonable, although keeping an eye on the computation costs is wise, as mentioned above. The main difficulty with nonshared list parameters for RPC implementations is identifying the shared and nonshared cases so that they can be handled appropriately—with an error or a call. The programmer's explicit use of VAL can declare the latter.

The transmission of list structures in parameters does not receive much further attention in this dissertation because Herlihy's thesis shows how both cyclic and acyclic graphs can be transmitted.

Remote access functions are an alternative approach to list parameters. The scheme gives lists objectlike properties since callers do not pass entire lists. Rather, only a list's root (handle) and appropriate list traversal and modification *access function* parameters are passed as arguments to the remote callee. These functions are then used by the callee to cause remote manipulation of the list back in its host machine. With this approach, the list is treated as an abstract type whose remote access operations are permanently bound to it. The scheme is expensive if the access functions are as trivial as, say, CAR and CDR, which are usually primitive operations on both host machines. (Thus the desire to pass only the list itself, above, is a solution where the list's nodes are abstract objects implemented on both machines.) The conclusion is that the abstract operations of a list type should be sufficiently powerful that the overhead of remote calls is acceptable. This access function approach is an obvious way to handle the shared graph parameters that were rejected by the VAL scheme.

A different solution to the remote list problem is again provided by Spector's work and the Cm* system. Their remote memory operations are fast enough that making CAR and CDR be *local* operations using *remote* addresses is feasible.

4.1.4.3 Procedure Parameters

The potential problems with procedure—and other control transfer—parameters are illustrated in the following Mesa example.

EnumerationProcedure: TYPE = PROCEDURE [...] RETURNS [...];
TakesOne: PROCEDURE [*proc*: *EnumerationProcedure*] = ...;
GivesOne: PROCEDURE [*reverseOrder*: BOOLEAN] RETURNS [*proc*: *EnumerationProcedure*] = ...;

UsesBoth: PROCEDURE = BEGIN *TakesOne*[*GivesOne*[TRUE]] END.

The main issue is that both *TakesOne* and *GivesOne* can have remote procedure (or program, exception, and so forth) parameters. In the earlier discussion of binding it was (probably) assumed that all remote procedures appear in an interface that could be specially handled—if necessary—by a remote binder. In this example, however, the *EnumerationProcedure* can be either local or remote and this knowledge must be kept from the programmer.

The problem is to make remote procedures full values in the language just as local procedures are (in Bcpl and Mesa, anyway; not in Ada). The solution, which is part of extending *RTransfer*, requires three things:

> Calls on procedure values that are bound to remote procedures must be invoked through the RPC runtime environment.
>
> The RPC runtime environment must maintain a mapping of local values that are remote to actual remote procedures. This is a mapping into *RTransfer*'s fully specified *ports*.
>
> There must be a conversion from local procedure values to remote procedure values. This conversion will establish the above mapping and will be applied whenever procedures are initially passed as remote parameters. It will be used, for example, when *GivesOne* is a remote procedure that returns a remote procedure value to *TakeOne*.

Notice that all of this information is available at compile time. For an *EnumerationProcedure* to ever be remote, it must be a formal parameter of *some* top-level remote procedure—for example, as a formal argument of *TakesOne* or as a formal result of *GivesOne*, or in their transitive closure up to some remote interface. The compiler can therefore locate the local-to-remote transition of any procedure value and perform any necessary local-to-remote conversions. This argument extends to all other control transfer types as well.

4.1.5 Concurrency Control and Exception Handling

Section 2.2.6 stated that remote procedure mechanisms must have independent concurrency control and exception handling to meet the goal of transparent semantics. This stipulation requires that the host programming language have good facilities in these critical areas. We now consider some particularly desirable features.

4.1.5.1 Concurrency

To achieve parallelism in distributed programs, the programmer must use the process and synchronization tools his programming language provides. In Mesa, for example, if a remote call can proceed in parallel with local computation then the programmer should FORK the remote call and JOIN the results later. Similarly, in Ada he would create a TASK and in Algol68 he would use COBEGIN and COEND. If a remote call accesses shared data, then to prevent synchronization problems with local tasks concurrently using the same data, the programmer must encapsulate that data in a MONITOR or TASK—just as he would if two local processes were sharing it. In general, having uniform local and remote semantics forces consistent use of existing concurrency machinery. Furthermore, making the programmer explicitly create processes to handle remote parallelism encourages him to be aware of synchronization problems with those processes.

4.1.5.2 Exceptions

To handle program- and system-generated errors in a distributed system, the programmer must use the exception-handling tools in the language. Ada and Mesa, for example, have good mechanisms for handling synchronous intraprocess exceptions. Mesa's scheme is more powerful because it allows exceptions to take parameters (like procedures) and to be resumed. For interprocess exceptions, unfortunately, neither language gives much assistance. Mesa has no scheme for passing exceptions between processes; Ada's mechanism is better but is limited to passing FAILURE and ABORT signals.

Levin's thesis [54] talks about the importance of multiprocess exception mechanisms and discusses how to implement them in procedural languages. The need for asynchronous exceptions between processes has long been recognized and implemented in the message-passing world: Plits, Medusa, and RIG [23,68,51] all provide some method of interprocess exception. The RIG implementors, in particular, found this capability essential for designing reliable systems.

4.1.5.3 Aborts

Aborting a process—that is, forcibly halting its execution and destroying its computation—is required by RPC implementations in order to exterminate orphans. This mechanism, which is a part of the concurrency machinery, is needed by robust applications as well. The ability of a process to *finalize* itself is a vital ingredient here: Aborted processes must be given a chance to terminate cleanly. This means letting them carefully withdraw from any monitors they may have entered so that the proper invariants—including monitor locks—can be restored before death. Both Ada and Mesa have respectable process aborting mechanisms; each interjects a special asynchronous exception that the dying process can catch and use to perform cleanup. Of course, a truly renegade aborted process may try to regain control during finalization. Thus an absolute mechanism that bypasses finalization is needed too. Ada's ABORT is just this mechanism; Mesa has no equivalent.

Machinery for aborting processes and forcing asynchronous exceptions may seem somewhat violent to the gentle programmer of sequential abstractions. But realistic remote procedure environments and, indeed, realistic applications will in fact require this machinery to provide robust services and exploit the full performance of distributed systems.

4.1.5.4 Timeouts

Timeouts are important for reliability and responsiveness in distributed systems. Unfortunately, timeouts are often abused by offering them as a general catch-all error that conveys no real information to users. For example, assume A sends a *call* message to B with a ten-second timeout. In the absence of any further interpretation, if the timeout expires then A does not know if the message was lost, delayed, received, discarded, executed, returned, or anything else. Of course, low-level transport mechanisms usually deal with most of these uncertainties. The point is that timeouts should *not* be used for general indications of trouble. Rather, a specific exception should be generated that conveys the nature of the problem: *NetworkPartitioned*, *CallRejected*,

CallStartedButIncomplete, and so forth. I am not advocating that these particular exceptions are suitable for clients to handle; I am advocating that whatever abstract level intercepts exceptions should have specific information made available to it—not just *timeout*—for making intelligent decisions about error-handling alternatives.

At the application level, remote procedures should be used with timeouts for just one purpose: performance. For example, "I've waited two minutes for this operation and my human user is getting frustrated—I'll abort this one and try somewhere else." In this case the programmer decides to abandon the computation for performance reasons. He is not using the timeout to look for errors; he lets the RPC mechanism report these with exceptions.

4.2 The Pleasant Issues

Solutions to the problems posed by the essential issues are required for transparent RPC, but procedure call—local or remote—is not a panacea for distributed programming. Issues of efficiency, reliability, heterogeneity, and debuggability are also of concern. We now look at these pleasant issues, so called because their solutions make RPC a pleasant as well as transparent programming tool.

4.2.1 Good Performance

The struggle between elegant program structure and good program performance is unending. The fundamental goal of transparent syntax and semantics allows programmers to write distributed programs with the same elegance as local programs. But the price of this transparent structure, if too great, will force RPC to the bottom of the programmer's toolbox: a programmer who needs a balanced communication primitive is unlikely to use shiny-but-leaden RPC to build a distributed system.

The search for good performance within the framework of the essential issues, an important part of this work, is the subject of chapter 6.

4.2.2 Sound Remote Interface Design

Just as the methodology of local procedure call has changed over the years, so the style of remote procedure call will change and develop too. In one sense this is contrary to the notion of transparency. In another, realistic programs will often know which of their modules and resources are accessed remotely—or at least partially so. This knowledge may be motivated by anything—a desire to increase parallelism, a need to exploit the speed or efficiency of underlying communications, or a requirement to provide atomicity in the presence of extremely long delays.

Most of this decision making will fall on the remote interface designer, for the division and assignment of functions to remote interfaces determines the overall structure of a distributed system. Fortunately, these decisions are similar to the ones made in partitioning the work of any distributed

system, whether it uses RPC or not. Thus basic interface decisions should continue to be guided by non–communication primitive criteria, although the resolution of design details may use knowledge of specific primitives. This is a crucial point, for remote interface designers must not be seduced by the transparency of RPC. Using the descriptive power of remote procedures to design a wonderful interface is acceptable *only* if the interface satisfies the primary design criteria: A clean interface can still be an egregious performance catastrophe if the use and distribution of its functions are not first taken into account.

The RPC-specific details and guidelines of remote interface design can be learned only through actual experience with operational RPC mechanisms. While this chapter has covered many of these details in a general way, there is insufficient experience to draw firm conclusions. Remote interface design is revisited briefly in chapter 6.

4.2.3 Atomic Transactions

Maintaining data consistency and system reliability in the face of machine crashes is a topic of current research. One emerging approach, well exemplified by Lampson and Sturgis [46,48], is to provide *transactions* of *atomic operations*. In this scheme, programmers group restartable atomic operations together into transactions that are guaranteed to have the atomic property: if the transaction *commits* then all the operations happen; otherwise, if it *aborts*, none of them happens. After a crash, recovery is initiated simply by continuing the system from the last checkpoint. The intervening work of an uncommitted transaction is guaranteed to have had no side effects, and a committed transaction is always completed. Of course, to use this and similar schemes effectively, the application designer must structure his system to consist of operations that can be grouped into atomic actions. A methodology is emerging for this.

There are two schools of thought on the relationship between remote procedures and transactions: one believes that all remote calls should be atomic operations; the other maintains that remote calls should be nonatomic, like local procedures. Liskov's Guardian work has considered these two positions carefully and concludes that atomic RPC is necessary. She justifies this position with the extremely robust applications that Guardians address: airline reservation systems, bank accounting systems, and so forth. This thesis, however, covers a broader spectrum of applications and takes the opposite position for two reasons: First, the policy and expense of making each remote call atomic is too great a burden for many clients. Second, remote procedures and atomicity are basically independent notions that require more investigation and experience before being tied together. This completely acknowledges the importance of transactions, but takes the stand that remote procedures with the same power as local procedures should be studied before stronger semantics are added.

4.2.4 Respect for Autonomy

Respecting the autonomy of individual nodes is a basic requirement in many distributed systems. The homogeneous and cooperative setting in which the essential properties are cast largely

overlooks this consideration. The previous sections on binding and typechecking are exceptions to this; they explore some of the authentication, authorization, and load control issues that must be resolved when desire for "transparent" binding encounters the realities of autonomous environments. Turn back to those sections for more information.

4.2.5 Type Translation

Exploring the essential issues in a homogeneous environment also skips over important problems of language and machine heterogeneity. The type translation issues of binding, typechecking, and parameter functionality are covered in these same sections. Refer to those sections, and to Herlihy's and Levine's theses [33,55], for more details.

4.2.6 Remote Debugging

Even if remote procedure call works perfectly, the programs using it will not. Facilities for debugging multimachine configurations of modules are an essential part of a distributed programming environment. These debugging facilities must give a user on one machine the ability to control debugging on all of the others. This requires the means to start, interrupt, redirect, and abort the execution of remote procedures and their processes. Similarly, the ability to examine and modify data, set breakpoints, and perform all the operations of a local debugger must be provided. On the other hand, certain cautions are called for. In a multiple process environment these debugging actions must not disturb other processes. This is especially important for servers and other autonomous nodes that manage important resources for other programs. Attempts by a random debugger—that is, user—to "grab control" must therefore be carefully mediated to prevent unwanted intervention.

Some work on remote debugging has been done by Arpa's Network Working Group [70]. Two basic approaches are possible: The first is to include, as a *nub* in each machine, a simple interface that starts and stops the processor and reads and writes memory. If the debugger uses only this interface to access a machine—including itself—then remote debugging is readily implemented with remote procedure call. This scheme can often be retrofitted into single-machine debuggers quite easily. The remote Bcpl debugger on the Alto, for instance, was implemented with the nub approach [89].

A second approach is to have separate, complete instances of the debugger in each machine. These debuggers then communicate between themselves at a much higher level than read and write memory when a user's command crosses machine boundaries. This approach decentralizes the debugger's control and requires less internet traffic. It also requires very careful separation of the debugger's user interface functions from the actual debugging primitives (i.e., those likely to be called as remote procedures). In this sense it is just an extension of the first method with a higher-level nub.

Debugging programs of cooperating modules written in different languages presents new problems. Unfortunately, the most frequent approach is to use a "least common denominator"

assembly language debugger. As poor as such debuggers are, however, the presence of heterogeneous machines can make even this approach impossible. Suitable layers of abstract machines must be defined, and corresponding debuggers written, to overcome these problems. Otherwise, the unattractive alternative of simultaneous, independent debugging on many machines presents itself.

4.3 Summary of Ideal Properties

This chapter discusses a number of remote procedure issues. The essential issues deal directly with the problem of providing transparent syntax and semantics in a homogeneous language. The pleasant issues address other problems such as efficiency, autonomy, and heterogeneity. Each issue is now resolved into a brief statement of ideal behavior. These statements are one set of *essential* and *pleasant properties* for transparent RPC mechanisms.

4.3.1 Essential Properties

These five properties are essential to a remote procedure mechanism that is fully integrated into a homogeneous programming language and that provides transparent local and remote procedure semantics.

> *Uniform call semantics.* In the absence of crashes, remote procedures must have the exactly-once semantics of local procedures. In the presence of crashes, remote procedures must have the last-one semantics of crashing local procedures. Atomic procedure call is too expensive to be the standard RPC mechanism; the cheapest uniform semantics is obtained by automatically exterminating orphans after crashes. Remote calls must be invoked via concurrent invocation, not serial invocation.
>
> *Powerful binding and configuration.* Programming language facilities for binding—i.e., specifying, linking, and loading—configurations of separately compiled modules must be extended to handle modules that reside and execute on remote machines. The binder must permit flexible machine naming, and binding facilities must be available both declaratively and dynamically.
>
> *Strong typechecking.* The type calculus of the standard programming environment must be fully applied over the distributed system. Any operation that causes a type violation in the local environment must cause the same violation in a remote environment.
>
> *Excellent parameter functionality.* Parameters to remote procedures must be passed by value-result. Nearly all language- and user-defined datatypes allowed as parameters of local procedures must be valid as parameters of remote procedures. This includes many traditional uses of pointers, but excludes list structures: while Herlihy and others show how to pass shared structures in parameters, automatic graph handling is not suggested here. Global variables are disallowed as well.
>
> *Standard concurrency control and exception handling.* The standard parallel-processing and exception-handling facilities of the language must interact identically with local and remote procedures. This may present performance problems for languages with poor (or no) concurrency control. It may also make error handling more troublesome for languages without good exception mechanisms, especially those without interprocess exception mechanisms.

4.3.2 Pleasant Properties

These six properties are not fundamental for an RPC *language* mechanism but they do make any proposed scheme much more palatable for real distributed programmers.

Good performance of remote calls. To be a realistic tool, remote procedures must be comparable in cost to the application-tuned protocols they are intended to replace. It is well known that, in the limit, efficiency-minded clients will always trade abstraction and elegance for direct manipulation of bits.

Sound remote interface design. Remote interface designers must always evaluate their work in light of the increased cost of remote procedures. While transparency is wonderful for implementors and users, the interface designer must be careful to distribute functions cost-effectively.

Atomic transactions. Robust applications demanding high reliability must use independent transaction mechanisms. This is vital because the recommended RPC semantics are not atomic in the presence of crashes.

Respect for autonomy. The binder must respect both the autonomy and the capabilities of its host machine. This requires policy decisions based on the degree of cooperation in the environment.

Type translation. The compiler and binder should cooperate to perform automatic type translations in heterogeneous language and machine environments. Runtime negotiation can optimize performance.

Remote debugging. A language-level debugger that deals with multimachine configurations is vital for real programmers. Problems of heterogeneity and autonomy add complexity.

Nuku Hiva—Marquesas
8°54' S 140°08' W
The first Polynesians sail for Havai'i-of-the-North, A.D. 750

5

Design Approaches for a Transparent Mechanism

Three implementation approaches for remote procedure mechanisms were proposed in chapter 2: the *RTransfer* primitive, compiled *RTransfers*, and source-level stubs. Chapter 4 showed that for any of these approaches to define a *transparent* mechanism, the approach must satisfy the five essential properties. Fully satisfying these properties, however, requires more than just a basic mechanism. In particular, algorithms are required to exterminate orphans after crashes, and a distributed binder is required to interconnect modules so that they can communicate with RPC.

In response to these requirements, this chapter develops *Emissary*, a design for a transparent RPC mechanism that satisfies all of the essential properties. Emissary is also designed for high efficiency; chapter 6 shows that Emissary satisfies the good performance property in addition to the essential properties.

5.1 Emissary's Semantics

Emissary is a transparent RPC mechanism that satisfies the five essential properties defined in chapter 4. Since Emissary's semantics are completely characterized by the essential properties, it is well to restate them briefly here. Because Emissary's design is largely language-independent within the general context of procedural languages, concrete details of (for example) typechecking, datatypes, and exception handling are not given here. When specific examples in the chapter do require a concrete language setting, Mesa is used.

> *Uniform call semantics.* Emissary has identical local and remote call semantics: exactly-once in the absence of crashes and last-one in the presence of crashes.
>
> *Powerful binding and configuration.* Emissary's binder permits the flexible specification and assignment of program modules to the nodes of a distributed system.

Strong typechecking. Emissary ensures that the compiler and binder perform the same typechecking for remote calls that they do for local calls.

Excellent parameter functionality. Emissary's runtime mechanism permits nearly all datatypes as parameters, including address-containing types that are not list structures. Parameters are passed by value-result.

Standard concurrency control and exception handling. Emissary invokes and executes remote procedures with no hidden concurrency—the caller blocks while the callee executes. All exceptional conditions are reported with the standard exception mechanism.

5.2 Design Overview

Emissary's design is composed of three distinct parts: orphan algorithms, remote call mechanisms, and distributed binding. The remainder of the chapter considers each in turn. The following shows how the parts combine to satisfy the five essential properties:

Orphan algorithms. To obtain semantic transparency during crashes, the uniform call semantics property demands that orphans be exterminated. This requires orphan extermination algorithms that are reliable in the face of crashes. In addition, crash notifications must be delivered using the standard exception-handling machinery of the programming language.

Remote call mechanisms. Once a distributed program has been loaded and started on all of its nodes, it executes normal, *steady-state* remote calls. To achieve local and remote transparency for steady-state calls, they must satisfy the uniform call semantics, typechecking, parameter functionality, and concurrency and exception control properties. The compiler (or stub translator) and RPC runtime environment bear most of these responsibilities.

Distributed binding. Before a distributed program can perform steady-state calls, it must undergo a binding *transient* where modules are configured, bound, loaded, and started. Transparency requires that remote binding schemes satisfy the strong typechecking and powerful binding properties. Careful extensions to local binders can transform them into distributed binders that meet this requirement.

All programs in this chapter are written in Mesa [62]. Readers unfamiliar with Mesa but versed in Pascal will find appendix 1 useful for explaining some of Mesa's uncommon features.

5.3 Orphan Algorithms

The purpose of orphan algorithms is to exterminate orphans after node failures, therefore guaranteeing last-once semantics in the presence of crashes. Since the Emissary remote call mechanism to be discussed in section 5.4 guarantees exactly-once semantics in normal, noncrash situations, Emissary's orphan algorithms act in conjunction with the remote call mechanism to satisfy the uniform call semantics property.

The orphan algorithms presented in this section are unimplemented. In addition, some low-level interactions between these algorithms and the underlying RPC machinery are not fully described. These interactions are clarified in section 5.4.1.4 on call mechanism details.

Readers familiar with Lampson's orphan discussion [49] will notice some common threads between this work and his. The work presented here discusses some new issues and elaborates a few of Lampson's general points in much greater detail.

5.3.1 Algorithm Definitions

The following terms and techniques will be used throughout this section. The models for crashes and stable storage are roughly those of Lampson's transactions model [48].

> *Nodes.* Each independent processor in the distributed system is called a *node.*
>
> *Processes.* Each node has some number of active processes. These processes can be enumerated from the *Processes* set. A process's state contains information that identifies its parent process and any remote call it may be executing.
>
> *Calls.* To clarify internode communication, remote procedure calls will be explicitly indicated. A remote call of procedure *P* in node *n* is written *n.P[arguments]*; a local call is simply *P[arguments]*. The declaration of a procedure that is usually called remotely is flagged with the keyword REMOTE. If a procedure is not declared REMOTE, it can only be called locally.
>
> *Exceptions.* Exceptions are handled using Mesa's standard mechanism (appendix 1).
>
> *Crashes.* A node has three cyclic phases in its life:
> *Normal* is when the node is conducting its usual business;
> *Crashed* or *down* is when the system is stopped;
> *Recovery* is when a crashed node restarts and makes itself consistent before declaring itself back to normal. While a node is recovering, it does *not* execute client programs or accept any remote calls except those necessary for the special business of recovery. Nodes are expected to be in the normal state except when they are down. The length of a crash is often short (seconds), but can last indefinitely (e.g., when there are hardware failures). Recovery is usually brief (seconds to minutes).
>
> *Stable storage.* Each node has stable storage that survives crashes. In the algorithm, information kept in stable storage is declared STABLE.
>
> *Communication.* Internode crash-recovery communication takes place via remote procedure calls (after this much work, it would be silly to use anything else). Because these remote calls during recovery are to repair the normal RPC mechanism, they employ logically different connections from those used for normal calls. They must be sent reliably—with connections newly established during recovery (or equivalent)—because orphan extermination messages that are lost, duplicated, or delayed can be catastrophic.

5.3.2 Orphan Definitions

Precise definitions of orphan-related terms clarify the algorithm descriptions and make the algorithms more concise. The definitions that follow are illustrated with examples from figure 5.1.

In the figure, calls between nodes are shown with heavy directed lines. Each call is identified with the names of its originating nodes (the nodes it is working for). For example, starting at the left, call *AB* is from node *A* to node *B*. In *B*, the two processes executing call *AB* each make a subcall to node *C*, ABC_1 and ABC_2. C makes an yet another subcall to *D*, ABC_1D. The thin lines within nodes *B* and *C* indicate this genealogy. Calls with no intranode relationships (i.e., thin lines) are independent. Thus calls *CA*, *BA*, and *CD* are independent from each other and from *AB* and *AB's* descendents.

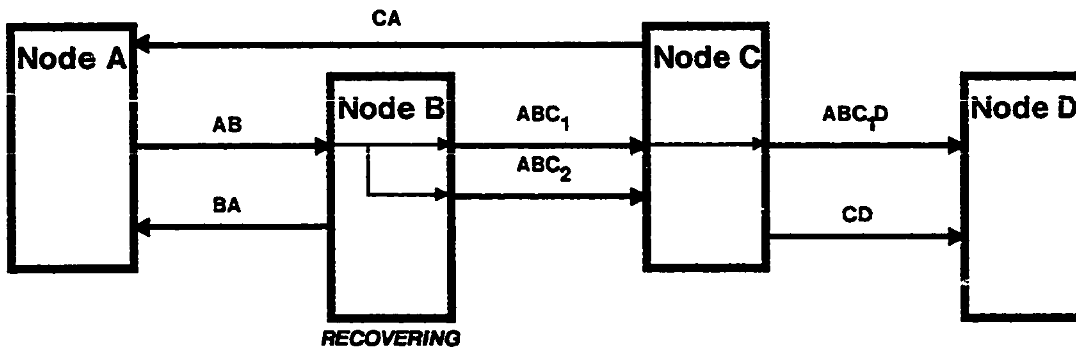


Figure 5.1: A distributed system with orphans originating at recovering node *B*.

With the illustration in mind, here are the definitions.

Child processes. The *child processes* of a remote call *P* are the processes in a remote node *N* that are servicing *P*. Initially *P* has only one child process in *N*, and it is invoked concurrently with other processes in *N* as discussed in section 4.1.1.7. *P* acquires additional child processes in *N* as the first process forks auxiliary concurrent processes, and they fork more, and so on. All of *P's* child processes are in node *N*. In figure 5.1, there is initially a single child process in node *B* resulting from call *AB*, and this child process spawns another. These concurrent child processes of *AB* are represented by the intranode lines in *B*, and they have initiated concurrent remote calls ABC_1 and ABC_2.

Child calls. The *child calls* of *P* are the *remote* calls made by *P's* child processes. In the figure, ABC_1 and ABC_2 are child calls of call *AB*, and ABC_1D is a child call of ABC_1.

Parent processes. The *parent process* of a remote call *P* is the process that invokes *P*. In the figure, the parent process of call *AB* is the anonymous process in node *A* that invokes procedure *AB*.

Parent calls. The *parent call* of remote call *Q* is the call, *P*, for which *Q* is a child call of *P*. In the figure, the parent call of ABC_1 and ABC_2 is *AB*. *AB* has no parent call and is called the *root* of the *AB* call tree. The transitive closures of the *parent call* and *parent process* relations are never cyclic; the call tree is never a graph because a process can only execute one remote procedure at a given time.

Child nodes and parent nodes. The *child nodes* of node *N* are the nodes executing any of *N's* child calls. The *parent nodes* of *N* are the nodes that started all of *N's* parent calls. In the figure, node *A* has parents *B* and *C* and child *B*; node *B* has parent *A* and children *A* and *C*; node *C* has parent *B* and children *A* and *D*; and node *D* has parent *C* and no children. Notice that the transitive closure of the *child node* and *parent node* relations can be cyclic (e.g., nodes *A* and *B* because of calls *AB* and *BA*).

Ancestors and descendents. The previous parent and child relationships always apply between two directly connected nodes. *Ancestor* and *descendent* relationships are represented by the transitive application of the parent and child relations. In the figure, ABC_1, ABC_2, and ABC_1D are the descendents of AB; ABC_1 and AB are the ancestors of ABC_1D.

Remote work. The *remote work* of a distributed system—or of a given node, process, or call—is the work being done by all remote calls and all of their child processes. All other work is local and is therefore contained in one node and has no parent nodes, processes, or calls. All seven calls in figure 5.1 are doing remote work; local work is not shown.

Orphans. An *orphan* (also called *orphan call* and *orphaned call*) is a remote call that has some ancestor executing on a crashed node. In the figure, if node B has crashed and is recovering, then calls BA, ABC_1, ABC_2, and ABC_1D are all orphans. Together, these four calls are called the *orphans of B*. Individually, each call is an orphan of its parent node and its parent call, if any. For example, ABC_1D is an orphan of node B and call ABC_1, and ABC_1 and ABC_2 are orphans of A and AB. Since call BA is a root call and has no parent, it is just an orphan of B.

5.3.3 Extermination

With this background, consider a crashed node N that is performing crash recovery. One of N's recovery responsibilities is to ensure that all of its orphaned calls are exterminated before it returns to normal operation. Section 4.1.1.4 suggested that one way to exterminate an orphaned call is by tracking down all of its orphan processes and killing them. To accomplish this, N contacts all of the nodes it called before the crash and tells them to exterminate N's orphaned calls. Each of these machines, in turn, exterminates *its* orphans caused by the child calls that N's child processes have made on behalf of N. This process repeats for all the descendents of N's orphans.

A recursive algorithm for this method is presented in algorithm 5.1. Basically, it works as follows. Each node keeps two sets in stable storage: *nodesIn* contains the parent nodes of all incoming calls, and *nodesOut* contains the child nodes of all outgoing calls (stable storage decisions and requirements are explained in section 5.3.3.2). During crash recovery, these sets are used to exterminate orphaned calls and to notify parents of the extermination (since the sets are stable they survive crashes). After a crash, the recovery code at the end of algorithm 5.1 is called. The code works in two steps (follow the algorithm as you read below).

Orphan extermination. The first part of recovery exterminates the recovering node's orphans, including orphaned descendents, by calling *RecoveryExtermination.* For each child node in *nodesOut*, *RecoveryExtermination* calls *ExterminateAll* on the child. Assuming that the child node is not itself in crash recovery (this is considered below), the child *Aborts* all of the calling parent's child processes. In addition, for each child that is itself executing a remote call, the *Abort* of the child process recursively *Exterminates* the child's orphan (this latter call is a *grandorphan*, or second-level descendent of the recovering parent). This extermination procedure repeats for all descendents. Note that *Exterminate* aborts only the child processes of the parent *process*, not all the child processes of the parent *node* like *ExterminateAll.*

Parent crash notification. The second part of recovery informs the recovering node's parent nodes that it crashed by calling *RecoveryCrashNotification*. For each parent in *nodesIn*, a remote call of *ChildCrashed* is made to the parent. This raises the *Crashed* exception in every parent process that had a remote call to the recovering node. Alternatively, a parent application program could have already received a *Failed* exception and explicitly called *ClientAbort* to abort and exterminate the failed (actually crashed) call. In this case, *ClientAbort's Exterminate* call (via *Abort*) repeats until the child's crash recovery code begins accepting remote *Exterminate* calls. It then succeeds and aborts the process. *Abort* ignores the unwanted *Crashed* exception should it arrive during the extermination.

An example clarifies the algorithm's behavior. In figure 5.1, assume that node *B* is recovering as indicated. *B's nodesIn* list contains *A*, and *nodesOut* has *A* and *C*. In the first part of recovery, *B's RecoveryExtermination* procedure calls *ExterminateAll* on *A* and *C*. Since *A's* remote work for *B* is contained wholly in *A*, *A* just aborts the local child processes of call *BA* and makes no recursive calls to *Exterminate*. Node *C*, however, has called *D* on behalf of *B*. Hence *C* not only aborts the child processes of calls ABC_1 and ABC_2, but it also calls *Exterminate* on node *D* to kill ABC_1D's child processes there. *C's* other calls, *CA* and *CD*, are unaffected. In the second part of recovery, node *B's RecoveryCrashNotification* procedure makes a remote call to *ChildCrashed* on *A*. If *A* is still waiting for call *AB* to complete—which is possible even though *A* earlier exterminated *independent* call *BA*—then a *Crashed* exception notifies *AB's* caller of the crash. If the caller is already aborting *AB*, the abort will complete when recovery is over.

5.3.3.1 Mutually Interacting Crash Recoveries

Extermination is more complex when two or more recovering nodes are participating in the same extermination. The reason is that the "call tree" information stored in *ProcessObjects* and used by *Exterminate* and *ExterminateAll* to track orphans is destroyed in a crash. For example, in figure 5.1, if both node *A and* node *B* are recovering, then when node *A* calls *ExterminateAll* on *B*, *B* will have no explicit record of calls ABC_1 and ABC_2. Instead, *B* has just its *nodesOut* list, which contains both *A* and *B*. At first glance this problem can appear easy to solve: When either *Exterminate* or *ExterminateAll* is called on a recovering node such as *B*, just make the call wait for *B's* normal recovery to complete—that is, for *B's* call of *RecoveryExtermination* to finish. This guarantees that *all* orphans, not just those requested by *Exterminate* or *ExterminateAll*, are exterminated. Unfortunately, this waiting approach is ideal *except* when two recovering nodes are in a child-parent node cycle, as are nodes *A* and *B*. When there is a cycle, deadlock results: if *A's* call of *B.ExterminateAll* is waiting for *B* to finish extermination, and *B's* call of *A.ExterminateAll* is similarly waiting for *A*, then the recoveries are deadlocked.

The solution to the problem of mutually recovering nodes is to not wait, but rather to carry on an independent *RecoveryExtermination*. This is why, in algorithm 5.1, that both *Exterminate* and *ExterminateAll* call *RecoveryExtermination* when they are invoked during the extermination phase of recovery. Unfortunately, this extra call of *RecoveryExtermination* involves some wasted effort since, for example, both *A* and *B* execute *RecoveryExtermination* twice (not all of *RecoveryExtermination's* work is done twice, however). Furthermore, not waiting causes a new

problem: mutual *RecoveryExterminations* can get into an infinite recursion because node information—which can be cyclic—rather than call information—which is acyclic—is used. These cycles are detected and broken by using the *recoveringAncestors* parameter of all the extermination operations. Basically, the *recoveringAncestors* set allows the call tree to be reconstructed from the cyclic graph of *nodesOut* information.

This revised behavior is clarified by finishing the example where *A* and *B* are in mutual recovery (this excludes all nodes and calls in figure 5.1 except *A*, *B*, *AB*, and *BA*). Considering just node *B*, when *B* first enters crash recovery it calls *RecoveryExtermination*. *RecoveryExtermination* finds node *A* in *nodesOut* (from call *BA*) and calls *ExterminateAll* on *A*. Node *A*, also in recovery, is unable to do the *ExterminateAll* and calls its own *RecoveryExtermination* procedure instead. This *RecoveryExtermination* on *A* finds *B* in *A*'s *nodesOut* list (from call *AB*) and calls *ExterminateAll* back on *B*. Node *B*, of course, cannot perform an *ExterminateAll* either, and calls its *RecoveryExtermination* routine (recursively) for the second time. But *B's RecoveryExtermination*, before doing anything else, checks *recoveringAncestors* and finds *B* in the list. This stops the recursion, and the chain of extermination calls then backs out successfully. Node *A* performs a symmetric sequence of calls as part of its recovery, although only checking is performed since all processes in both *A* and *B* are aborted in their crashes.

5.3.3.2 Costs and Stable-storage Requirements

An important property of the Emissary extermination algorithm is that it costs nothing for normal, steady-state calls except some small space. All information required for recovery is kept in the *nodesIn* and *nodesOut* stable sets, which can have a very compact bit-vector representation. While updating these sets may be costly because of the stable storage operations required, the expense is usually paid only once, when the first call is made between any two nodes. Further, any necessary dynamic manipulation of these sets *need not be completely accurate*: Old nodes can remain in the lists until removing them is convenient, say, with an aging scheme. Having the old nodes in the lists causes extra exterminations to be attempted during recovery, but this is not functionally harmful—it just degrades the performance of recovery. New nodes, however, must be added immediately: For an outgoing call, *nodesOut* must be updated before the call is sent; for an incoming call, *nodesIn* must be updated before the call is accepted for execution.

An alternative scheme to using *nodesIn* and *nodesOut* for extermination is to use only the *Process.worker* and *Process.parent* state information. The advantage of this scheme is that orphans can be precisely located through the family tree of descendents recorded in each child process—there is no need for the complicated *recoveringAncestors* cycle detection used in algorithm 5.1. But the disadvantage is that, to work in the presence of crashes, *worker* and *parent* must *always* be kept in stable storage. Since this requires writing the child process information to stable storage for *every* remote call, it is probably an unacceptably expensive method.

```
-- Emissary's recursive algorithm to exterminate orphans by having each node kill its own.
   Node: TYPE = ...; -- Internetwork address.
   myNode: Node = ...; -- The node that this variable lives in.
   NodeState: TYPE = {startRecovery, ..., exterminating, notifying, ..., normal};
   myNodeState: NodeState ← startRecovery; -- Current processor state of myNode.
   Process: TYPE = POINTER TO ProcessObject;
   myProcess: Process ← ... -- Currently executing process.
   ProcessID: TYPE = RECORD [ -- Indicates our internet-wide parent process.
            node: Node, -- Home node of process; myNode if local.
            process: Process ];
   ProcessObject: TYPE = RECORD [
            ... -- Other parts of a process object.
            worker: Node, -- worker will be myNode iff executing in this node.
            parent: ProcessID ]; -- parent is the ProcessID of the parent if there is one;
                             if parent is a root process then parent is parent.process (circular).
   Processes: SET OF Process ← ...; -- Contains all active processes in this node.
   nodesIn: STABLE SET OF Node ← EMPTY; -- Contains the nodes of all incoming calls.
   nodesOut: STABLE SET OF Node ← EMPTY; -- Contains the nodes of all outgoing calls.

LocalRootProcessID: PROCEDURE [process: Process] RETURNS [root: ProcessID] =
   -- Finds the local root process of process. If the root process is in this node, then the root's
   parent is itself (first case checked by UNTIL). If the top-level local process has a remote root
   (because of a remote call), then the top-level process's parent will name the root (second
   case).
            { FOR root ← process, root.parent.process -- Follow the parent link.
                 UNTIL root.parent.process= root OR root.parent.node# myNode DO ENDLOOP;
              RETURN[root.parent] };

Failed: SIGNAL = ...; -- Exception indicating that a remote call has failed because the remote
   callee refuses to respond (there is evidence that the child node has crashed or partitioned).
   Failed is raised by the underlying RPC mechanism and is never explicitly raised in this
   algorithm. Client programs that catch the Failed exception and decide to abort their remote
   work (thereby guaranteeing last-one semantics) should call ClientAbort.

Aborted: SIGNAL = ...; -- Exception indicating that the process in which Aborted is raised is
   being aborted and should finalize itself if desired. Aborted is signalled by DestroyProcess.

Abort: PROCEDURE [p: Process, recoveringAncestors: SET OF Node ← EMPTY] =
   -- Aborts process p and exterminates p's resulting remote orphan, if one. (The
   recoveringAncestors set is used only by Exterminate.) DestroyProcess[p] halts and destroys p
   and preserves process family relations by making p's spawned processes have p.parent as
   their parent. DestroyProcess SIGNALS Aborted IN p, and ignores any Crashed exceptions that
   might be waiting from a remote call to myNode's ChildCrashed routine.
            { IF p.worker # myNode THEN
                    p.worker.Exterminate[ [myNode,p], recoveringAncestors
                                              ! Crashed => CONTINUE ];
              DestroyProcess[p ! Crashed => RESUME ] };

ClientAbort: PROCEDURE [p: Process] =
   -- Aborts process p, but also ensures that extermination succeeds for the Abort. ClientAbort
   is intended to be called by clients who decide to abort outstanding activity—including
   remote calls—for some reason. This is why the Failed exception causes the Abort to be
   retried until it succeeds. Note that as it is written, ClientAbort (via Abort) aborts an entire
   process as well as exterminating the process's remote call (if any). Performing just the
   extermination (leaving the process intact) can be done by writing a new ClientExterminate
   operation that performs the first half of Abort. It's possible that clients may not want to
   exterminate orphans for some reason; this is a policy decision that is not made here.
            { Abort[p ! Failed => RETRY] };
```

Exterminate: REMOTE PROCEDURE [*parent: ProcessID, recoveringAncestors*: SET OF *Node*] =
-- Exterminate the orphaned call *myNode* has undertaken for the remote *parent*. How *myNode* does this depends on whether *myNode* is in crash recovery or operating normally. If *myNode* is in the *exterminating* phase of recovery, the *parent's* process(es) are already killed, as desired. As a consequence, however, it is impossible to recursively kill *parent's* grandorphans because that grandorphan information was in the now-destroyed processes. Thus *myNode* must call *RecoveryExtermination* to ensure that all orphans are killed. (It is not sufficient to wait for *myNode's* crash recovery to complete first, because waiting can cause deadlock; see the text.) If, on the other hand, *myNode* is operating normally, then *myNode* exterminates *parent's* call by aborting all of *parent's* child processes in this node. If any of *myNode's* processes are performing remote work for *parent*, their orphaned calls (the grandorphans) are automatically exterminated by *Abort*.

```
{ SELECT myNodeState FROM
        exterminating => RecoveryExtermination[recoveringAncestors];
        IN [notifying .. normal] =>
            FOR p IN Processes DO
                IF LocalRootProcessID[p] = parent THEN
                    Abort[p, recoveringAncestors] ENDLOOP;
        ENDCASE => ERROR };
```

ExterminateAll: REMOTE PROCEDURE [*parentNode*: *Node*, *recoveringAncestors*: SET OF *Node*] =
-- *ExterminateAll* operates similarly to *Exterminate* except that *all* calls *myNode* has undertaken for *parentNode* are killed by aborting *all* of *parentNode*'s processes in this node.

```
{ SELECT myNodeState FROM
        exterminating => RecoveryExtermination[recoveringAncestors];
        IN [notifying .. normal] => {
            FOR p IN Processes DO
                IF LocalRootProcessID[p].node = parentNode THEN
                    Abort[p, recoveringAncestors] ENDLOOP;
            nodesIn ← nodesIn - parentNode };
        ENDCASE => ERROR };
```

RecoveryExtermination: PROCEDURE [*recoveringAncestors*: SET OF *Node*] =
-- *RecoveryExtermination* is called by *myNode* during crash recovery. Because the precrash process information is gone after a crash, *RecoveryExtermination* cycles through *nodesOut*, calling *ExterminateAll* on each of our child nodes to ensure that all of *myNode's* orphans are killed. The purpose of *recoveringAncestors* is to prevent infinite recursion by detecting child node cycles. It works like this: Each node that performs an extermination during recovery (via *RecoveryExtermination*) adds itself to the set of *recoveringAncestors* (this set includes only ancestors in the *exterminating* phase, not those in normal operation). Later, during the extermination initiated by *myNode's* call of *child.ExterminateAll*, if *myNode* is asked to exterminate one of its own calls, it notices itself in *recoveringAncestors* and immediately returns, stopping the recursion.

```
{ IF myNode IN recoveringAncestors THEN RETURN;
  FOR child IN nodesOut DO
        child.ExterminateAll[myNode, myNode + recoveringAncestors];
        nodesOut ← nodesOut - child;
        ENDLOOP };
```

Crashed: SIGNAL = ...; -- Exception indicating that the remote call in which *Crashed* is raised went through a crashed node. The crashed node has already recovered and the call been exterminated. *Crashed* is raised only in *ChildCrashed*.

```
ChildCrashed: REMOTE PROCEDURE [child: Node] =
  -- Tell myNode's processes--just in case they are waiting—that their remote calls to child
  have crashed and been exterminated. Usually myNode will have given up and already
  started exterminating child's orphans, but perhaps myNode decided to wait for a long time.
  It is necessary to inform still-waiting processes of the crash so that their calls do not repeat
  without an intervening Crashed exception.
        { FOR p IN Processes DO IF p.worker = child THEN SIGNAL Crashed IN p ENDLOOP;
          nodesOut ← nodesOut - child };

RecoveryCrashNotification: PROCEDURE =
  -- RecoveryCrashNotification is called by myNode during crash recovery. Tell each parent of
  myNode that myNode has crashed and recovered. The crash recovery code below ensures
  that each ChildCrashed call completes successfully.
        { FOR parent IN nodesIn DO
                parent.ChildCrashed[myNode];
                nodesIn ← nodesIn - parent;
                ENDLOOP };

-- Extermination starts in two ways. First, application programs can call ClientAbort. This
  exterminates the one remote call in the aborted process, if any. Second, a crashed node can
  enter crash recovery. In this case, all orphan calls are exterminated. The code that performs
  exterminations during recovery appears below.

-- If myNode is in crash recovery the following code is run to perform extermination. The only
  emote calls allowed into myNode during this part of recovery are Exterminate and
  ExterminateAll. This allows two (or more) nodes that are in a child or parent node cycle to
  avoid deadlock if both (or all) are recovering from crashes simultaneously. The calls of
  RecoveryExtermination and RecoveryCrashNotification must successfully complete before
  ending recovery, otherwise there is no guarantee that all orphans are killed and all parents
  are notified. (Retrying after the Failed exceptions causes these operations to repeat until
  they succeed.)
                BEGIN -- Crash recovery.
                 ...;
                 myNodeState ← exterminating;
                 RecoveryExtermination[EMPTY ! Failed => RETRY];
                 myNodeState ← notifying;
                 RecoveryCrashNotification[ ! Failed => RETRY];
                 ...;
                END; -- Crash recovery.
                myNodeState ← normal.
```

Algorithm 5.1: Emissary's orphan extermination algorithm.

Observe that the cost of the extermination algorithm during crash recovery is not a major issue. Crashes are expected to be an infrequent event; putting off until crash recovery work that hinders normal operation is usually a sound decision. This is why *nodesIn* and *nodesOut* are used for extermination and not *Process.worker* and *Process.parent*: The former method complicates crash recovery substantially and results in some wasted motion, but it has negligible steady-state overhead. The latter method is cheaper in recovery, but has high steady-state overhead.

5.3.3.3 *Incomplete Exterminations*

An unfortunate disadvantage of the extermination algorithm is that recovery may never complete. If any of the nodes executing orphaned calls are themselves crashed and down, a call to *Exterminate* on the down node cannot complete. This prevents its orphans from being killed, and recovery must wait indefinitely for the broken node to come to life in order to guarantee last-once semantics. For example, if node C in figure 5.1 is down, then node B cannot complete recovery because it is impossible to get C to exterminate B's orphans—in this case, call ABC_1D. If node D is down instead of C, then C faces the same dilemma because C must now discover that D has no orphans. A partitioning of the network causes exactly the same problem because the nodes of the isolated subnet are effectively down, e.g., if C and D were cut off from A and B. Furthermore, a partitioning that isolates a parent node also prevents extermination from completing because *RecoveryCrashNotification* cannot terminate.

Consider some solutions to this problem:

> One solution is simply to wait on the assumption that the down nodes or broken communication will eventually be repaired. But this is clearly unrealistic, especially on performance grounds.
>
> A second solution is to abandon extermination altogether for some other scheme, but the low steady-state cost of extermination is too attractive for this.
>
> A third solution is to modify the extermination scheme to deal with the problem of crashes in orphan nodes. Even if the modifications are expensive, this course is sound as long as the steady-state cost is negligible. This is true because the modified algorithm will need to be used only when extermination itself fails, and this is expected to be rare.

The next two sections present solutions of the second and third type. The first solution is a completely new scheme that can either augment or replace extermination; both possibilities are discussed. The second solution is an enhancement to extermination that is primarily useful for dealing with down nodes.

5.3.4 Expiration

If the nodes of a distributed system have synchronized clocks, then orphans can be killed simply by establishing a time limit on each remote call. More precisely, an *expiration time* is associated with each process. Strictly local processes—those doing no remote work—have a limit that never expires. Processes that are doing remote work, however, inherit their expiration times from their

parent processes. In this way, the root process of a remote call sets the time limit for all of call's child processes. Whenever a process reaches its expiration time and is still executing, it is declared a (potential) orphan and promptly aborted.

Notice that the expiration scheme requires no internode communication to detect or eliminate orphans: since the expiration time of a given remote call arrives with the call, each node kills the orphans of a given call independently and simultaneously (within the clock skew of the system). This has two desirable properties:

> First, it works equally in the presence of partitioned and down nodes. It therefore overcomes the only drawback of extermination.
>
> Second, it requires none of the complicated cycle-detection mechanism that is used by extermination during mutual recoveries.

On the negative side, however, expiration has an extremely unpleasant property: If a call's expiration time is too short, then the call can be declared an orphan and aborted *before* it has time to complete. For this reason, expiration is best used in conjunction with extermination. The hybrid extermination-expiration scheme works as follows: The expiration time for every remote call is set into the future by a very large *ExpirationInterval* (for example, minutes). If a remote call fails because of a crash or an explicit call to *ClientAbort*, extermination is tried first. Extermination will usually succeed, and very quickly, unless there is a down node or a partitioning. In the latter cases, extermination will fail and the exterminating node must simply wait for expiration—i.e., for *ExpirationInterval* to pass—before continuing. This ensures that the now-expired orphans have been killed.

There is another reason why *ExpirationInterval* must be longer than the time taken for a call to receive the *Failed* exception. If *ExpirationInterval* were shorter, then the root process of a remote call could repeat the call without knowing that its descendent calls had been aborted. This would violate exactly-once semantics. For example, assume that a remote call has expired, all of its child processes are aborted, and the root process is *not* notified of the expiration (virtual crash). If the root process's RPC mechanism now retransmits the same *call* message without knowing that the call expired, then the remote work of the call will start again because the child nodes have no record of the now-expired call. An alternative method of root-process notification is to call explicitly a *ChildCrashed*-like routine on the root node. This could eliminate the implicit notification that results from making *ExpirationInterval* longer than the call failure time, but seems unnecessarily complicated because *ExpirationInterval* needs a very large value for other reasons, as discussed above.

The details of one expiration algorithm are given in algorithm 5.2. It operates basically as described above: Every *ProcessObject* has an *expirationTime* that is set by *CreateProcess* to either an inherited *expirationTime* or *NeverExpires*. There is a special *CheckExpirationTimes* process that wakes up every *ExpirationInterval* and aborts any expired orphans. (Notice that this does not abort a process precisely when its limit expires, but within one *ExpirationInterval* of the expiration. This is discussed in the algorithm.) Finally, the *ClientAbort* procedure and crash recovery code must be changed to wait for all orphans to expire when extermination fails.

As an example of expiration, assume that node *B* in figure 5.1 crashes and stays down. The process making call *AB* eventually gets the *Failed* exception; assume that *ClientAbort* is invoked to abort the call. Unfortunately, *ClientAbort's* extermination of call *AB* also fails because *B* is down. Now if node *A* relied on extermination alone, it would be forced to repeat the extermination of *AB* until node *B* recovered. With expiration, however, *A* just waits for all of *AB's* orphans to expire (the exact waiting time is explained in the algorithm): In node *B*, *B's* crash has already killed *AB*; in node *C*, *C's* *CheckExpirationTimes* process aborts ABC_1 and ABC_2; in node *D*, *D's* *CheckExpirationTimes* process aborts ABC_1D. Finally, in node *A*, call *BA* is aborted by *A's* own *CheckExpirationTimes* process.

```
-- Emissary's expiration algorithm to kill potential orphans by automatically aborting all
   processes whose assigned time limits expire. Familiarity with the previous extermination
   algorithm is assumed.
   Time: TYPE = ...; -- Monotonically increasing time; resolution of at least seconds.
   NeverExpires: Time = LAST[Time]; -- Distinguished time value; infinity is convenient.
   ExpirationInterval: Time = ...; -- Conveniently large value; probably at least minutes.
   MaxClockSkew: Time = ...; -- Upper bound on the error in all system Clocks (see text).
   NodeState: TYPE = {startRecovery, ..., exterminating, notifying, ..., normal};
   myNodeState: NodeState ← startRecovery; -- Current processor state of myNode.
   Process: TYPE = POINTER TO ProcessObject;
   myProcess: Process ← ... -- Currently executing process.
   ProcessObject: TYPE = RECORD [
        ... -- Other parts of a process object.
        expirationTime: Time, -- Deadline after which this process is to be aborted.
        worker: Node, -- worker will be myNode iff executing in this node.
        parent: ProcessID ]; -- parent is the ProcessID of the parent if there is one;
                        if parent is a root process then parent is parent.process (circular).
   Processes: SET OF Process ← ...; -- Contains all active processes in this node.

Clock: PROCEDURE RETURNS [time: Time] = {...};
   -- Clock appears to be a built-in operation at this level. Clocks are assumed to be
   synchronized system-wide, but some synchronization error (skew) is permitted; see the text.
   The maximum error is assumed to be MaxClockSkew.

CreateProcess: PROCEDURE [parent: Process, ...] RETURNS [child: Process] =
   -- CreateProcess is (logically) called by Spawn, FORK, or whatever process-creation primitives
   a language provides. Shown here is the child's inheritance of the parent's expirationTime: If
   parent is performing remote work, child's expiration is inherited to ensure that all child
   processes have the same expiration time. If parent is performing local work, there is no
   expiration time. (Of course, if crash recovery starts the system with the expirationTime of
   the local root process set to NeverExpires, then simply assigning child.expirationTime ←
   parent.expirationTime is sufficient (and desirable). The conditional assignment below is for
   clarity.)
        { ... -- Other parts of process creation.
         child.expirationTime ← IF LocalRootProcessID[parent].node # myNode
              THEN parent.expirationTime ELSE NeverExpires;
         ... };
```

```
CheckExpirationTimes: Process =
  -- CheckExpirationTimes is a special high-priority process that periodically (every
  ExpirationInterval) looks through Processes for any process that has exceeded its
  expirationTime. Any process that has expired is aborted with LocalAbort. (LocalAbort, not
  shown, is identical to Abort except that any remote call of p is not exterminated (it will
  expire too). LocalAbort is therefore the same as DestroyProcess.) Observe that an expired
  process may not be aborted exactly when its limit expires, but rather within one
  ExpirationInterval of the expirationTime. This delay is handled by the revised crash
  recovery, below. Also, expired processes may not actually be killed until a short time after
  CheckExpirationTimes starts executing. This is not harmful since no (expired) process can
  execute as long as CheckExpirationTimes is running.
        { SetProcessPriority[high];
          DO
            SuspendProcess[ExpirationInterval];
            FOR p IN Processes DO
                IF p.expirationTime < Clock[] THEN LocalAbort[p]; -- Time limit expired.
                ENDLOOP;
          ENDLOOP };

-- An extermination that fails can start a wait for expiration in two ways. First, ClientAbort in
  the previous algorithm can get a Failed exception from Exterminate. In this case,
  ClientAbort is changed to wait for expiration instead using RETRY in the catchphrase.
  Second, RecoveryExtermination and RecoveryCrashNotification can get Failed exceptions
  from Exterminate and ChildCrashed, respectively. In this case, the crash recovery code
  invoking them is also changed to wait and not RETRY. These changes are presented below.
  The total waiting time of 2*ExpirationInterval+ MaxClockSkew has three components:
  The first ExpirationInterval component is for the orphans to expire (the expiration time of
  any of myNode's remote calls cannot be more than ExpirationInterval into the future). The
  second ExpirationInterval component is to ensure that the CheckExpirationTimes processes
  on all nodes have had an opportunity to run (while their periods are identical, they are not
  synchronized). The third MaxClockSkew component is to compensate for any possible
  Clock errors in the distributed system.
                BEGIN -- Crash recovery with expiration.
                 ...;
                 myNodeState ← exterminating;
                 RecoveryExtermination[EMPTY ! Failed => GOTO WaitForExpiration];
                 myNodeState ← notifying;
                 RecoveryCrashNotification[ ! Failed => GOTO WaitForExpiration];
                 ...;
                EXITS
                 WaitForExpiration =>
                       SuspendProcess[2*ExpirationInterval+ MaxClockSkew];
                END; -- Crash recovery with expiration.
                myNodeState ← normal.
```

Algorithm 5.2: Emissary's orphan expiration algorithm.

5.3.4.1 Costs and Clock Requirements

The cost of the expiration algorithm is not great: a *Time* must be carried in every *call* message, and a *CheckExpirationTimes* process must run—infrequently—in every node. Furthermore, by setting *ExpirationInterval* to a sufficiently large value, very little nonorphan work will be aborted by premature expirations. This can be practically assured by setting *ExpirationInterval* larger than the expected maximum time that any remote caller is likely to wait for a *return* before aborting the call. (A variation of expiration that permits shorter time limits without unnecessary aborting is discussed later, in section 5.3.7.3.)

A hidden but significant cost of expiration is the work required to synchronize clocks. There are two important observations here: First, any realistic system will clearly have clocks, and they will be synchronized to some reasonable degree. Second, the amount of error in the clocks—their skew—does not have to be zero. If a bound can be established on the maximum skew in the system (*MaxClockSkew* in the algorithm), this skew can be added to *ExpirationInterval* to bound the total waiting time required. Lamport's excellent paper on time and clocks [43] gives both a clock synchronization algorithm and an upper bound on its postsynchronization skew.

Readers unhappy with clock synchronization requirements will find a nonclock scheme below.

5.3.5 Epochs and Reincarnation

The existence of synchronized clocks is a critical assumption in the expiration algorithm. Fortunately, in their absence, ensuring that all orphans are killed when an extermination fails is still possible. The basic method is to synchronize nodes only very coarsely—just at the times when extermination fails and outstanding orphans must still be killed—rather than very finely, with clocks. Before discussing how to achieve this coarse synchronization, which occurs at the beginning of *epochs*, it is first necessary to modify last-one semantics slightly.

5.3.5.1 Weak Last-one Semantics

To handle extermination failures, the following change to the last-one semantics guarantee is proposed. Suppose that instead of exterminating orphans during recovery—an extremely selective distributed *reset*—orphans are just exterminated when their execution first interferes with that of new, nonorphan calls. Assume that a lost orphan in node *N*—that is, an orphan that *N* would have killed had extermination succeeded—can be spotted and killed even though extermination failed. If this lost orphan that is continuing to execute is killed *before* any of its parent's new calls begin to execute in *N*, then transitive last-one semantics are not violated. *Weak* last-one semantics are when orphans are killed in each node before any new parent calls can restart *in that node*. Compare this with the original *strong* last-one semantics, where orphans are killed before their parent calls can restart *in any node*. Thus the weak last-one guarantee ensures that orphans are killed before they can interfere with any new work, but it does permit them to continue executing in realtime after a recovery.

5.3.5.2 Regular Reincarnation

Achieving weak last-one semantics is straightforward. When an extermination fails because of a down node or a partitioning, the node that started the extermination declares a new *epoch* and immediately *reincarnates*. To reincarnate, a node aborts all processes rooted in *nodesIn*, stores the new epoch, and then continues its local work (which will probably start new remote calls that circumvent the down node). As a part of the epoch scheme, every node knows what epoch it is operating in, and every message carries the epoch in which it was sent. (In expiration, the former corresponds to *Clocks* and the latter to a call's *expirationTime*). Whenever a node receives a *call* message from a new epoch, later than its own, it also reincarnates. Because reincarnation is an act of crash recovery, all normal work ceases until it is complete.

After reincarnating, all of a node's remote calls will carry the new epoch. Furthermore, all calls received from previous epochs receive a notification of the new epoch in reply and are otherwise ignored. This causes any called or calling nodes—and thus their called or calling nodes, and so on throughout the system—to reincarnate and abort all processes doing remote work: these are the processes that *might* be orphans, and they must all be aborted. In this way, the propagation of a new epoch eventually kills all possible orphans, thus guaranteeing weak last-one semantics. (Readers who find this scheme too severe will find relief below.)

When all nodes in a network are in the same epoch, we say that the network is in *equilibrium*. If desired, a new epoch can be heralded by broadcasting the epoch change rather than by waiting for minstrel calls to slowly spread word of the new era. Because broadcasting is assumed to be unreliable, however, each call must still carry its epoch for isolated hamlets.

As an example of reincarnation, assume that node *B* in figure 5.1 is down and unable to recover. Further assume that call *AB* receives the *Failed* exception and decides to call *ClientAbort*. As it is written, *ClientAbort* attempts extermination forever, but assume that it is changed to cause reincarnation if extermination fails. When *A* finally reincarnates and declares a new epoch, it aborts the child processes of calls *CA* and *BA* because *B* and *C* are in *A's nodesIn* list. The new epoch can now spread in a number of ways; assume that call *AB* can be rerouted around *B* to *C*. When node *C* receives the new epoch from *A*, it kills the remote work in its *nodesIn* list, which contains node *B*. When *C* finishes reincarnation and calls *D* for *A*, node *D* reincarnates and aborts all of *C's* old-epoch child processes. The new epoch has now reached every active node; the network will regain equilibrium when *B* returns to service. Note that when *B* enters crash recovery after being down, it will try to exterminate its orphans. *B's* first remote call of *Exterminate* will be rejected and return the new epoch, however, so *B* reincarnates without delay.

Some unusual cases of epoch interaction deserve special attention. Consider the following situations of down nodes and partitioned networks.

> *Returning to service.* When a down node returns to service it will be in an old epoch until either the first new-epoch call arrives or one of its new calls is rejected and returned with a notification of the new epoch. In either case, when the node notices the new epoch it reincarnates immediately. Since the node is returning to service, it can be executing no

orphan calls locally. If the node had outstanding remote calls when it crashed, then these orphans are either already dead (if the rest of the network is in equilibrium) or will be killed (not yet in equilibrium).

Merging partitioned networks. There are two possibilities when two partitioned subnets merge with each other and each is in a local state of equilibrium. First, both may be in the same epoch. In this case neither subnet is doing work for the other and they can merge trivially. Second, both may be in different epochs. In this case again, neither subnet is doing any work for the other, but this time the earliest subnet must reincarnate, albeit unnecessarily. Finally, if one or both of the subnets is not in equilibrium, then the latest epoch of both will eventually propagate throughout the network and bring the merger into equilibrium. This could take arbitrarily long—indeed, a very large or crash-prone distributed system might *never* reach equilibrium—but this does not matter since the reincarnation guarantees only weak last-one semantics.

Merging independent epochs. In a large system, two or more separate exterminations can fail at the same time. In this situation, each node that started an extermination declares a new epoch. These nodes then spread the epoch to other nodes in the system. This case is identical to the network merging situation described above.

5.3.5.3 Gentle Reincarnation

Unfortunately, declaring a new epoch is very close to a master *reset* that crashes the entire distributed system. In particular, aborting all processes rooted in *nodesIn* kills all remote work whether it is actually orphaned or not. Furthermore, without *nodesIn* or the family history in a *ProcessObject, all* processes would have to be aborted because strictly local processes could not be distinguished from those belonging to orphans. Back in section 4.1.1.4, this latter scheme was judged unacceptably severe. Of course, this judgment is now invalid because epochs are declared only when extermination fails, and exterminations can fail only after crashes; thus new epochs occur seldom. Still, it is easy to change the epoch scheme to preserve as much remote work as possible, that is, abort only orphan processes.

The revised extermination method that preserves as much work as possible is called *gentle reincarnation.* It works roughly as follows: When a node reincarnates gently, it checks with the parent nodes of all of its remote calls (in *nodesIn*). If a parent does not respond, the parent's calls are declared orphans and exterminated. If a parent does respond, the parent is guaranteeing that he is in the new epoch and has no remaining orphans. The parent asserts this because he has already recursively checked with his parents, and so on for all ancestors. When the parent is alive, the child preserves the parent's work intact. Notice that all remote work saved in this gentle way is aborted by regular reincarnation. The process of checking ancestors in reincarnation is analogous to tracking orphans (descendents) in extermination, except the search goes up the call chain instead of down.

An example of gentle reincarnation is enlightening. In figure 5.1, assume that node B is down and that A declares a new epoch because it cannot exterminate *Failed* call AB. A must check its two parents B and C when it declares the epoch:

A's parent B. A finds parent B down, and exterminates call BA.

A's parent C. When A checks with parent C, C discovers it is in an old epoch (because A's check call carries the new epoch). While A waits, C recursively reincarnates and checks its own parents. During its reincarnation, C finds its parent B down and exterminates both calls ABC_1 and ABC_2. (Gentle reincarnation has now killed two lost orphans that A's *Failed* extermination abandoned.) C's extermination of call ABC_1 also recursively tries to kill orphan ABC_1D in node D:

> *C's child D.* D is still in the old epoch and ignores C's *Exterminate* call, reincarnating itself instead (because C's call carries the new epoch). D's reincarnation simply checks back with C, which is already in the new epoch. D then exterminates orphaned call ABC_1D but preserves valid call CD. (This kills the third lost orphan.)

Node C, now finished exterminating ABC_1D, continues. Since C has exterminated all orphans and reincarnated completely, it returns to A (because A asked C to reincarnate to begin with). Because C returns in the new epoch, A preserves call CA.

Nodes A, C, and D are now in the new epoch, and nonorphaned calls CA and CD are still executing. All other calls were orphans and were killed.

5.3.5.4 Costs and Communication Requirements

The extra per-node cost of gentle reincarnation over regular reincarnation is the time spent checking with ancestors. The computation to do this is little in each node, but the waiting time is potentially long because reincarnation cannot complete until all ancestors are traced to either a root node or a crashed ancestor. Putting these small per-node costs aside, gentle reincarnation has a much more serious system-wide communication cost: An epoch declared by gentle reincarnation must still propagate to all the nodes in a distributed system, even though it does not abort all the remote work in them. Thus even gentle reincarnation does checking proportional to the size of the whole distributed system, although it aborts only orphaned work. This problem is not serious in small systems, but in those with thousands or millions of nodes checking ancestors can involve enormous communication traffic: a three-node call can result in an extermination failure that (eventually) causes thousands or millions of nodes to reincarnate. For this reason, reincarnation is considered inferior to expiration.

5.3.6 A Comparison of Orphan Algorithms

Comparing the different activities of the four orphan algorithms gives clear insight into their behavior. In extermination, only nodes with orphans are checked and only orphaned work is aborted. In expiration, all nodes periodically check their own processes for orphans, and only orphaned processes are aborted. In regular reincarnation, all nodes are checked and all remote work is aborted. In gentle reincarnation, all nodes are checked but only orphaned work is aborted.

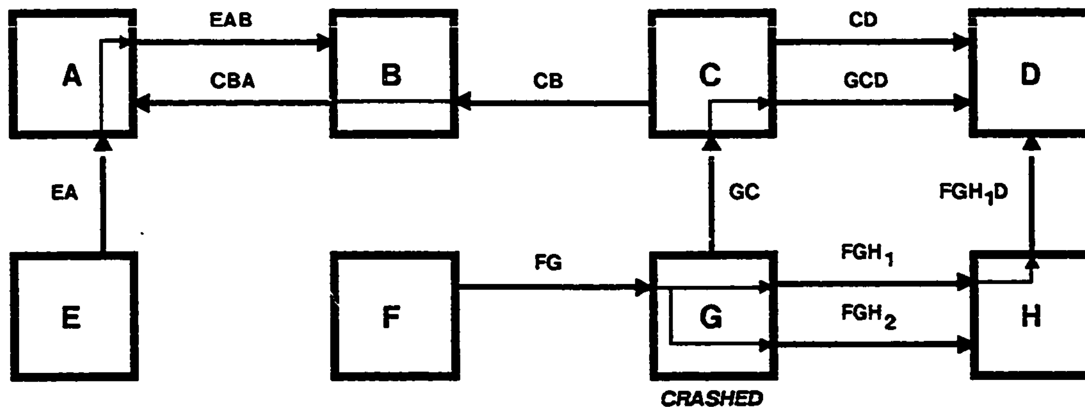


Figure 5.2: A distributed system with orphans originating at crashed node *G*.

In figure 5.2 (notation from figure 5.1), assume that node *G* crashes as indicated and that node *F* eventually notices the failure and tries to abort call *FG*. The four algorithms will then behave as follows.

Extermination. Assuming that *G* enters crash recovery quickly and does not stay down, *G* kills its own orphans by exterminating calls *GC*, FGH_1, and FGH_2 as the first part of recovery. As a consequence, *C* and *H* recursively exterminate calls *GCD* and FGH_1D on *D*. Finally, *G* calls *ICrashed* on *F* to raise the *Crashed* exception. Notice that nodes *A*, *B*, and *E* are *never* involved.

Expiration. If *G* remains down then *F's* extermination of call *FG* will fail. If node *F* suspends recovery and waits for expiration to complete, then orphaned calls *GC*, *GCD*, FGH_1, FGH_2 and FGH_1D are killed by the orphan expiration tasks in nodes *C*, *D*, and *H*.

Regular reincarnation. If *G* remains down then *F's* extermination of call *FG* will fail. In this case, node *F* performs a regular reincarnation. The new epoch declared by *F* eventually arrives at the nodes *A–H* through alternate call paths not including *G*. As it travels, all eleven remote calls and their child processes are aborted. In regular reincarnation, all remote calls must be aborted because, when *G* is down, it is impossible to tell which outstanding calls are on behalf of *G*. Thus calls *EA*, *EAB*, *CBA*, *CB*, and *CD* are needlessly killed.

Gentle reincarnation. If node *F* uses gentle rather than regular reincarnation when *G* stays down, then only orphaned calls *GC*, *GCD*, FGH_1, FGH_2 and FGH_1D are killed. But as the new epoch is carried through the system, each node *A–H* has to check its parents, resulting in wasted effort for nodes *A*, *B*, and *E*.

5.3.7 Other Orphan Schemes

The merit of Emissary's extermination and expiration algorithms is their low steady-state cost: carrying the *expirationTime* with each remote transfer and checking it occasionally. There are, however, other methods of killing orphans that have not been mentioned.

5.3.7.1 Reincarnation Only

Killing orphans is possible by using just reincarnation without extermination. The problem with this, as discussed, is reincarnation's master *reset* characteristic. Resetting after every crash—rather than just when extermination fails—is unacceptable because of the massive amount of aborted work. Gentle reincarnation solves this problem by checking with parents and exterminating only when necessary, but this still includes checking in proportion to the remote work of the entire distributed system. Gentle reincarnation is attractive, however, when a system does not have synchronized clocks.

5.3.7.2 Reliable Broadcasting

Orphans can be killed by a completely new method when all the nodes in a distributed system can be reached with a *reliable* broadcast [9,21]. Assume that the root process of any remote call assigns the call a unique identifier, *uid*. Require that any child processes working on a remote call—direct or spawned—inherit this *uid* just as they inherit *expirationTimes* in the expiration scheme. When a node crashes, its recovery phase simply broadcasts (reliably) an *abort* message with a list of its outstanding *uids*. When the broadcast finishes, all normal and recovering nodes have killed their orphans. Down nodes are fine; orphans in them are killed when they receive the (late) broadcast during recovery.

Reliable broadcasting is a simple and effective scheme, but it has two serious drawbacks. First, it is not clear how much work is needed to perform a reliable broadcast; certainly stable storage is needed. Second, partitions of the network cause the scheme to fail unless partitions carefully coordinate their merging. This coordination seems remarkably similar to merging epochs (or synchronizing clocks).

5.3.7.3 Deadlining with Postponement

Lampson has proposed an orphan scheme similar to expiration; he calls it *deadlining* [49]. His scheme has a nice enhancement that allows very short deadlines (expiration times) to be used. If the work of a call ever passes its deadline, it is not immediately aborted. Instead, the processes whose deadlines have expired try to *postpone* the deadline first. To postpone a deadline, a process asks its parent process if the deadline can be delayed; this checking is performed all the way up the call stack to the root process. If some parent node has crashed, the deadline will not be postponed and the orphan work is aborted. If the root node is reached and it postpones the deadline, nonorphan work proceeds with a new deadline. Note that requesting a postponement in the deadlining scheme is very similar to checking with ancestors in gentle reincarnation.

The main added cost of postponement is the extra internode communication. For this reason, deadlines must be chosen carefully: setting them too early causes unnecessary postponement(s); setting them too late causes delays during crash recovery. The expiration algorithm confidently sets a very large deadline because, during most recoveries, extermination kills orphans immediately. Thus expiration has no postponement overhead, and recovery is delayed only when there are down nodes or network partitions.

5.3.8 Reflections

Emissary's extermination and expiration algorithms have not been implemented. An operational scheme will unquestionably vary from the schematic methods given here; dealing with the details of a specific system's process machinery is enough to ensure this. In addition, an implementation for a real distributed system will want to evaluate the following optimizations and difficulties in light of the system's particular environment.

> *Fast stable storage.* The availability of fast stable storage (i.e., stable storage with read and write times comparable to those of main memory) can change the orphan algorithms significantly. In particular, with fast stable storage it is possible to record *Process.parent* and *Process.worker* (call tree information) for every remote call. This greatly simplifies extermination by removing the need for code that detects node cycles during mutual recoveries. Expiration or reincarnation are still needed, however, to deal with prolonged crashes and partitionings. Finally, fast stable storage can possibly eliminate orphan algorithms altogether. Orphan algorithms guarantee only last-one semantics, but—with fast stable storage—transaction-oriented at-most-once semantics may be feasible for every call.
>
> *Enumeration.* In algorithm 5.1, the FOR loops are all elaborated serially. They could just as well be done in parallel. This is especially useful in enumerations of *nodesIn* and *nodesOut* by *RecoveryExtermination* and *RecoveryCrashNotification*, where great speedups are possible because *Exterminate*, *ExterminateAll*, and *ChildCrashed* can truly execute in parallel.
>
> *Synchronization.* Many synchronization details were omitted in the algorithms. For instance, *Abort* removes a process from *Processes* at the same time that *Exterminate* is enumerating this set. If *Processes* is represented as a linked list then care is needed to maintain consistency. All process manipulation operations should be in a monitor to prevent this and similar troubles, although this is not shown.
>
> *Bookkeeping.* The straightforward statement of the algorithms causes them to undertake some unnecessary work. The process tree, for example, is walked step-by-step in *LocalRootProcessID* when only the root is wanted. Additional bookkeeping, perhaps in the simple guise of reorganized data structures, can eliminate most of this extra work. In some cases, more complicated algorithms can perform necessary work only once, or in a better order (e.g., eliminating redundant *RecoveryExterminations* during mutual recoveries).

The details, complications, and optimizations discussed here—and all the others that only an implementation effort will discover—are left as essential future work.

While orphan algorithms are interesting in their own right, their fundamental purpose must not be forgotten: to guarantee last-one semantics. The basic extermination algorithm meets this guarantee with low cost, but it also has some probability of failure. When this probability is unacceptable, the expiration and reincarnation algorithms reduce it to zero, but for a much greater cost—system-wide synchronized clocks, or system-side gentle reincarnation. The important point is that the cost of orphan insurance gets higher as the desired semantic coverage gets stronger. This section has focussed on absolute orphan coverage, and this may be impractical for some environments. Realistic distributed systems may want to underwrite specially tailored policies—with appropriate premiums—by selecting combinations of orphan algorithms.

5.3.8.1 Transparency and the Essential Properties

The goal of this section was to develop orphan extermination algorithms that satisfy the uniform call semantics and standard exception-handling properties in the presence of crashes. Emissary's extermination algorithm eliminates orphans quickly and efficiently when none of the orphaned calls are in down nodes. When nodes are down and crash recovery must proceed anyway, the expiration algorithm ensures that orphans are killed after a suitable waiting period. Acting as a team, Emissary's orphan algorithms guarantee last-one semantics in the presence of crashes; this satisfies the uniform call semantics property. In addition, both algorithms raise *Crashed*, *Failed*, and *Aborted* exceptions in orphan-related processes whenever possible; this satisfies the standard exception-handling property.

5.4 Remote Call Mechanisms

This section discusses the details of Emissary's actual remote procedure invocation and execution machinery. The development proceeds in four steps:

> *Remote call machinery.* An approach to compiling remote procedure calls is presented. This includes code sequences for compiled calls and a complete program to handle all RPC runtime operations. The Emissary runtime mechanism gives exactly-once semantics under normal conditions and uses the orphan algorithms of section 5.3 to handle crashes.
>
> *General RTransfers.* An implementation of general *RTransfers* (section 2.1.3.1) is described in terms of the previous *procedure* machinery. A general mechanism for *RTransfers* is necessary so that all language-level communication primitives—exceptions, coroutines, and so forth—can be implemented.
>
> *Marshaling parameters.* A scheme for marshaling common Mesa datatypes is presented. (Marshaling was defined in section 2.2.5.1.) The result is an approach that compiles parameter marshaling and unmarshaling code for most types, including allowable address-containing types. The proposed marshaling mechanism has excellent, but not complete, parameter functionality.
>
> *Stubs.* The remote procedure, *RTransfer*, and marshaling schemes developed in the previous three steps assume that the compiler and runtime system can be changed. When such direct changes are infeasible, the stub approach introduced in section 2.1.4 can be used to retain high transparency at some cost in performance. This final step discusses the transparent implementation of language-level stub mechanisms.

5.4.1 Remote Call Machinery

The Emissary remote procedure design has two components: the compiler changes needed to generate the code that calls and returns from remote procedures, and the abstract machine operations that support the generated code. The Emissary machinery described in this section, while unimplemented, is directly derived from a series of five operational mechanisms. I implemented three of these mechanisms (two were implemented by others); all five were studied and tested thoroughly for the Emissary design.

5.4.1.1 *Procedure Call Code Sequences*

To understand the compiler changes required for Emissary's remote calls, first consider the following code that is generated for a local call of procedure *P* by procedure *Q*.

The code in procedure *Q* that makes a local call to *P*:
[$result_1$, ..., $result_n$] ← *P*[$argument_1$, ..., $argument_n$];
    Allocate an *argumentRecord*;
    Compile and copy $argument_1$, ..., $argument_n$ into *argumentRecord*;
    resultRecord ← *InvokeProcedure*[*P*, @*argumentRecord*];
    Copy from *resultRecord* into $result_1$, ..., $result_n$;
    Deallocate *resultRecord*.

The body of local procedure *P*:
P: PROCEDURE [$argument_1$, ..., $argument_n$] RETURNS [$result_1$, ..., $result_n$]
    Copy *argumentRecord* to $argument_1$, ..., $argument_n$ in *P's* activation record.
    Deallocate *argumentRecord*.
    ...
    Do work of procedure...
    ...
    Allocate a *resultRecord*;
    Compile and copy $result_1$, ..., $result_n$ from activation record into *resultRecord*;
    ReturnFromProcedure[@*resultRecord*].

InvokeProcedure and *ReturnFromProcedure* are basic low-level hardware operations that use registers for their arguments. Notice that *Q's* parameters are explicitly moved into separate records (*argumentRecord* and *resultRecord*) before being copied into or returned from the activation record of *P*. This is necessitated by the definition of *Transfer*, and Mesa (logically) performs this separate copy operation. Compiler and runtime optimizations frequently reduce the number of copies for local calls.

Now consider the code for an Emissary remote call. Because all the information for a call—procedure name, arguments, and other bookkeeping information—must be eventually put into a message and sent to the remote callee, placing this information into a packet to begin with makes sense. The code sequence for a remote call thus obtains a free packet, rather than an *argumentRecord*, and marshals the parameters directly into it. The actual code that the compiler generates for an Emissary call is given in figure 5.3. The code for a return, which is similar, is shown as well. *Q's* code resides in node *Qnode*, *P's* is in *Pnode*. The details of the implicit packet *myProcess.pkt* (shown *explicitly* in the figure) are discussed below.

The *AllocPkt* operation gets a free packet and declares it to be a *call* or *return* packet. For efficiency, a pointer to the allocated packet is stored both in the *ProcessObject* of the executing process and in a machine register. The packet is therefore always known to the compiler, although its implicit use is explicitly indicated by *myProcess.pkt* in the example. The *Marshal* and *Unmarshal* operations are shown as procedures, but in practice it is vital that they be open-coded, i.e., compiled inline. Further discussion of marshaling is postponed until a later section. The *ClientCall.RemoteCall* operation is an abstract machine function implemented by *Qnode's* RPC

mechanism. It causes a remote call of *P* by sending *Q's call* packet (*myProcess.pkt*) to *Pnode*. In *Pnode*, *P's* procedure body is invoked by the corresponding RPC mechanism there, and *P's* RETURN to *Pnode's* RPC mechanism causes a *return* packet to be sent back to *Qnode*. *Q's RemoteCall* then completes. The exact details of these operations are explained shortly.

```
The code in procedure Q, in Qnode, that makes a remote call to P:
[result_1, ..., result_n] ← P[argument_1, ..., argument_n];
        myProcess.pkt ← AllocPkt[myProcess, call];
        Marshal[myProcess.pkt, argument_1, ..., argument_n];
        ClientCall.RemoteCall[myProcess, P, myProcess.pkt];
        Results come back in myProcess.pkt.
        Unmarshal[myProcess.pkt, result_1, ..., result_n];
        FreePkt[myProcess, myProcess.pkt].

The body of remote procedure P, in Pnode:
P: REMOTE PROCEDURE [argument_1, ..., argument_n] RETURNS [result_1, ..., result_n]
        Unmarshal[myProcess.pkt, argument_1, ..., argument_n];
        FreePkt[myProcess, myProcess.pkt];
        ...
        Do work of procedure...
        ...
        myProcess.pkt ← AllocPkt[myProcess, return];
        Marshal[myProcess.pkt, result_1, ..., result_n];
        RETURN.
```

Figure 5.3: The compiled code sequences for a remote call of *P* by *Q*.

Notice that the procedure value used by *Q* to identify *P* must contain *Pnode* in the remote case. The *procedure descriptor* for *P* is therefore a record with two components:

```
ProcedureDescriptor: TYPE = RECORD [
        node: Node, -- Node that inport's procedure context lives in.
        inport: ProcedureInport ].
```

The *node* component is ignored in local calls (and, as an optimization, may not even be present for them). The *inport* component identifies the procedure to be called, e.g., in some implementations it is just the address of the procedure body.

5.4.1.2 Local and Remote Transparency

Before elaborating the complete specification of Emissary's RPC runtime machinery, pausing to consider the semantics and transparency of the suggested call sequences and their supporting runtime operations is valuable. Doing this in terms of the relevant essential properties is a natural approach.

> *Exactly-once semantics.* The orphan algorithms of section 5.3 give last-one semantics in the presence of crashes. In the absence of crashes, exactly-once semantics demand that the *call* and *return* packets that are exchanged by the RPC mechanism be sent reliably. In the previous example, this requires that *Pnode* execute *Q's* call only once—even though *call* packets can be received many times—and that *Pnode* retain and retransmit the *return* packet until *Qnode* acknowledges receipt of the *return*. This acknowledgement can happen

implicitly, as the result of a later call from *Q* to *Pnode*, or explicitly, because *Pnode's* RPC mechanism requests an acknowledgement from *Qnode's* mechanism.

Typechecking. The typesafety of *Q's local* calls to *P* is guaranteed by the local binder when *P's* and *Q's* modules are bound together. The binder does this by checking that the interface exported by *P's* implementation is of the exact type that *Q* imports. For remote calls, typechecking is guaranteed by the *remote* binder, which performs precisely the same checking for *remote* interfaces. Remote binding is covered in section 5.5.

Parameter functionality. For local calls, restrictions on the types of *P's* arguments and results are determined solely by the programming language. For remote calls, however, the restrictions are established by the *Marshal* and *Unmarshal* operations that will be discussed in section 5.4.3. These restrictions are due primarily to address-containing types, and their consequences were discussed in section 4.1.4.

Concurrency control. The remote call code sequences in the example satisfy the concurrent invocation policy of section 4.1.1.7: In *Qnode*, the *RemoteCall* operation does not return until the call is complete. *Q* is therefore blocked for the duration of the call, as intended. In *Pnode*, the receiving RPC mechanism assigns a special process to execute *P* on behalf of *Q*. This process competes for procedure *P* asynchronously with all other processes, as intended.

Syntactic transparency. The suggested calling sequences for local and remote procedures are incompatible. This violates transparency because it requires that the declaration of a remote procedure carry the explicit attribute REMOTE. Notice, however, that this attribute appears in precisely one place, the point of declaration. It does *not* appear anywhere the procedure is called because the compiler, knowing the remote nature of the call from the declaration, can compile the right code without any notification from the programmer. Except for the declarations, then, programs are written in a location-independent manner. The only problem this nontransparency causes is an efficiency loss when a REMOTE procedure is called from within its own node. For example, if procedure *R* in *Pnode* calls *P*, then the RPC mechanism is invoked needlessly. One way to overcome this problem is to compile two entry points for remote procedures—one for local calls that use the stack, and another for remote calls that use a packet. Analogously, each *invocation* of a remote procedure is compiled to select the right entry point: by looking at the *node* field in the procedure descriptor, either a local call is made or *RemoteCall* is used. This approach, which is not developed further here, is valuable when the same procedures will be heavily called both locally and remotely.

5.4.1.3 Emissary's Runtime Mechanism

With this background on call sequences and the constraints of transparency, we now turn attention to the implementation of the *RemoteCall* operation. A complete Mesa program for *RemoteCall* and all other Emissary runtime routines appears in algorithm 5.3, which is presented in section 5.4.6. The program is divided into six parts.

Declarations. Type declarations set the stage for the program.

NetworkService module. NetworkService contains network I/O and packet buffer routines.

ClientCall monitor. ClientCall's operations manage the client node's half of a remote call.

ServerCall monitor. ServerCall's operations manage the *activeCalls* set of *call* and *return* packets. *ServerCall* is mainly used by the server node.

RpcServerProcesses. These processes actually execute remote calls in the server node. *RpcServerProcesses* primarily use operations in *ClientCall* and *ServerCall.*

Connection monitor. Connection's operations manage the set of RPC *connections* with other nodes.

At first glance the reader many find the Emissary program overwhelming. Its basic behavior, however, can be readily understood by reading the description that appears below. The complete algorithm is presented because RPC runtime machinery is at the heart of any remote procedure scheme: the success and performance of an overall RPC system hinges on the crucial design decisions made for the runtime machinery. A detailed specification of the algorithm is vital so that potential RPC implementors can see exactly how to construct an efficient and transparent RPC mechanism. On the other hand, readers uninterested in such precise details can skip the algorithm with little loss of continuity.

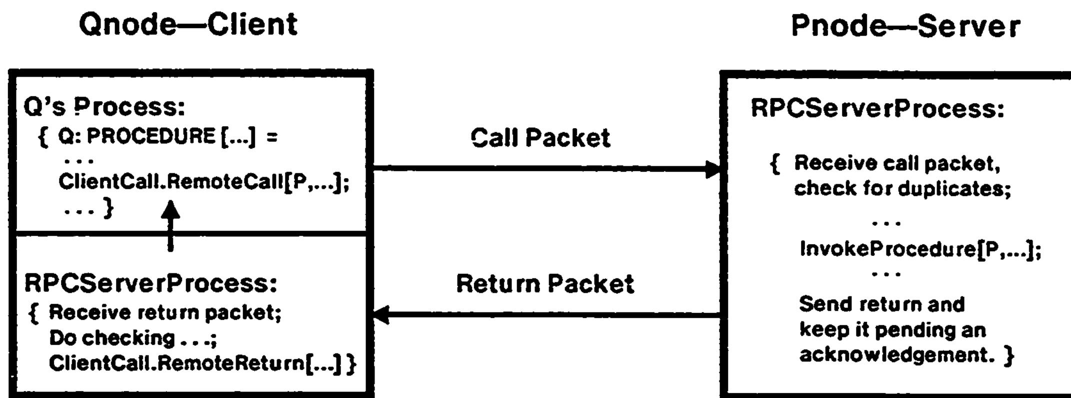


Figure 5.4: An overview of an Emissary remote call from *Qnode* to *Pnode.*

A good understanding of Emissary's machinery is gained by tracing the flow of a single remote call through the algorithm. Starting with the client remote call sequence in figure 5.3, a remote call of procedure *P* proceeds as follows. Figure 5.4 gives a top-level overview of the call.

In *Q's* process in the client *Qnode, Q* assembles a *call* packet into *myProcess.pkt* and invokes *P* by calling *ClientCall.RemoteCall.*

In *Qnode's ClientCall* monitor (in algorithm 5.3), *RemoteCall* ensures that *P.node* is in *nodesOut* as required for extermination and crash recovery. The *call* packet (still in *clientProcess.pkt*) is then transmitted to the server node, *Pnode.* If no *return* packet is received (by *ClientCall.RemoteReturn,* below) within *callRetransmitInterval,* the call is retransmitted. If no *return* is received after *retriesUntilFailure* retransmissions, the *Failed* exception is signalled.

In the server, *Pnode, P's call* packet is handled by a special *RpcServerProcess* (see *RpcServerProcessPrototype*) whose only job is to execute remote calls quickly. In *RpcServerProcess,* when the hardware receives an RPC packet in *inPkt* it NOTIFYs the WAITing server process directly through the preinitialized *rpcPktReceived* condition variable. This technique starts running an *RpcServerProcess* with only one process switch.

Continuing in *Pnode*, the *RpcServerProcess* for *P* first sets up another *RpcServerProcess* (called *nextHandler*) to receive the next RPC packet. *P's* server process then checks the generation of the just-received *call* packet by calling *Connection.Verify* (generations are discussed below). *Verify* discards old, delayed, and expired packets.

Continuing in *P's RpcServerProcess*, *call* packets that pass *Verify's* tests are checked for duplicates and implicit acknowledgement by *ServerCall.CheckDuplicatesAndAck*. Finally, if the *call* packet is new, *InvokeProcedure* invokes the body of procedure *P* in *Pnode* with arguments passed in *myProcess.pkt*. (The code for *P's* body is in figure 5.3.)

Still in *Pnode*, when *P* returns to its *RpcServerProcess*, *P's return* packet is immediately transmitted back to *Qnode*. A record of the *return* packet is kept in the *activeCalls* list of the *ServerCall* monitor until *P* acknowledges the *return* (acknowledgements are discussed below). This allows duplicate calls to be detected and ignored, and permits retransmission of the *return* if it is not received by *Qnode*.

Back in the client, *Qnode*, the *return* packet is received by one of *Qnode's RpcServerProcesses*. Preliminary processing of the *return* packet is exactly the same as the *call* packet's was above. Once the *return* passes the *Verify* test, it is passed to *ClientCall.RemoteReturn*.

Still in *Qnode's RpcServerProcess*, *RemoteReturn* checks to see if the original caller (in *RemoteCall*) still wants the *return* packet. If yes, *RemoteReturn* stores it in *clientProcess.pkt* for the caller and NOTIFYs *RemoteCall* (which is WAITing in *Q's* client process, *not* in *RemoteReturn's RpcServerProcess*). If the *return* is not wanted, *RemoteReturn* assumes the (old) *return* is a request for a *return* acknowledgement and transmits a *returnAck* packet back to *Pnode*.

Finally, in *Qnode*, *Q's RemoteCall* ends its patient WAIT and passes the *return* packet (still in *clientProcess.pkt*) back to the compiled invocation of procedure *P* (figure 5.3). *Q's* remote call of *P* is complete, and under normal conditions exactly two packets were exchanged: one *call*, and one *return*.

5.4.1.4 Some Fine Points of Emissary's Design

The previous trace of *Q's* call to *P* necessarily left out some important details. Some finer points are now discussed by retracing a few of the call's steps.

Serial Numbers. Every RPC packet contains a *Packet.rpc.callID* field that identifies the node, process, and serial number of the packet. This information is sufficient to identify the packet in the absence of crashes as long as the serial number does not overflow. This constraint on the *SerialNumber* TYPE is necessary so that all of the *call* packets sent in a process's between-crash lifetime are unique. Otherwise, delayed and duplicated *call* packets, or unacknowledged *return* packets, could be mistaken for new *call* and *returns* (i.e., those with overflowed serial numbers). In practice, a *call* that has been outstanding for a *very* long time is assumed to be *Aborted* (expired), and a packet that has been delayed for a similar length of time is assumed to be dead. This time limit can be used to set a practical upper bound on the size of *SerialNumber*, but operational experience indicates quite strongly that 16 bits is not sufficient—24 or perhaps even 32 are recommended. (If a process makes a remote call every millisecond, a 16-bit *SerialNumber* will overflow in about 1.1 minutes; 24-bit, about 4.5 hours; 32-bit, about 1200 hours.)

Since serial numbers are kept in volatile storage, the scheme above will fail whenever a node crashes. The Emissary algorithm uses the *Packet.rpc.generation* number to solve this problem. The *generation* of a node is a STABLE counter that is increased after every crash of *myNode*. As with *SerialNumber*, the *Generation* type must not overflow in the entire hardware lifetime of a node. In practice, the lifetime of a remote call (or of a long-delayed packet) set a reasonable upper bound on the size of *Generation*.

An important implementation trick often used to implement stable counters is simply to use the system time. This works as long as the time is monotonically increasing, has a sufficiently fine grain (at least seconds), and will not expire (overflow) sufficiently far into the future.

Composite Serial Numbers. The composite serial number of a *Packet* is the pair (*rpc.generation, rpc.callID.serial*). The additional pair (*rpc.callID.node, rpc.callID.process*) identifies the process that sent the packet. The advantage of the two-part (*generation, serial*) scheme is efficiency: *serial* is volatile and is updated on each call at trivial cost; *generation* is stable and is updated expensively, but very infrequently.

Node-to-node Connections. Most of the *Connection* monitor was not discussed in the Emissary walkthrough. The function of *Connection* is to keep an accurate record of (*node, generation*) pairs for all of the nodes in *nodesIn*—that is, all of the nodes to which *myNode* has outstanding remote calls. This set of pairs is called *connections* and is used by *Verify* (via *CheckConnection*) to identify delayed RPC packets from previous generations. There is at most one connection in *connections* for any pair of nodes, not one connection per call per pair. (More closely coupled process-to-process connections, which are needed for reliable transmission and exactly-once semantics, are discussed below.)

Emissary creates connections *after* communication is attempted, but *before* it is allowed to proceed (unlike many schemes that establish a connection before communication can start). *Verify* ensures this discipline as follows: When *Verify* receives a packet from a node, *new*, that is not in *connections*, it ignores the packet and calls *EstablishConnection* to create a connection with *new*. *EstablishConnection* does this in two steps: it first makes a remote *generationRequest* call to *new* that returns *new's* generation, and it then has *AddConnection* add the pair (*new, new's generation*) to *connections*. The *HandleTraffic* routine in node *new* never ignores *EstablishConnection's* special *generationRequest* call from *myNode*, even if *new* does not yet have a connection with *myNode*. This bootstrapping method prevents deadlock when connections are first established. Once a connection to *new* is complete, communication can proceed.

Acknowledgements. The Emissary algorithm acknowledges *return* packets in two ways (acknowledgements are necessary for exactly-one semantics).

> *Implicit acknowledgement.* An implicit acknowledgement of *Q's* call to *P* on *Pnode* occurs when the client process (*Q's* process in *Qnode*) makes another call to the server node (*Pnode*). In this case, *ServerCall.CheckDuplicatesAndAck* uses *ServerCall.FindOtherCaller* to locate any *oldCallPkt* from *Q's* process which is waiting for an acknowledgement. If so, *CheckDuplicatesAndAck* deletes the *oldCallPkt* from *activeCalls* since it is acknowledged by

> the *newCallPkt*: the client's new call cannot have started until the old one finished because remote calls are synchronous. (The *oldCallPkt* can also be an already-acknowledged *return* packet that serves to maintain a process-to-process connection; this is discussed below.)
>
> *Explicit acknowledgement.* Explicit acknowledgements are requested by the special *ReturnRetransmitter* process. This process uses *ServerCall.RequestAckIfNeeded* to retransmit *return* packets that have not yet been implicitly acknowledged. *ClientCall.RemoteReturn* handles these *return* packets as described in the first walkthrough.

Thus a typical remote call takes only two packets—*call* and *return*—because of the assumption that there will be a succeeding call that implicitly acknowledges the earlier one. When there is no later call, *ReturnRetransmitter's* explicit acknowledgement strategy sends two more packets: an extra *return*, and its answering *returnAck*. This scheme places all of the state information needed for *return* acknowledgement processing in the server, and it is symmetric in the following sense: client nodes have responsibility for retransmitting *calls* until they receive a *return*; server nodes have responsibility for retransmitting a *return* until they receive a *returnAck* (unless, of course, the *return* is implicitly acknowledged).

This two-packet acknowledgement strategy can be replaced by a one-packet strategy that works most of the time. In the one-packet scheme, the *client* node assumes responsibility for retransmitting a *returnAck* to the server whenever the client does not implicitly acknowledge a call within a given time. The problem with this scheme is that the client's *returnAck* can be lost, so the server must still be prepared to handle the whole acknowledgement procedure from his end. Because of this, the asymmetric three-packet scheme seems unnecessarily complicated and unattractive.

Process-to-process Connections. The node-to-node connection scheme described above eliminates only packets from incorrect generations. To ensure reliable transmission—and thus exactly-once semantics—between calling processes and *RpcServerProcesses*, a finer grain connection scheme is required. In algorithm 5.3, this is accomplished by having the server's RPC machinery keep a record of the last call from each remote client process. The simple method used is to have *ServerCall.CallAck* keep, in *activeCalls*, the last *return* packet sent in response to each client *call*. Because all packets in *activeCalls* have a complete *CallID* in *packet.callInfo.source*, these *CallIDs* serve as connection identifiers that allow *ServerCall.CheckDuplicatesAndAck* to eliminate duplicate and delayed packets from client processes.

There is a problem with this method as it is implemented in the algorithm: Valuable packet buffers are wasted because acknowledged, connection-maintaining *return* packets remain (largely) unused. There are several attractive solutions to this space-efficiency problem:

> *Keep just the CallID.* One solution is to have *ServerCall.CallAck* release the *return* packet and keep just *packet.callInfo* in *activeCalls*. This can be done by redeclaring *PacketObject* to be a variant record with a possibly empty body (i.e., empty non-*callInfo* part).
>
> *Do not use activeCalls.* Another solution is not to use *activeCalls* to keep track of process-to-process connections. Instead, a separate structure such as a *processConnections* set of client-process *CallIDs* can be used.

Use timed connections. A final solution is to continue to use *activeCalls* to hold the connection identifiers, but to automatically delete both the *return* (or a shorter variant, as above) and the connection after a suitable time interval. The interval must be sufficiently large that duplicate or delayed packets from a client process can no longer exist. Section 5.4.1.5 discusses this further in the context of indefinite acknowledgements.

Orphan Algorithm Interactions. The interactions between Emissary's runtime machinery and the orphan algorithms of section 5.3 merit explicit comment. There are several interchanges of information and one interaction of control.

Information—nodesIn and nodesOut. Extermination depends on the *nodesIn* and *nodesOut* sets. The *nodesIn* set is updated by *Connection.AddConnection*, and the *nodesOut* set is updated by *ClientCall.RemoteCall*. (The removal of nodes from these two sets is left to an aging scheme as discussed previously.)

Information—Process.worker and Process.parent. In addition to *nodesIn* and *nodesOut*, extermination also depends on the *Process.worker* and *Process.parent* information in *ProcessObjects*; this information is used by *Exterminate* to follow the multinode stack of a remote call. The *worker* field in client processes is maintained by *ClientCall.RemoteCall*, and the *parent* field in server processes is maintained by the *RpcServerProcesses*.

Information—Process.expirationTime. The expiration algorithm depends on *Process.expirationTime.* The *expirationTime* of a *call* packet is set by *NetworkService.AllocPkt*, and the *expirationTime* of an *RpcServerProcess* executing a remote call is set by the *RpcServerProcess*. As an optimization, *Connection.Verify* also checks the *expirationTime* of arriving *call* packets.

Control—RpcServerProcesses. Both extermination and expiration kill an orphaned remote call by aborting all of the call's child processes. When an *RpcServerProcess* is *Aborted*—during an extermination or expiration—it catches the corresponding *Aborted* exception and finalizes itself before being destroyed. This finalization is shown in *RpcServerPrototype*, where the server process deletes and frees its (potential) *call* packet from *activeCalls*.

5.4.1.5 Omitted Details in Emissary's Algorithm

The Emissary mechanism presented in algorithm 5.3 has a few missing parts. These parts are intentionally omitted to simplify the presentation and exposition of the algorithm; it is complicated enough as it is. In a real implementation, however, these parts must be present for smooth RPC operation. They are now briefly discussed.

Multiple Packets. Most networks impose a limit on the maximum size of a packet, and hence remote calls with sufficiently large parameter records will not fit into exactly one packet. Emissary's algorithm has no provision for this situation. A sketch for one simple solution is the following: Add two new packet types to *RpcPacketType*, *longCall* and *longReturn*. For a remote call (or return) that takes *n* packets, the first *n*-1 are transmitted as *longCall* (or *longReturn*), and the last packet as *call* (or *return*). These *n* packets are linked together and the head of their list is stored in the *ProcessObject*, in place of (or in addition to) the *Process.pkt* field.

Handling multiple packets is complicated and demands extremely careful design if the performance of the vital one-packet case is not to suffer. Having two parallel internal implementations—the existing fast one, and a general slow one—is a recommended starting position.

Flow Control and Congestion Control. The transmission strategies used in algorithm 5.3 have no provision for flow control or congestion control [8]. In the context of RPC, *flow control* is controlling the rate at which *call* packets are sent to a given node, and *congestion control* is controlling the rate at which RPC packets are sent over an internet. *ClientCall.RemoteCall*, in particular, retransmits *call* packets every *callRetransmitInterval* whether the packet has been received or not. When *call* retransmissions are actually unnecessary, this is precisely the wrong strategy because it aggravates rather than alleviates the traffic problem. For example, when a call has been received but its execution takes longer than *callRetransmitInterval*, flow control can prevent redundant *call* packets from being resent and discarded as duplicates. Furthermore, if a *call* packet has not been received because of transit delays due to slow or congested links, then congestion control can prevent retransmissions from occurring too quickly and adding to the congestion.

Adaptive retransmission schemes are the usual solution to flow control and congestion control problems. These methods use dynamic delay measurements and low-level control messages to determine appropriate retransmission intervals. While a discussion of adaptive retransmission is beyond the scope of the dissertation, its complexity can be quite high, and it is typically found only in level 2 (and above) mechanisms such as bytestreams. For this reason, any adaptive retransmission mechanism for Emissary must be carefully designed not to interfere with the performance of steady-state calls, especially local network calls where congestion is never a problem (but flow control can be).

Indefinite Wait for Acknowledgements. As written, the *ReturnRetransmitter* process will resend *returns* forever if no *returnAck* acknowledgements are received for them. Under normal conditions this will never happen, and indeed even if a caller crashes the callee's *return* will be destroyed when the caller's node recovers and exterminates the call (assuming the *return* was never received). But if a caller stays down, or the network partitions, then the *return* will linger until communication is reestablished with the down (partitioned) node.

In practice, *returns* consume valuable buffer space and a large time limit may have to be set on the time waited for a *returnAck* to be received. If this time limit expires and the *return* is destroyed before the caller gives up and declares the call *Failed*, a retransmitted *call* packet cannot be detected as a duplicate. This can violate exactly-once semantics. Instead, last-one semantics *without crash notification* are given.

If the caller has crashed then destroying his *return* has no adverse affect. If, on the other hand, the caller is patiently waiting for a temporary partitioning to end, destroying his *return* will have disastrous last-one consequences. Making the acknowledgement time limit *much longer* than the

waiting time that elapses before *Aborting* a call is therefore imperative. In *ClientCall.RemoteCall*, this elapsed time before *Aborting*—actually, before raising *Failed*, which precedes *Abort*—is fixed at *retriesUntilCallFailure*callRetransmitInterval*. In a real system where clients can specify their own call time limits, the acknowledgement time limit must be either dynamic or longer than any (reasonable) call time limit.

The Failed Exception. When *ClientCall.RemoteCall* signals *Failed* because it has not received a return, no information is conveyed to the catcher of the exception other than the fact that the call has timed out. In section 4.1.5.4 it was suggested that specific information should be returned with exceptions, not just the low-level indication given by *Failed*. *RemoteCall* can easily be changed to do this by calling a special *DetermineAndSignalCallFailure* routine instead of signalling *Failed*. *DetermineAndSignalCallFailure* interrogates the network software, the server node (if possible), and perhaps other entities to determine more specific information. It then signals

```
Failed: SIGNAL [failureCode: FailureCode];
FailureCode: TYPE =
  {networkPartitioned, serverDown, noCallsAllowed, clientTimeLimitExpired, ..., unknown}.
```

FailureCode provides the desired information.

5.4.1.6 Performance Considerations in Emissary's Design

Emissary's RPC machinery is designed to be efficient as well as transparent. Some of the performance considerations included in the design are directly integrated into the structure of algorithm 5.3; other optimizations can be realized only in the actual low-level implementation of the algorithm. This section discusses some important performance issues in the Emissary design.

Hardware Packet Demultiplexing. In the *NetworkService* module, the *StartRpcReceive* operation controls only RPC packets—that is, only packets with *Packet.transport.type=RPC* or *Packet.internet.type=RPC*. This is arranged by having the hardware or microcode implementation of the network interface demultiplex predefined classes of packets automatically. Thus *StartRpcReceive* controls the delivery only of arriving RPC packets; the delivery of other packet types is controlled by analogous *Start*Type*Receive* routines. This hardware demultiplexing is essential because it removes the burden from software, where an expensive process switch is usually required.

Minimizing Process Creation and Process Switching. Emissary usually creates no new processes for a remote call because both the client's and server's *rpcHandlers* caches will be nonempty. Further, Emissary performs only three process switches for a roundtrip remote call: one in the server node to receive and execute the *call* packet, one in the client node to receive the *return* packet, and another in the client to restart the waiting caller. This small number of switches is obtained by having the precreated *RpcServerProcesses* be started *directly* by the network hardware's *rpcPktReceived* interrupt. There are no intervening operating system processes; there is no unnecessary demultiplexing. Even in Mesa, which has very efficient process machinery, process creation is relatively expensive and process switching (WAIT-NOTIFY) is not cheap. For performance, then, avoiding these operations is essential, and Emissary works hard to eliminate them.

Minimizing Data Copying. Copying large blocks of data such as packets has significant overhead in any high performance system. Emissary itself performs no packet copying, and the *Marshal* and *Unmarshal* operations to be discussed in section 5.4.3 copy a parameter record only once, between a procedure's activation record and a *call* (or *return*) packet.

Inline Implementation of Frequent Operations. Performance improvements can be made when the most commonly executed Emissary code is compiled inline or put into microcode. Some of these situations are indicated in algorithm 5.3, where procedures such as *FreePkt* are declared with INLINE bodies for inline expansion. More generally, algorithm 5.3 must have a microcoded implementation of the instruction sequences that are executed by a *typical* roundtrip call. This typical call *Transmits* exactly one *call* and *return* packet, is immediately dispatched by waiting *RpcServerProcesses* in both the client and the server, and takes place between two nodes that already have mutual connections established. If a call that begins its life by executing in this RPC microcode violates any of these conditions, it immediately traps back to a software layer that handles all of the nontypical conditions. Microcoding the typical case is especially attractive for language systems that already have a microcoded implementation of the *Transfer* primitive. Integrating the remote call microcode with the local *Transfer* microcode can produce an *RTransfer* implementation that easily achieves high performance.

Reduced Local Network Overhead. In the *NetworkService* module, the *Transmit* and *StartRpcReceive* operations give special service to local network packets, i.e., those addressed to other nodes on the directly connected network. Chapter 6 shows that for local network calls, explicitly using the local network transport mechanism, rather than the general datagram mechanism, is an extremely valuable optimization.

5.4.2 General *RTransfers*

The preceding section presented Emissary's approach for performing remote procedure calls. Procedures were used as the basis of the runtime mechanism because they are by far the most frequent control transfer primitive, and putting optimization effort into the most crucial common case is the only sensible thing to do. But just as focusing on optimization is important for performance, developing a fully general *RTransfer* mechanism (section 2.1.3.1) is vital for uniformity. Otherwise, remote exceptions, coroutines, and so forth cannot be implemented, and remote versions of these primitives are necessary for transparency. Fortunately, *RTransfers* can be built on top of remote procedures quite conveniently—*RTransfer* is simply implemented with a remote procedure call.

For example, in the source node of an *RTransfer*,

RTransfer(*destinationInport, returnOutport, argumentRec*)

is readily compiled into the following. The implicit packet *myProcess.pkt* is again shown explicitly.

```
myProcess.pkt ← AllocPkt[myProcess, call];
Marshal[myProcess.pkt, destinationInport.context, returnOutport, argumentRec];
ClientCall.RemoteCall [
    myProcess, [node: destinationInport.node, inport: RTransferHandler], myProcess.pkt ];
FreePkt[myProcess, myProcess.pkt];
SuspendContextAndProcess[].
```

In the *destinationInport* node, the remote call is executed by a special *RTransferHandler* procedure that performs the remote part of the *RTransfer*.

```
RTransferHandler: PROCEDURE =
  { Unmarshal[myProcess.pkt, destinationInport.context, returnOutport, argumentRec];
    Perform an asynchronous Transfer (destinationInport.context, returnOutport, argumentRec);
    RETURN }; -- Return immediately, before the Transfer finishes.
```

There are two important points to observe:

> The asynchronous *Transfer* in *RTransferHandler* allows the caller's *RemoteCall* to complete quickly so that the caller can suspend itself, pending a new *RTransfer* on its *inport*. This is necessary because, in general, *returnOutport* will not specify a return to the source of the *RTransfer*. In this case, the caller must be prevented from waiting for a *return* message that will never come. This asynchronous behavior does not violate the standard concurrency property because the caller blocks (at *SuspendContextAndProcess*) as soon as the *RTransferHandler* call completes.
>
> Each *RTransfer* requires a minimum of two packets—one for the *RTransferHandler call*, and one for its null *return*. In a situation where node *A RTransfers* to node *B*, which in turn *RTransfers* to *C*, this is the best possible behavior. It is most likely, however, that *B* will transfer back to *A*. In this case, at least four packets will be sent when only two are strictly necessary. This is precisely the reason that procedure call—the most common *RTransfer*—was specially implemented. To perform efficient general *RTransfers* between a pair of interacting machines, it is easy to build a synchronous two-packet *RTransfer* mechanism using the Emissary procedure mechanism as a model. The basic change required is that RPC *PacketObjects*, in addition to having a *Packet.rpc.inport* field (*destinationInport.context*, above), are augmented with a *Packet.rpc.outport* field (*returnOutport.context*). The *outport* field specifies the context in the caller's original process which is to be resumed on return. In the previous procedure machinery, this context is implicitly the one invoking *ClientCall.RemoteCall*, e.g., procedure *Q* in figure 5.4.

In most Algol-like programming languages, nonprocedural language-level communication—raising exceptions, transferring between coroutines, initializing and finalizing modules—always occurs between pairs of modules. In the remote case, then, the *RTransfer* implementation of these primitives is between a pair of machines as in the latter point, above. In languages where nonprocedural transfers are used extensively, it may be desirable to implement the special two-machine *RTransfer* mechanism rather than simply map the remote versions of other control transfers into procedure calls.

For the same reason that remote procedures must be declared REMOTE, the remote versions of other control transfers must also have the REMOTE attribute. For example, the declarations of REMOTE SIGNALS, REMOTE PORTS (coroutines), REMOTE PROGRAMS, and so on. This allows the compiler to treat them properly.

5.4.3 Marshaling Parameters

A simple example of marshaling parameters was given in section 2.2.5.1. This section elaborates on that example by describing how to compile code for marshaling and unmarshaling general parameter records. The scheme used is similar to the one used by compilers to compile arguments and results for local procedure calls. The major differences for remote calls are that parameters are compiled into a packet, not a stack or *argumentRecord*, and that address-containing types are usually flattened, not transmitted as pointers.

The marshaling approach presented in this section is based on my experience in implementing two operational marshaling mechanisms. Each mechanism is incorporated into a stub translator and generates code that executes correctly.

5.4.3.1 Basic Operations

To move data between parameters and packets two copying operations are useful:

> *CopyToPkt*: PROCEDURE [*pkt*: *Packet*, *paramAddress*: POINTER, *paramSize*: CARDINAL];
> *CopyFromPkt*: PROCEDURE [*pkt*: *Packet*, *paramAddress*: POINTER, *paramSize*: CARDINAL].

CopyToPkt copies *paramSize* words starting at *paramAddress* into *pkt*. Since the base of the parameter part of *pkt* is *pkt.rpc.parameters*, and its next free location is at *pkt.rpc.parameters+pkt.rpc.paramLength*, *CopyToPkt* moves *paramSize* words to this latter location and increases *pkt.rpc.paramLength* by *paramSize*. By adjusting *pkt.rpc.paramLength* in this way, each succeeding call of *CopyToPkt* packs its new parameter right after the preceding one in *pkt*. Similarly, *CopyFromPkt* moves *paramSize* words from *pkt* to *paramAddress*, increasing *rpc.pkt.paramLength* by *paramSize*. It is extremely important that these two copy operations be efficient. If not completely implemented in microcode, they should at least compile open and use the hardware's fastest block transfer instruction. A chain of *CopyToPkt* or *CopyFromPkt* calls should be further optimized by keeping *pkt's parameters* and *paramLength* fields in special registers, and storing them only at the end of the chain.

In this section, the details of large parameter records that require multiple packets are not considered. Multiple packets can be handled easily by changing the implementations of *CopyToPkt* and *CopyFromPkt* to detect packet boundaries and automatically switch to the next packet.

5.4.3.2 Marshaling Specific Types

The marshaling of common Mesa datatypes is now described. Unless otherwise stated, unmarshaling is performed in the same way as marshaling except that *CopyFromPkt* is called instead of *CopyToPkt*. Since marshaling complexity lies in the treatment of pointers, this discussion focuses attention on address-containing types. The general method used to marshal types which compose new types from existing ones—e.g., records and arrays—is to marshal their subtypes recursively. Since marshaling *always* begins with a parameter record, the *Marshal* and *Unmarshal* operations used in figure 5.3 actually take an argument or result record as input, not the individual arguments or results themselves.

INTEGER, CARDINAL, UNSPECIFIED, ***Enumeration, Subrange***, BOOLEAN. Mesa's basic built-in types are marshaled by calling *CopyToPkt* with their their bitstring representations. For example, for an INTEGER *i*,

CopyToPkt[*pkt*, @*i*, SIZE[INTEGER]].

ARRAY ***IndexType*** OF ***ElementType***. In Mesa, arrays have a fixed length that must be manifest at compile time. Because this size is always known, arrays are represented by a fixed block of storage. If an array's *ElementType* does not require marshaling, then it is marshaled simply by treating it as an uninterpreted block. For example, for *array*: *ArrayType*,

CopyToPkt[*pkt*, @*array*, SIZE[*ArrayType*]].

If an array's elements do require marshaling, then the array is marshaled with a loop that handles each element. For example,

FOR *element* IN *array* DO *Marshal*[*array*[*element*]].

DESCRIPTOR FOR ARRAY OF ***ElementType***. Dynamic-length arrays are implemented with array descriptors in Mesa. The representation of a descriptor is given approximately by *DescriptorType* record, below, although the *BASE* and *LENGTH* fields of the record are accessed with the BASE and LENGTH operators.

DescriptorType: TYPE = RECORD [*BASE*: POINTER, *LENGTH*: CARDINAL].

If the *ElementType* of a descriptor does not need marshaling, then the descriptor is marshaled by simply sending its length and elements in two parts. For example, for *desc*: DESCRIPTOR,

CopyToPkt[*pkt*, @LENGTH[*desc*], SIZE[CARDINAL]];
CopyToPkt[*pkt*, BASE[*desc*], LENGTH[*desc*]*SIZE[*ElementType*]].

Unmarshaling must proceed in two steps since *desc's* length is dynamic: first, the length must be retrieved and storage allocated for all the elements; second, the elements must be read into the new storage. Inverting the previous marshaling thus proceeds as follows. The *base* and *length* variables are assigned to registers.

base: POINTER; *length*: CARDINAL;
CopyFromPkt[*pkt*, @*length*, SIZE[CARDINAL]];
base ← *AllocateStorage*[*length**SIZE[*ElementType*]];
CopyFromPkt[*pkt*, *base*, *length**SIZE[*ElementType*]];
desc ← DESCRIPTOR[*base*, *length*].

If a descriptor's elements do require marshaling, then a loop is required as with fixed arrays. In addition, the receiver must still allocate the empty space that will be filled by the element-by-element unmarshaling. These changes result in the following code:

-- Marshaling for an array descriptor with address-containing elements:
CopyToPkt[*pkt*, @LENGTH[*desc*], SIZE[CARDINAL]];
FOR *element* IN [0..LENGTH[*desc*]) DO *Marshal*[*desc*[*element*]].

-- Unmarshaling for an array descriptor with address-containing elements:
base: POINTER; *length*: CARDINAL;
CopyFromPkt[*pkt*, @*length*, SIZE[CARDINAL]];
base ← *AllocateStorage*[*length**SIZE[*ElementType*]];
desc ← DESCRIPTOR[*base*, *length*];
FOR *element* IN [0..LENGTH[*desc*]) DO *Unmarshal*[*desc*[*element*]].

HANDLE. In its simplest form, a handle identifies an abstract object that is never changed remotely (except, of course, by procedure calls on its abstract operations). The usual

representation for a handle is an encapsulated pointer that is passed around remotely but is never used (dereferenced) in any address space but its original one. Handles can therefore be treated exactly like the UNSPECIFIED built-in type. A handle is an address-containing type that does *not* need marshaling.

PROCEDURE ***and other control transfer types.*** Procedures and other control transfers are represented by full descriptors that specify their resident *nodes* as well as intranode *inports* (section 5.4.1.1). Control transfer types are therefore marshaled simply be sending their uninterpreted descriptors (but these transfer types must have the REMOTE attribute, as discussed in section 4.1.4.3). For example, for the procedure descriptor of procedure *P*,

CopyToPkt[*pkt*, @*P*, SIZE[*ProcedureDescriptor*]].

Marshaling transfer types is also discussed in section 5.4.4.3.

STRING. As covered in section 2.2.5.1, Mesa strings are represented by a pointer to a *StringBody*:

```
StringBody: TYPE = RECORD [
        length, maxlength: CARDINAL,
        text: PACKED ARRAY [0..0) OF CHARACTER].
```

The best way to marshal a string *s*: STRING (*s*: POINTER TO *StringBody*) is by calling

CopyToPkt[*pkt*, @*s*, 2+(*s.length*+1)/*CharsPerWord*].

This sends just the meaningful part of *StringBody*—*s.length*, *s.maxlength*, and *text*[0..*s.length*). Furthermore, it sends them all in one operation, not in the three pieces *length*, *maxlength*, and *text*. This is very efficient. Unmarshaling a string is done similarly, but space for *StringBody* must be dynamically allocated before *StringBody* is filled by *CopyFromPkt*. The following sequence unmarshals *s* properly. In practice *temp* is assigned to registers.

```
temp: RECORD [length, maxlength: CARDINAL];
CopyFromPkt[pkt, @temp, 2];
s ← AllocateStorage[2+(temp.maxlength+1)*CharsPerWord];
s.length ← temp.length; s.maxlength ← temp.maxlength;
CopyFromPkt[pkt, @s+2, (temp.length+1)*CharsPerWord].
```

POINTER. As discussed in section 4.1.4.2, handling general pointers is difficult without higher-level semantic knowledge of the intended representation. The previous DESCRIPTOR, HANDLE, PROCEDURE, and STRING types all use pointers and are readily marshaled because the nature of their representations is known. Marshaling other types containing pointers must proceed on a similar case-by-case basis.

RECORD. A record is a fixed-size object containing a heterogeneous collection of components represented in contiguous storage. To minimize computational overhead, a record is marshaled by treating it as a block. For example, for *record*: *RecordType*,

CopyToPkt[*pkt*, @*record*, SIZE[*RecordType*]].

After marshaling the body of the record, any address-containing fields in the record (i.e., those fields with referents *outside* the record proper) are marshaled separately as follows:

FOR *field* IN *record* DO IF *record.field* needs marshaling THEN *Marshal*[*record.field*].

Unmarshaling these address-containing fields will again require dynamic storage allocation, as described above for types represented with pointers.

Records are marshaled as a block, rather than field-by-field, so that one large *CopyToPkt* call is made rather than a number of smaller ones. This has a high payoff when a large record has only a few components that need marshaling. (It does, however, waste some space compared to the field-by-field method: the block method trades space for time.) For example, in

many: RECORD [*one*, ..., *ten*: INTEGER, *string*: STRING, *eleven*, ..., *twenty*: BOOLEAN],

only *many.string* needs to be marshaled, and it is easily handled after *many* has been sent as a block. Performing twenty separate calls of *CopyToPkt* for the integer and Boolean fields would result in terrible waste motion. Consider, for a moment, the effect of separately marshaling *every* field of *every* remote call's parameter records: this would be totally unacceptable.

***Variant* RECORD.** The *variant part* of a variant record is a field whose type is the union of a set of record types. The *tag* field of a variant part identifies which record type the variant part is currently bound to. For example:

```
VariantRecord: TYPE = RECORD [
        commonPart: ...,
        variantPart: SELECT tag: {red, blue, green} FROM
                red => [ --The fields of a red VariantRecord.-- ],
                blue => [ --The fields of a blue VariantRecord.-- ],
                green => [ --The fields of a green VariantRecord.-- ]
                ENDCASE ].
```

Marshaling *VariantRecord* is similar to marshaling a standard record except that marshaling the *variantPart* must dynamically choose whether to marshal a *red*, *blue*, or *green* variant. This decision cannot be made at compile time, nor is it sufficient to marshal just the largest variant: the pointers in a *red* record's variant will not in general occupy the same fields as the pointers in a *blue* record's variant; each variant requires separate marshaling. Of course, if none of the variants need to be marshaled, then the *variantPart* can be treated as a plain, uninterpreted record. But when *variantPart* does need to be marshaled, the following code must be compiled for the *variantPart* field of *VariantRecord*:

```
WITH VariantRecord.tag SELECT FROM
        red => Marshal[red VariantRecord];
        blue => Marshal[blue VariantRecord];
        green => Marshal[green VariantRecord];
        ENDCASE => ERROR.
```

Unmarshaling is exactly the same, assuming that *tag* has already been unmarshaled. This should be the case because *tag* is actually in the *commonPart* of *VariantRecord*.

In Mesa, COMPUTED and OVERLAID variant records cannot be automatically marshaled because they contain no explicit *tag* field. Of course, if the records contain no pointers and therefore need no marshaling, they can be treated as plain records as mentioned above.

VAR *parameters*. For remote calls, VAR parameters are passed as call-by-value-result. This involves no special handling for argument records, but it does mean that result records get augmented with the VAR arguments. For example, consider the following procedure *P*:

P: PROCEDURE [*a*: VAR *A*, *b*: *B*, *c*: VAR *C*, *d*: *D*] RETURNS [*q*: *Q*, *r*: *R*, ...].

For marshaling purposes, the result record of *P* is redefined to be

RETURNS [*a*: *A*, *c*: *C*, *q*: *Q*, *r*: *R*, ...].

An important additional consequence of copy-back semantics is that the *referents* of address-containing VAR parameters must not be allocated anew, but rather must be copied

back over their original referents. For example, after a remote call that takes an *s*: VAR STRING argument, the *StringBody* returned by the callee must be copied directly into *s.StringBody* without changing the caller's value of the pointer *s*. The normal procedure of copying *StringBody* into new storage, and readjusting *s*'s pointer to the new storage, is incorrect. Unmarshaling VAR parameters therefore involves slight changes to the unmarshaling techniques given above. In particular, unmarshaling a VAR parameter's result uses the corresponding VAR argument's storage, and unmarshaling all other address-containing results uses dynamic allocation as before.

5.4.3.3 Other Languages

This section described marshaling Mesa datatypes. In other programming systems, of course, the details of particular type representations will change. But Mesa is a rich language, and the marshaling techniques presented here extend quite naturally to other languages.

5.4.4 Stubs

Implementing remote procedures with language-level stubs generated by a stub translator is nearly as easy as compiling them directly. The Emissary RPC runtime machinery does not change at all, and the compiler's calling sequence for a remote procedure is identical to the sequence for a local procedure—differences are hidden in the stubs, not compiled into the caller or callee as before.

5.4.4.1 An Example

To illustrate the details of the Emissary stub approach, consider the *ReadPage* procedure in figure 5.5.

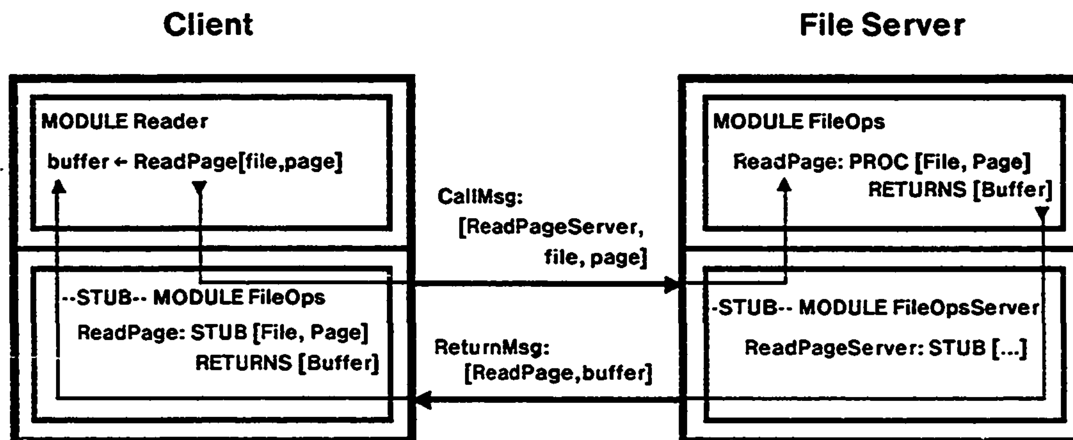


Figure 5.5: A remote *FileOps.ReadPage* call from client to server.

The translator's "STUB" procedures for *ReadPage* are remarkably similar to the compiled remote call sequences presented in figure 5.3. The difference is that the sequences are now contained in the stubs and written in a high-level language. Consider the following *ReadPage* example, reading the comments within each procedure:

```
ReadPage: --CLIENT STUB-- PROCEDURE [file: File, page: Page] RETURNS [buffer: Buffer] =
-- This transparent stub implementation of FileOps's ReadPage procedure has the same
semantics as the real implementation of ReadPage. But rather than perform the actual call, this
client stub procedure communicates with the remote server's stub procedure, below, which
invokes the real implementation. The standard Emissary RPC runtime mechanism, invoked
via RemoteCall, performs the communication.
        { pkt: Packet = AllocPkt[call];
         Marshal[pkt, file, page];
         ClientCall.RemoteCall[pkt, [node: serverNode, inport: ReadPageServer] ];
         --Results come back in pkt.
         Unmarshal[pkt, buffer]; FreePkt[pkt];
         RETURN [buffer] };

ReadPageServer: --SERVER STUB-- PROCEDURE [pkt: Packet] =
-- ReadPageServer is called by the server's RPC runtime mechanism whenever a call pkt for
ReadPageServer arrives. ReadPageServer calls the real ReadPage implementation and puts
ReadPage's results in a return packet that is sent back to the client's waiting ReadPage stub.
        { file: File; page: Page; buffer: Buffer;
         Unmarshal[pkt, file, page]; FreePkt[pkt];
         buffer ← FileOps.ReadPage[file, page]; -- Perform the real work.
         pkt ← AllocPkt[return];
         Marshal[pkt, buffer];
         RETURN }. -- Return to the server's RPC machinery, which sends the return pkt.
```

In this example, the *call* and *return pkts* are explicitly shown because source-level stubs will usually handle them directly, not through implicit *myProcess.pkt* packets as before. Similarly, the remote *ProcedureDescriptor* in *RemoteCall* is explicitly given as the pair [*serverNode*, *ReadPageServer*] because the language's runtime system does not implement remote procedure descriptors, just local descriptors containing an *inport*. The *Marshal* and *Unmarshal* operations are again open-coded procedures that behave exactly as before, except that their code is now written in a high-level language (but see below for procedure parameters).

As this example shows, implementing stubs is fairly straightforward. The key concept is that all of the information needed to compile the stubs for a procedure *P* appears in the *declaration* of *P*, i.e., the stubs depend only on the *FileOps's* interface and are completely independent of *P's* callers and implementation.

5.4.4.2 Stub Advantages

In realistic systems with unusual parameter types, performance requirements, or heterogeneous nodes, stubs and stub translators have some potential advantages over the compiled approach discussed before.

> *Extensibility.* Because stub translators generate language-level stub programs, the internal operation of RPC stubs is clearly visible and changeable by applications. This modifiability can be exploited to perform enhanced marshaling, to do special performance tuning, and so on.
>
> *Flexibility.* The perspicuous nature of stubs also allows them to be modified to work in heterogeneous environments—that is, manually changed to have some of the pleasant properties when the pleasant properties are not built in. This can be useful in several situations: First, if different languages have different (homogeneous) RPC mechanisms,

their stubs can be changed to use an RPC protocol common to all of the languages. Second, in heterogeneous processor environments, stubs can be changed to translate automatically between primitive datatypes.

5.4.4.3 Marshaling Procedure and other Control Transfer Parameters

Marshaling parameters with procedure and other control transfer types is not as easy as suggested above. There are two problems to be solved:

ProcedureDescriptors. Complete *ProcedureDescriptors* that include a *node* component are usually not implemented in languages with stubs, and thus marshaling control transfer types as discussed in section 5.4.3.2 will not work. A full descriptor for each control transfer parameter must be anonymously fabricated and used instead.

Marshaling. Parameters with control transfer types must have their *own* parameters properly marshaled for remote use. In the compiled Emissary approach, this was accomplished by having the compiler automatically check all remotely used control transfer parameters for the REMOTE attribute, as discussed in section 4.1.4.3.

A language-level stub translator must use somewhat circumspect solutions to these two problems. The basic approach is to compile *nested stubs* for all procedure and control transfer parameters—one in the client and one in the server. As an example of this, the nested stubs for the *ItemProcedure* parameter of the following *Enumerate* routine are presented below.

Item: TYPE = ...;
ItemProcedure: TYPE = PROC [*item*: *Item*] RETURNS [*stopEnumerating*: BOOLEAN←FALSE];
Enumerate: PROCEDURE [*itemProcedure*: *ItemProcedure*] RETURNS [*lastItem*: *Item*].

Enumerate works in this way: *Enumerate* steps through a data base of *Items*, calling a client-supplied *itemProcedure* for each *item*. To finish the enumeration early, *itemProcedure* returns with *stopEnumerating* set to true. *Enumerate* always returns the *lastItem* enumerated.

Here is the flow of control in a remote call of *Enumerate* (read this while following the code below): A client calls the *Enumerate* stub, which makes a remote call to *EnumerateServer*, which calls the server's real *Enumerate* routine. *Enumerate* then calls the server's local *itemProcedure* stub for each *item*. Each call on the *itemProcedure* stub actually calls nested stub *ItemProcedureServerConverter*, which makes a remote call to nested stub *ItemProcedureClientConverter*, which calls the client's real *itemProcedure*. The return sequence is the reverse: the real *itemProcedure* returns to *ItemProcedureClientConverter*, which returns to *ItemProcedureServerConverter*, which returns to the real *Enumerate*, which returns to *EnumerateServer*, which returns to the client's *Enumerate* stub.

Note that the client and server roles are reversed in the nested stubs *ItemProcedureClientConverter* and *ItemProcedureServerConverter*. This is because the *server* uses *itemProcedure* to call the *client*, the reverse direction of the top-level *Enumerate* call. The nested—not global—declarations of control transfer parameter stubs are necessary so that (for example) *itemProcedure* and *itemProcedureClientConverter* are correctly and uniquely bound in the client and server. This permits the proper operation of recursive or concurrent calls to *Enumerate*

and *EnumerateServer.* Finally, although the stubs that marshal procedure and other control transfer parameters are complicated, generating them is straightforward if the stub translator is designed to be called recursively—i.e., to generate one stub in the middle of another.

```
Enumerate: --CLIENT STUB-- PROCEDURE [itemProcedure: ItemProcedure]
                                             RETURNS [lastItem: Item] =
-- Client programmers call this transparent Enumerate stub and give it a local itemProcedure
argument. The nested ItemProcedureClientConverter stub anonymously converts this local
procedure into a remote one that can be used by remote Enumerate.
    { ItemProcedureClientConverter: --SERVER STUB-- PROCEDURE [itemPkt: Packet] =
      -- ItemProcedureClientConverter is called remotely by ItemProcedureServerConverter
      and converts the server node's stubbed itemProcedure calls into local calls on the real
      itemProcedure procedure parameter.
            {item: Item; stopEnumerating: BOOLEAN;
            Unmarshal[itemPkt, item]; FreePkt[itemPkt];
            stopEnumerating ← itemProcedure[item]; -- Invoke the real proc. parameter.
            itemPkt ← AllocPkt[return];
            Marshal[itemPkt, stopEnumerating];
            RETURN }; -- Return to RPC machinery and send a return to server.
      pkt: Packet = AllocPkt[call];
      Marshal[pkt, ItemProcedureClientConverter];
      ClientCall.RemoteCall[pkt, [node: serverNode, inport: EnumerateServer] ];
      -- Results come back in pkt.
      Unmarshal[pkt, lastItem]; FreePkt[pkt];
      RETURN [lastItem] }; -- Return to the original client.

EnumerateServer: --SERVER STUB-- PROCEDURE [pkt: Packet] =
-- EnumerateServer is called by the server node's RPC mechanism when a remote call of
EnumerateServer is received. The transparent nested ItemProcedureServerConverter stub
anonymously converts local itemProcedure calls into remote ones.
    { itemProcedureClientConverter: PROCEDURE [pkt: Packet]; -- Set by Unmarshal below.
      lastItem: Item;
      ItemProcedureServerConverter: --CLIENT STUB-- ItemProcedure =
      -- ItemProcedureServerConverter is called by Enumerate and converts Enumerate's
      local calls to the stubbed itemProcedure into remote calls to the
      client node's ItemProcedureClientConverter.
            {itemPkt: Packet = AllocPkt[call];
            Marshal[itemPkt, item];
            ClientCall.RemoteCall[ itemPkt,
                        [node: clientNode, inport: itemProcedureClientConverter] ];
            -- Results come back in pkt.
            Unmarshal[itemPkt, stopEnumerating]; FreePkt[itemPkt];
            RETURN[stopEnumerating] }; -- Return to local Enumerate procedure.
      Unmarshal[pkt, itemProcedureClientConverter]; FreePkt[pkt];
      lastItem ← Enumerate[itemProcedure: ItemProcedureServerConverter]; -- Do real call.
      pkt ← AllocPkt[return];
      Marshal[pkt, lastItem];
      RETURN }. -- Return to the server's RPC machinery, which sends the return pkt.
```

5.4.4.4 Stub Translator Design Issues

The following points address some remaining issues of stubs and stub translators.

Symbol Table Information. The generation of the two stubs for a procedure *P* primarily needs only the text of *P's* declaration: the text supplies the names of *P*, its arguments, and its results. Marshaling, on the other hand, requires detailed knowledge of *P's* parameter types. In a language with a rich type calculus, properly parsing, checking, and processing this type information is a formidable task properly left to the compiler. Stub translators should therefore be designed to use as much of the compiler's existing, precompiled symbol table information as possible. Indeed, if a translator is being designed for a language that has separately compiled interfaces, using the compiled interface as the sole input to the translator may be possible. This works exceptionally well in Mesa because compiled interfaces have an elaborate symbol table, with type descriptions that are more than adequate for marshaling.

Remote Binding. If remote procedures are implemented with a stub translator, it is unlikely that the elaborate binding scheme to be presented in section 5.5 will be available. In lieu of that elegant approach, the following simplistic binding method is suggested for expedient stub systems.

In figure 5.5, for the client's *ReadPage* stub to call *ReadPageServer* remotely, the stub must pass *ReadPageServer's* procedure descriptor to *RemoteCall.* To obtain this descriptor in the first place, assume that the client's *FileOps* implementation imports a record of *FileOpsServer's* server stubs by making a special binding call, *ImportStubInterface*, to the server's RPC mechanism. This call returns *FileOpsServer's* stub procedure descriptors, which the client *FileOps* implementation then uses.

To make this more concrete, here is a stub binding example in the setting of figure 5.5.

> *Exporting.* When the server's *FileOpsServer* implementation is initialized, it calls
>
> *ExportStubInterface[FileOpsServer, FileOpsTypeID, [OpenFileServer, ..., ReadPageServer]]*.
>
> This tells the server's RPC mechanism that import requests for *FileOpsServer* should be given the interface of *[OpenFileServer, ..., ReadPageServer]* procedure descriptors in reply. The *FileOpsTypeID* argument is used to check that the *FileOps* version the client needs is precisely the one the server is exporting.
>
> *Node binding.* When the client's *FileOps* stub implementation is initialized, it first locates *serverNode*, the node where *FileOpsServer* resides. The client implementation can determine this in a number of ways: by knowing it in advance, by receiving it as a parameter of initialization, and so on.
>
> *Importing.* Once the client's *FileOps* implementation has determined *serverNode*, it performs the following remote call to *serverNode*:
>
> *[OpenFileServer, ..., ReadPageServer]* ← *ImportStubInterface[FileOpsServer, FileOpsTypeID]*.
>
> The *serverNode's* RPC mechanism recognizes this call as a special binding operation and returns the *OpenFileServer, ..., ReadPageServer* descriptors to the client after checking that the client and server *FileOpsTypeID's* match.

> *Interface binding.* The remote binding process completes when each client stub such as *ReadPage* has been given its remote *ReadPageServer* descriptor. This binding is easily accomplished by using a *readPageServer* variable in *RemoteCall* and assigning *ReadPageServer* to *readPageServer* at the end of initialization.

The binding scheme sketched above certainly does not satisfy the powerful binding and configuration property. This explicit, procedure-driven import-export approach may, however, be the only one feasible for modest stub implementations.

Performance. There is a marked reduction in the efficiency of remote calls when Emissary's compiled call sequences are abandoned in favor of translated language-level stubs. The performance loss comes in three areas: First, stubs are themselves procedures that are invoked with normal procedure calls. Since there is an extra layer of calls in both the client and the server, the stub approach incurs an added overhead of two complete calls. Second, each of these additional calls passes the stub's parameters a total of two more times. For large parameters—either arguments or results—this extra copying overhead can be significant. Third, the RPC runtime mechanism of the stub approach is unlikely to be integrated into the underlying process machinery and network software as well as Emissary's. The absence of this microcode support and close network coupling will cause performance to suffer by a factor or two or more. This is an expected result when an abstract machine—the RPC machine—is implemented with high-level software.

Other Control Transfer Primitives. Thus far this section has discussed only procedure stubs. For overall transparency, the translator must generate stubs for other language-level communication primitives as well. This is easy if the stubs for the other remote primitives are implemented with remote procedures as described in section 5.4.2. Since stubs incur performance losses anyway, this approach can be quite acceptable.

5.4.5 Reflections

Emissary's RPC machinery is an evolutionary outgrowth of five closely related remote procedure mechanisms that I have either built or tested. At the language level, all of these mechanisms are accessed through stub translator. At the runtime level, they use a number of different implementations, and features from all of them contribute to Emissary: this is discussed in chapter 6. Thus, while the exact Emissary mechanism presented in algorithm 5.3 has not been implemented, it is the final product of substantial design, implementation, and evaluation.

This section has included a wealth of detail on the construction of transparent and efficient RPC mechanisms. In particular, Emissary's runtime machinery is developed completely, and marshaling parameters is described in depth. Even with the focus on these areas, however, two of their important components have been shortchanged. While both have been discussed before, they deserve final mention here.

> *Large parameter records.* The Emissary runtime machinery and marshaling mechanism explicitly sidestep the problem of parameter records that are too large for one packet. While adding multipacket functionality is conceptually quite simple, in practice such

changes must not reduce the performance of the frequent one-packet case. RPC designers must heed this warning well.

Marshaling list structures. The marshaling approach given in section 5.4.3.2 does not handle general list structures. This restriction was carefully considered in chapter 4, but sufficient client demand for automatic marshaling of trees and graphs could overrule this decision.

5.4.5.1 Transparency and the Essential Properties

The section's goal was to develop approaches for the language-level compilation and runtime execution of remote procedure calls and other language-level communication primitives. This goal was met by designing both compiled and stub-translated approaches. A second and crucial goal was that the call mechanisms must satisfy these four essential properties: uniform call semantics, strong typechecking, excellent parameter functionality, and standard concurrency control and exception handling. These properties are satisfied as follows.

Call semantics. Emissary's RPC runtime mechanism, which is used by both the compiled and translated approaches, uses concurrent invocation and guarantees exactly-once semantics in the absence of crashes. It relies on the previous orphan algorithms in the presence of crashes. This satisfies the uniform call semantics property.

Typechecking. Section 5.4.1.2 discussed how the binder guarantees strong intermodule typechecking by validating the importers and exporters of remote interfaces. In the compiled Emissary approach, remote interface typechecking is performed by the remote binder of section 5.5. In the stub translator approach, it is done by the runtime binding routines described in section 5.4.4.4. Both of these methods satisfy the strong typechecking property.

Parameter functionality. The parameter functionality of a remote procedure mechanism is determined by the power of its underlying marshaling machinery. Section 5.4.3.2 shows how to marshal most Mesa datatypes except list structures, which are specifically excluded. Even with this restriction, Emissary's marshaling is very powerful; this satisfies the excellent parameter functionality property.

Concurrency control and exception handling. Emissary uses concurrent invocation to execute remote calls and reports all exceptions in the standard fashion. This satisfies the standard concurrency control and exception handing property.

5.4.6 The Emissary RPC Algorithm

This separate section contains the text of Emissary's remote procedure mechanism, algorithm 5.3.

```
-- Type definitions, constants, and variables for the Emissary remote procedure algorithm. Familiarity
   with the orphan algorithms is assumed, especially with the time-related definitions of expiration.

   Generation: TYPE = ...; -- Monotonic counter increased after a crash; see text.
   RpcPacketType: TYPE = {call, return, returnAck, generationRequest, generationAck, ...};
   SerialNumber: TYPE = ...; -- Must not overflow in expected process lifetime; see text.
   myGeneration: STABLE Generation ← ...; -- Incremented after each crash.
   myProcess: Process ← ...; -- Currently executing process.
   myNode: Node = ...; -- The node that this variable lives in.
   nodesIn: STABLE SET OF Node ← EMPTY; -- Contains the nodes of all incoming calls.
   nodesOut: STABLE SET OF Node ← EMPTY; -- Contains the nodes of all outgoing calls.

   Node: TYPE = RECORD [ -- Components of an internetwork address.
           network: NetworkID, -- Network number.
           host: HostID ]; -- Host number (may or may not be unique across networks).
   ProcessID: TYPE = RECORD [ -- Internet-wide process identifier, used for extermination.
           node: Node, -- Node executing process, myNode if local.
           process: Process ]; -- Process number.
   CallID: TYPE = RECORD [ -- Compound, volatile (intracrash) identifier for a given call.
           node: Node, -- Node issuing the remote call.
           process: Process, -- Client process making the remote call.
           serial: SerialNumber ]; -- Serial number of process's current call.
   ProcedureDescriptor: TYPE = RECORD [
           node: Node, -- Node that inport's procedure resides in.
           inport: ProcedureInport ]; -- Identifies a procedure context in node.

   Packet: TYPE = POINTER TO PacketObject;
   PacketObject: TYPE = MACHINE DEPENDENT RECORD [
           callInfo: RECORD [ -- Call bookkeeping information that is not transmitted.
             state: {idle, pending, working, returnSent, requestAck}, -- Server calls cycle through state.
             source: CallID ], -- For an unacked return packet, identifies the source of the original call.
           transport: RECORD [ -- Physical transport header (e.g., Ethernet).
             source, dest: Node, -- Node, as before.
             type: {internet, RPC, ...} ], -- Transport packets have internet, RPC, and other types.
           internet: RECORD [ -- Internetwork header (e.g., Pup).
             ... -- Other fields of an internet packet.
             type: {..., RPC, ...} ], -- Internet packets also have special RPC subtype.
           rpc: RECORD [ -- Fields used by RPC packets.
             expirationTime: Time, -- Time that the call must be automatically aborted.
             generation: Generation, -- Sending node's generation when packet sent.
             callID: CallID, -- For this generation, ID of current call.
             type: RpcPacketType, -- Flavor of RPC message.
             inport: ProcedureInport, -- Procedure to call in callID.node.
             paramLength: CARDINAL, -- Relative length of parameters.
             parameters: RECORD [...] ], -- Variable length record.
           transportEnd: RECORD [...] ]; -- Transport error control info, if any.

   Process: TYPE = POINTER TO ProcessObject;
   ProcessObject: TYPE = MONITORED RECORD [
     ... -- Other parts of a process object.
           expirationTime: Time, -- Time that the process expires (for expiration).
           worker: Node ←myNode, -- Node doing myProcesses's remote call, if any (for extermination).
           parent: ProcessID←[myNode, myProcess], -- Parent process of myProcess (for extermination).
           rpcLock: MONITORLOCK, -- For mutual exclusion in ClientCall monitor.
           serial: SerialNumber ← 0, -- Number of most recent remote call.
           callState: {idle, returnWanted, returnReceived} ← idle, -- Client calls cycle through callState.
           callDone: CONDITION, -- Used to synch callState with RPC machinery.
           pkt: Packet ← NIL, -- Cached packet for RPC machinery.
           pktInUse: BOOLEAN ← FALSE ]; -- Used by packet allocators to reclaim the cached pkt.
```

NetworkService: MODULE = BEGIN
-- The *NetworkService* module contains network and packet buffer routines.

GetPacketBuffer: ENTRY PROCEDURE RETURNS [*pkt*: *Packet*] = { ... };
-- Gets a packet from the system's queue of packet buffers. (Should be in a *PacketBuffer* monitor.)

ReturnPacketBuffer: ENTRY PROCEDURE [*pkt*: *Packet*] = { ... };
-- Returns *pkt* to the system's free packet buffer queue. (Should be in a *PacketBuffer* monitor.)

AllocPkt: PROCEDURE [*p*: *Process*, *type*: *RpcPacketType*] =
-- *AllocPkt* is used by the compiler. It gets a packet buffer for process *p* and initializes it as much as possible, including setting the *expirationTime* of a remote *call* packet. For efficiency, a process may already have a cached packet allocated to it. In this case, some fields are already initialized properly and they are not touched here.

```
{ IF p.pkt = NIL THEN {
        pkt: Packet ← p.pkt ← GetPacketBuffer[];
        pkt.transport.type ← pkt.internet.type ← RPC;
        pkt.rpc.generation ← myGeneration;  pkt.rpc.callID ← [myNode, myProcess, ] };
    p.pkt.rpc.type ← type;  p.pktInUse ← TRUE;
    p.pkt.rpc.expirationTime ← IF p.expirationTime = NeverExpires
        THEN Clock[] + ExpirationInterval ELSE p.expirationTime;
    p.pkt.rpc.callID.serial ← (p.serial ← p.serial + 1) };
```

FreePkt: PROCEDURE [*p*: *Process*] =
-- *FreePkt* is used by the compiler. It marks process *p's* cached packet as not in use but does not deallocate it. Instead, a daemon process (perhaps even garbage collector) will want to run frequently and reclaim all the unused (~*pktInUse*) cached buffers. This keeps the cache meaningful, and the buffer pool nonempty. The daemon must be cautious about synchronization.

```
INLINE { p.pktInUse ← FALSE };
```

Transmit: PROCEDURE [*node*: *Node*, *pkt*: *Packet*] =
-- *Transmit* sends *pkt* to *node*. For performance, it sends *pkt* directly if *node* is on the local network, which is the most common case. This bypasses the level 1 communication software overhead and shortens the packet because the internet header can be eliminated. If the local network optimization fails because the transport transmitter is busy, then the regular internet mechanism (which has output queueing) is used. When internet transmission is used, some implementations may need to copy *pkt* so that there is no interference between internet and client processes.

```
INLINE { IF node.network # myNetwork OR TryTransportTransmit[node.host, pkt].busy
            THEN InternetTransmit[node, pkt] };
```

StartRpcReceive: PROCEDURE [*rpcWakeup*: POINTER TO CONDITION, *buffer*: *Packet*] =
-- *StartRpcReceive* sets up the I/O control registers so that the next RPC packet is received into *buffer*. Since this packet can be either a transport RPC packet with *buffer.transport.type*=*RPC* (e.g., Ethernet packet), or an internet RPC packet with *buffer.internet.type*=*RPC* (e.g., Pup packet), the implementation of the network device interface must look at both of these type fields and deliver both kinds of RPC packets to *buffer*. This low-level demultiplexing is necessary for performance. When *buffer* is received, the device automatically performs a hardware NOTIFY on *rpcWakeup*↑ (called a *naked* NOTIFY). The high-level logical operation of *StartRpcReceive* is shown below.

```
INLINE { -- Set up the local network device interface for these steps:
        -- buffer↑ ← NewRpcPacket;
        -- NOTIFY rpcWakeup↑ };
```

CheckReceive: PROCEDURE RETURNS [*status*: {*ok*, ...}] =
-- *CheckReceive* is called after every network input (initiated by *StartRpcReceive*) to ensure that the packet is not damaged in any way. Any *status* but *ok* means that the packet should not be used.

```
INLINE { -- status ← IF ChecksumOK[NewRpcPacket] THEN ok ELSE ... };
```

END; -- of *NetworkService* module.

```
ClientCall: MONITOR LOCKS clientProcess.rpcLock USING clientProcess: Process = BEGIN
  -- The ClientCall object monitor synchronizes data in the RPC part of the ProcessObject.

 RemoteCall: ENTRY PROCEDURE [clientProcess: Process, dest: ProcedureDescriptor] =
  -- RemoteCall is the operation used the the compiler to perform a remote call. Before starting a call,
  RemoteCall makes sure that the destination node is in nodesOut for extermination purposes; it also
  maintains worker for Exterminate as well. RemoteCall then sends the call packet in clientProcess.pkt
  to dest.node. The call is re-sent at intervals of callRetransmitInterval up to retriesUntilFailure times.
  If no valid return packet comes (via RemoteReturn, below), the Failed exception is raised. If a valid
  return does come, then RemoteReturn puts the return packet in clientProcess.pkt for the compiler.
  When RemoteCall returns to its caller, this return packet is ready for the caller.
        { IF dest.node ~IN nodesOut THEN nodesOut ← nodesOut + dest.node;
          clientProcess.worker ← dest.node; -- Record worker node for Exterminate.
          clientProcess.pkt.rpc.inport ← dest.inport;
          clientProcess.callState ← returnWanted;
          THROUGH [0..retriesUntilFailure) DO
                Transmit[dest.node, clientProcess.pkt];
                WAIT clientProcess.callDone; -- With timeout of callRetransmitInterval.
                IF myProcess.callState = returnReceived THEN EXIT;
                -- Repeat loop and retransmit the call packet if the WAIT timed out.
            REPEAT
                FINISHED => SIGNAL Failed;
            ENDLOOP;
          clientProcess.worker ← myNode; -- No longer need worker for Exterminate.
          clientProcess.callState ← idle };

 RemoteReturn: ENTRY PROCEDURE [clientProcess: Process, returnPkt: Packet]
                        RETURNS [freePkt: Packet] =
  -- RemoteReturn is called by an RpcServerProcess when a return packet arrives. RemoteReturn first
  checks the state of the local process that originated the call. If the process is not currently performing
  a remote call, then it is not expecting a return packet, and therefore the return packet must be a
  duplicate or a request for an explicit acknowledgement of the return. In this case, a returnAck packet
  is sent back to the source of the return. If the process is performing a call and does want a return,
  then the return packet is stashed in clientProcess.pkt if the serial number matches. RemoteReturn
  then NOTIFYs RemoteCall, above, so that the remote call can complete. In any case, RemoteReturn
  always returns a freePkt to its caller.
        { freePkt ← returnPkt;
          SELECT clientProcess.callState FROM
                idle, returnReceived =>
                        {returnPkt.rpc.type ← returnAck;
                        Transmit[returnPkt.rpc.callID.node, returnPkt]};
                returnWanted =>
                        IF clientProcess.serial = returnPkt.rpc.callID.serial THEN
                          {freePkt ← clientProcess.pkt;
                          clientProcess.pkt ← returnPkt;
                          clientProcess.callState ← returnReceived;
                          NOTIFY clientProcess.callDone };
                ENDCASE => ERROR };

END; -- of ClientCall monitor.
```

```
ServerCall: MONITOR = BEGIN

  activeCalls: SET OF Packet ← EMPTY;
  -- ServerCall protects the activeCalls set of RPC call (and call-turned-into-return) packets that are
  being processed by the RpcServerProcesses.

 AddCall: ENTRY PROCEDURE [callPkt: Packet] =
  -- AddCall is called at interrupt level by an RpcServerProcess to atomically add a new callPkt packet
  to the set of activeCalls. AddCall also sets callPkt's callInfo.state to pending for later use by
  CheckDuplicatesAndAck, and source.callID to callPkt's source callID for the transmission of the
  eventual return. In practice, it is probably necessary to make AddCall work even when the
  ServerCall monitor is locked because some locked operations like FindOtherCaller take a long time.
  It is not acceptable for interrupt-level calls of AddCall to wait very long for the monitor lock.
    INLINE { callPkt.callInfo ← [state: pending, source: callPkt.rpc.callID];
             activeCalls ← activeCalls + callPkt };

 DeleteCall: ENTRY PROCEDURE [callPkt: Packet] =
  -- DeleteCall is called by RpcServerProcess to remove a callPkt that does not Verify. It is okay if
  callPkt is not in activeCalls.
    INLINE { activeCalls ← activeCalls − callPkt };

 CheckDuplicatesAndAck: ENTRY PROCEDURE [newCallPkt: Packet] RETURNS [execute: BOOLEAN] =
  -- CheckDuplicatesAndAck is called by RpcServerProcess for every call packet. If newCallPkt is
  from the same node and process as an existing call, then its serial number is examined. If
  newCallPkt is an old duplicate call—i.e., is not the most recent call—it is ignored. If newCallPkt is a
  duplicate for the most recent call—i.e., has the latest serial number—there are three cases to
  consider: first, if oldCallPkt is in the idle state, then newCallPkt is a (long) delayed call and is
  ignored; second, if oldCallPkt is in the pending or working states, then oldCallPkt's execution is in
  progress and newCallPkt is ignored again; third, if oldCallPkt is in the returnSent or requestAck
  states, then the call is complete and the (probably lost and never received) return is resent. If
  newCallPkt is not a duplicate, then it is executed after implicitly acknowledging the oldCallPkt.
  Because back-to-back call packets can pile up on activeCalls at interrupt level, there can be many
  matching oldCallPkts for every newCallPkt. FindOtherCaller only returns one, but this is fine since
  CheckDuplicatesAndAck is called for each call packet in activeCalls, and thus all duplicates are
  eventually eliminated.
          { oldCallPkt: Packet = FindOtherCaller[newCallPkt];
           IF oldCallPkt # NIL THEN
              SELECT oldCallPkt.callInfo.source.serial FROM
                 >= newCallPkt.callInfo.serial => {
                       IF oldCallPkt.callInfo.source.serial = newCallPkt.callInfo.serial THEN
                         -- oldCallPkt is a recent duplicate.
                         SELECT oldCallPkt.callInfo.state FROM
                               idle, pending, working => NULL;
                               returnSent, requestAck => -- The return was lost; resend it.
                                     Transmit[oldCallPkt.callInfo.source.node, oldCallPkt];
                               ENDCASE => ERROR;
                       activeCalls ← activeCalls − newCallPkt;
                       RETURN [execute: FALSE] };
                 < newCallPkt.callInfo.serial => CallAck[oldCallPkt, implicit];
                 ENDCASE => ERROR.
           newCallPkt.callInfo.state ← working;
           RETURN [execute: TRUE] };
```

ExplicitCallAck: ENTRY PROCEDURE [*ackPkt*: *Packet*] =
-- *ExplicitCallAck* is called when an *RpcServerProcess* receives a *returnAck* packet. A *returnAck* is sent by the caller to the server whenever a caller sees the same *return* message more than once. The *returnAck* acknowledges an outstanding *return* only if the *return* is still pending and the *returnAck* has a matching serial number.

```
        { oldReturnPkt: Packet = FindOtherCaller[ackPkt];
          IF oldReturnPkt # NIL AND oldReturnPkt.callInfo.source.serial = ackPkt.rpc.callID.serial
              THEN CallAck[oldReturnPkt, explicit] };
```

CallAck: INTERNAL PROCEDURE [*oldCall*: *Packet*, *kind*: {*implicit*, *explicit*}] =
-- *CallAck* is called when *oldCall* has been acknowledged either *implicitly* or *explicitly*. (Note: at this point, the *oldCall* is actually the *return* packet sent back to the caller). Since the acknowledgement guarantees that the *return* was received by the caller, *oldCall* moves into the connection-maintaining *idle* state. The purpose of *idle* is to keep (in *activeCalls*) a *CallID* record of the last call from *oldCall.callInfo.source*; this connection ID permits the detection of duplicate *calls* even after an *explicit returnAck* is received. (This is wasteful of packets; some simple and effective optimizations are discussed in the text.) If *oldCall* is being *implicitly* acknowledged by a new call, then the new one replaces *oldCall* and *oldCall* is deleted from *activeCalls*.

```
  INLINE { SELECT oldCall.callInfo.state FROM
              idle => NULL; -- Ignore duplicate acks, e.g., implicit after explicit.
              returnSent, requestAck => oldCall.callInfo.state ← idle; -- Got ack, go into idle.
              ENDCASE => ERROR; -- Should never get an ack for a pending or working call.
          IF kind = implicit THEN
              {activeCalls ← activeCalls − oldCall;
              NetworkService.ReturnPacketBuffer[oldCall]} };
```

FindOtherCaller: INTERNAL PROCEDURE [*keyPkt*: *Packet*] RETURNS [*foundPkt*: *Packet*←NIL] =
-- *FindOtherCaller* searches through *activeCalls* for a *foundPkt* from the same node and process as *keyPkt*. There can be more than one such *foundPkt*, but only the first one found is returned. Since *keyPkt* is itself in *activeCalls*, it is not allowed to match itself. The implementation given below uses a simple linear search. In practice, a much faster scheme such as hashing is needed, perhaps with a microcode assist since *FindOtherCaller* must be invoked for each remote call.

```
        { FOR pkt IN activeCalls DO
              IF pkt.callInfo.source.node # keyPkt.rpc.callID.node
               AND pkt.callInfo.source.process # keyPkt.rpc.callID.process
               AND pkt # keyPkt
                 THEN RETURN[pkt];
              ENDLOOP };
```

FlushReturns: ENTRY PROCEDURE [*toNode*: *Node*] =
-- *FlushReturns* is used by *CheckConnection* to delete old *return* packets destined for node *toNode*. Explicit deletion is necessary after *toNode* has crashed and begins new calls in its next generation. *Idle returns* must be deleted because they contain connection IDs for invalid connections. *ReturnSent* and *requestAck returns* must be deleted because they are no longer needed and will never be acknowledged.

```
        { FOR pkt IN activeCalls DO
            IF pkt.callInfo.source.node = toNode THEN
                SELECT pkt.callInfo.state FROM
                        idle, returnSent, requestAck => {
                              activeCalls ← activeCalls − pkt;
                              NetworkService.ReturnPacketBuffer[pkt] };
                        ENDCASE => ERROR;
            ENDLOOP };
```

```
RequestAckIfNeeded: ENTRY PROCEDURE [returnPkt: Packet] =
  -- RequestAckIfNeeded is called by the ReturnRetransmitter process, below. RequestAckIfNeeded
  looks for return packets that have not been acknowledged—either implicitly or explicitly—by their
  callers. It asks that all old return packets in the requestAck state be explicitly acknowledged by
  resending the returns. This tells the callers to send returnAcks in reply. RequestAckIfNeeded also
  changes the state of fresh returns (in returnSent) so that they will be handled as old returns the next
  time around.
        { SELECT returnPkt.callInfo.state FROM
                idle => NULL; -- Already acked; just maintaining a calling-process connection.
                pending, working => NULL; -- Still performing call, no return sent yet.
                returnSent => returnPkt.callInfo.state ← requestAck; -- Request ack next time.
                requestAck => Transmit[returnPkt.callInfo.source.node, returnPkt];
                ENDCASE => ERROR };
END; -- of ServerCall monitor.
```

```
ReturnRetransmitter: Process =
  -- This background process cooperates with RequestAckIfNeeded, above. It makes sure that all
  return packets in activeCalls are received and acknowledged by their callers. Usually, returns are
  implicitly acknowledged by the next call from the same process (see CheckDuplicatesAndAck). But
  when this is not the case, explicit acknowledgements are requested (by resending returns) every
  acknowledgementInterval. (The text explains what happens if a returnAck is never received). As it is
  written, the FOR loop has a synchronization bug because activeCalls is in the ServerCall monitor. In
  practice, this enumeration should take place outside the monitor for performance reasons but should
  not make inconsistent use of activeCalls.
      { DO
          SuspendProcess[acknowledgementInterval]; -- A fairly long time, say, several minutes.
          FOR pkt IN activeCalls DO ServerCall.RequestAckIfNeeded[pkt] ENDLOOP;
        ENDLOOP };
```

-- *RpcServerProcesses* that handle arriving RPC packets and execute incoming remote calls.

```
RpcServerProcess: TYPE = Process;
rpcHandlers: SET OF RpcServerProcess ← EMPTY; -- Software cache of RpcServerProcesses.
firstHandler: NEW RpcServerPrototype; -- Create first handler and receive first packet.
NetworkService.StartRpcReceive[@firstHandler.rpcPktReceived, firstHandler.inPkt];

RpcServerPrototype: RpcServerProcess =
-- This process is a prototype for the rpcHandlers process pool. Each process in the pool receives and
processes RPC packets. Each RpcServerProcess starts by getting a packet buffer for itself. It then
waits for a naked NOTIFY telling it that an RPC inPkt has been received. If the inPkt is ok, and it is a
call, the call's parent process is recorded for extermination and the call is added to activeCalls. Then
a new RpcServerProcess is readied to receive the next packet. For performance, the new process is
obtained from the rpcHandlers cache if possible. The preceding all happens very quickly, at
interrupt level, so that back-to-back packets can be received as fast as possible. Once inPkt is
received and the interrupt dismissed, Verify is called to check connections and other details. If Verify
says the inPkt is okay then it is acted on, otherwise it is discarded. If inPkt is a call packet and is not
a duplicate, then the call is executed with its specified expirationTime. This is accomplished by
invoking the local procedure specified by inPkt.rpc.inport. Arguments and results are passed
through myProcess.pkt, as explained in the text. A return packet for the call is sent, and the return's
state is set to returnSent so that the return will be retained until an acknowledgement is received.
This gives exactly-once semantics. If inPkt is a return packet then RemoteReturn completes the
remote call and continues the caller. If inPkt is an explicit returnAck then the acknowledgement is
handled in the ServerCall monitor.
{ rpcPktReceived: CONDITION;
  inPkt: Packet ← NetworkService.GetPacketBuffer[]; myProcess.pktInUse ← TRUE;
  DO    ENABLE Aborted => -- This cleanup is done if myProcess is Aborted by the orphan algs.:
                { ServerCall.DeleteCall[inPkt]; NetworkService.ReturnPacketBuffer[inPkt] };
        WAIT rpcPktReceived;
        IF NetworkService.CheckReceive[] # ok THEN
                { NetworkService.StartRpcReceive[@rpcPktReceived, inPkt]; LOOP };
        IF inPkt.rpc.type = call THEN { -- Record remote call's parent for Exterminate.
                myProcess.parent ← [node: inPkt.rpc.callID.node, process: inPkt.rpc.callID.process];
                ServerCall.AddCall[inPkt] };
        nextHandler: RpcServerProcess = IF ~EMPTY rpcHandlers
                THEN TAKE FIRST rpcHandlers ELSE NEW RpcServerPrototype;
        NetworkService.StartRpcReceive[@nextHandler.rpcPktReceived, nextHandler.inPkt];
        SetProcessPriority[normal]: -- No longer at interrupt level.
        IF Connection.Verify[inPkt]
          THEN SELECT inPkt.rpc.type FROM
            call => { IF ServerCall.CheckDuplicatesAndAck[inPkt].execute THEN {
                        myProcess.expirationTime ← inPkt.rpc.expirationTime; -- Set time limit.
                        myProcess.pkt ← inPkt;
                        InvokeProcedure[inPkt.rpc.inport, NIL];
                        -- myProcess.pkt is now the returnPkt.
                        Transmit[myProcess.pkt.callInfo.source.node, myProcess.pkt];
                        myProcess.pkt.callInfo.state ← returnSent;
                        myProcess.expirationTime ← NeverExpires; -- No more limit.
                        inPkt ← NetworkService.GetPacketBuffer[] };
                    myProcess.parent ← [myNode, myProcess] }; -- No parent any longer.
            return => inPkt ← ClientCall.RemoteReturn[inPkt.rpc.callID.process, inPkt];
            returnAck => ServerCall.ExplicitCallAck[inPkt];
            ENDCASE => ERROR
          ELSE IF inPkt.rpc.type = call THEN ServerCall.DeleteCall[inPkt];
        rpcHandlers ← rpcHandlers + myProcess;
        ENDLOOP }
}; -- of RpcServerPrototype.
```

```
Connection: MONITOR = BEGIN

  ConnectionRecord: TYPE = RECORD [node: Node, generation: Generation];
  connections: SET OF ConnectionRecord ← EMPTY;
  -- The Connection monitor synchronizes access to the connections set, which is a record of the nodes
  that myNode has open connections to. A node increases its generation each time it crashes. Since all
  packets carry the generation they were sent in, delayed packets from previous generations can be
  detected and discarded.

Verify: PROCEDURE [pkt: Packet] RETURNS [ok: BOOLEAN ← FALSE] =
  -- Verify is called by RpcServerProcesses to check the validity of each packet: Expired call packets
  are ignored (old return and returnAck packets are okay because acknowledgements can use them
  even after their call expires). Generation packets are always honored so that new connections can be
  established. Packets from nodes not in nodesIn are ignored until a connection is established (this
  prevents out-of-order (delayed) packets from causing inconsistencies). Packets that are known to be
  from old generations are ignored. If a packet passes all of these tests it is valid, and—on the
  assumption that it will be sent back in reply—its generation is set to myGeneration. In practice all of
  these tests except the last one using CheckConnection can be performed very efficiently.
  CheckConnection is discussed below.
      { source: Node = pkt.rpc.callID.node;
        SELECT TRUE FROM -- Any TRUE condition invalidates the packet.
            pkt.rpc.type=call AND pkt.rpc.expirationTime<Clock[] => NULL; -- Expired call packet.
            pkt.rpc.type IN {generationRequest, generationAck} => Connection.HandleTraffic[pkt];
            source ~IN nodesIn => Connection.EstablishConnection[source];
            ~Connection.CheckConnection[[source, pkt.rpc.generation]] => NULL;
            ENDCASE => {ok ← TRUE; pkt.rpc.generation ← myGeneration} }; -- Packet okay.

CheckConnection: ENTRY PROCEDURE [candidate: ConnectionRecord] RETURNS [found: BOOLEAN] =
  -- CheckConnection is called by Verify to see if myNode has an existing connection with candidate.
  In addition to looking for an existing connection, CheckConnection also sees if candidate's generation
  is later than oldGeneration. If so, the new generation replaces the old one. This can happen because
  nodesIn is not kept completely accurate. But it can cause no problem for pending calls in activeCalls
  because there can be none from candidate.node: If there were any calls, candidate.node would have
  contacted myNode during its extermination or expiration (candidate.node must crash to increase its
  generation). This will have already caused myNode to increase candidate's generation and
  exterminate the calls, so there can be no call packets left in activeCalls. ActiveCalls can, however,
  contain oldGeneration return packets that candidate.node never acknowledged or that ServerCall used
  to maintain process-to-process connections with candidate.node. In both cases,
  ServerCall.FlushReturns is called to flush any of these candidate-specific return packets.
        { oldGeneration: Generation;
          [found, oldGeneration] ← FindConnection[candidate];
          IF found AND oldGeneration < candidate.generation THEN {
                connections ← connections − [node: candidate.node, generation: oldGeneration];
                connections ← connections + candidate;
                ServerCall.FlushReturns[candidate.node] } };

AddConnection: ENTRY PROCEDURE [new: ConnectionRecord] =
  -- AddConnection is called by EstablishConnection to record new in connections if it is not already
  there. It also updates nodesIn for extermination.
        { IF ~FindConnection[new].found THEN {
                nodesIn ← nodesIn + new.node;
                connections ← connections + new} };
```

```
FindConnection: INTERNAL PROCEDURE [candidate: ConnectionRecord]
        RETURNS [found: BOOLEAN ← FALSE, generation: Generation] =
 -- FindConnection is called by CheckConnection and AddConnection to see if myNode has an existing
 connection with candidate. In addition to just searching for myNode's record of candidate's node and
 generation, FindConnection also returns candidate's recorded generation. The implementation of
 FindConnection given below uses a simple linear search. In practice, since FindConnection is called
 for every incoming packet, it must be implemented extremely efficiently. The microcoded hashing
 approach suggested for ServerCall.FindOtherCaller is one method. Another is to use a hybrid
 approach: For nodes on the local network, represent connections as a vector of generations indexed
 by host. This requires no searching for packets exchanged on the local network, which is most of
 them. For other networks, use a more space-conservative hashing scheme.
    { FOR old IN connections DO
        IF old.node= candidate.node THEN RETURN [found: TRUE, generation: old.generation];
        ENDLOOP };

EstablishConnection: PROCEDURE [node: Node] =
 -- EstablishConnection is called by Verify to make a connection with node. This is done by
 performing a special generationRequest remote call to node. The special call is trivially completed by
 HandleTraffic, below, since only node's current generation is desired. Because no actual procedure is
 called, the procedure inport is NIL. Since multiple generationRequests from the same node can
 complete, AddConnection ignores duplicate connection attempts. (A valuable but missing
 enhancement to EstablishConnection is to exchange, compare, and adjust (via independent time
 authorities) Clock values when connections are made. This extra synchronization insurance makes
 the expiration algorithm much more reliable.)
    { NetworkService.AllocPkt[myProcess, generationRequest];
      ClientCall.RemoteCall[myProcess, [node: node, inport: NIL] ];
      AddConnection[[node, myProcess.pkt.rpc.generation]];
      NetworkService.FreePkt[myProcess] };

HandleTraffic: PROCEDURE [genPkt: Packet] =
 -- HandleTraffic is called by Verify to handle both connection requests from other nodes and
 connection responses that myNode solicited. Generation requests always get myNode's current
 generation returned in response. In addition, the requestor's generation is returned in the inport
 field so that obsolete responses can be discarded. Generation acknowledgements (responses) that are
 not obsolete cause RemoteReturn to be called, completing the connection request made by
 EstablishConnection, above.
    { SELECT genPkt.rpc.type FROM
      generationRequest =>
        {genPkt.rpc.type ← generationAck;
        genPkt.rpc.inport ← genPkt.rpc.generation; -- Return caller's generation back.
        genPkt.rpc.generation ← myGeneration;
        Transmit[genPkt.rpc.callID.source, genPkt] };
      generationAck =>
        IF genPkt.rpc.inport = myGeneration
            THEN ClientCall.RemoteReturn[genPkt.rpc.callID.process, genPkt];
      ENDCASE => ERROR };

END; -- of Connection monitor.
```

Algorithm 5.3: Emissary's high performance remote procedure mechanism.

5.5 Distributed Binding

This section presents a scheme for binding the modules of distributed programs. In the overall Emissary design of section 5.2, the role of distributed binding is to satisfy both the powerful binding and configuration property and the strong typechecking property. In other words, the role of distributed binding is to specify conveniently and assign in a typesafe manner the modules of a distributed program to the nodes of a distributed system.

A distributed programming environment with separately compiled modules is ideal for studying remote binding because each module is usually a self-contained abstraction. The purpose of remote binding is to *name* these distributed abstractions, *specify* their communication relationships, and *connect* them together in the way the programmer desires. One way to resolve these three naming, specification, and connection issues is is with a *configuration language* in which programmers declare module names and their hierarchical relationships. Examples of one particular language appear below, but a rough analogy is provided by the linker or loader *command files* (i.e., configuration descriptions) that are used to combine separately compiled programs into executable units on most batch and timesharing systems.

The remote binder discussed in this section is constructed by extending a single-machine binder. To make this approach maximally effective, a concrete environment with a powerful, existing binder had to be chosen. The Mesa binder was selected because Mesa already supports a sophisticated binding scheme for uniprocessor environments. Mesa's *C/Mesa* configuration language is fully described in chapter 7 of the Mesa Manual [62]. While some familiarity with C/Mesa is helpful, it is not necessary. Readers familiar with Euclid [45], for instance, should have no trouble with understanding the examples.

Despite the choice of C/Mesa as a concrete context for a remote binder design, I have not implemented a remote binder and there is no algorithmic specification for a remote binder in this section. There are two reasons for this decision.

> *Binder complexity.* C/Mesa is a rich language whose compiler is a large and complicated program. Reproducing the details of a redesigned, distributed binder is outside the scope of this dissertation. Instead, appropriate models are presented for the critical parts of the binding system. These single-machine models are then extended to perform similar remote functions. The extensions are sketched in sufficient detail to convince the reader that the design is sound.
>
> *Distributed C/Mesa.* The main focus of this remote binding work is on how to gracefully extend C/Mesa to specify distributed configurations of programs. Because C/Mesa is a language, extending it is first a language design problem and second an implementation task. Consequently, the C/Mesa extensions are presented as examples with descriptions of behavior. This section strives to make modest proposals; fortunately, remote interfaces and distributed programs come easily to Mesa.

5.5.1 Background

Mesa's binding process has two distinct steps: binding and loading. Each step deals with configurations of modules. Precise descriptions are given below; the location of each definition in the binding-time spectrum of section 4.1.2.1 is indicated in italics.

> *Configurations.* A *configuration* is either a single compiled Mesa module (a *compile time* atomic configuration) or a group of configurations that has been previously bound into a single configuration (a *link time* nonatomic configuration). A configuration is basically a "relocatable" object program that has been linked together from compiled modules.
>
> *Binding.* The Mesa *binder* is a *link time* compiler that reads a C/Mesa configuration description and constructs the description's specified program by binding together existing atomic and nonatomic configurations. The binder is responsible for typechecking all intermodule interface requirements to ensure that all imported and exported interfaces (abbreviated *imports* and *exports*) are satisfied in a consistent manner. (Interfaces were defined in section 4.1.2.2.) The result of binding is another nonatomic configuration.
>
> *Loading.* The Mesa *loader* is the *static runtime* program that loads configurations into virtual memory by linking them together and resolving the last unbound symbols. The loader typechecks configurations just as the binder does, and it starts a fully bound configuration by transferring to the configuration's CONTROL module (a control module is the first module in a configuration to be executed; it is explicitly specified by the programmer). The loader can also be called by an executing program (at *dynamic runtime*) to alter its configuration dynamically.

For full flexibility, both the binder and the loader must be callable at runtime. This is highly desirable in a distributed system where crashes and partitioning may necessitate arbitrary dynamic reconfigurations of programs that have demanding reliability or performance goals. The programmer who must provide such robust service should obviously have as much power as possible to solve his reconfiguration problems. Giving him the full capabilities of the binder and, especially, of the loader are the first steps. Thus, while most of the following discussion takes place in a pre-execution context, this temporal setting is for the convenience of exposition and is not a restriction. The binder and loader must be runtime procedures as well as tools of the programming environment. Mesa's own binder and loader are available at runtime, for example, although not with the full generality required here.

5.5.2 Dynamic Loading of Configurations

Because the loader performs vital dynamic linking between running and soon-to-run programs, it is essential to understand how it works. Having a good model of its local behavior is especially valuable below, where extensions that make a local loader into a remote one are presented.

5.5.2.1 A Local Loader Model

The current loader's operation is easy to understand by explaining how its basic *Load* operation loads a new configuration. Consider a Mesa program M, an atomic or nonatomic configuration, with imports $I_1,...,I_n$ and exports $E_1,...,E_m$. When *Load*[M] is called, each exported interface E is

added to a master list of *AvailableExports*. In addition, the particular interface components (section 4.1.2.2) of E that M provides are added to the *interface record* for E. (Mesa modules need not export whole interfaces. This permits a collection of modules to implement an abstraction cooperatively by having each module supply just the interface components it implements. The *interface record* of E is the record into which each module's exported components of E are put. When the record is filled, E's abstraction has a complete implementation available.)

Each imported interface I of M is put on the *NeedsImports* list of all modules that still have outstanding interface requirements. The *NeedsImports* list is then scanned to fill (typesafely) as many of those requirements as possible from *AvailableExports*. Because *AvailableExports* is usually augmented by M's own exports, more modules than just M will (usually) have additional import requirements filled during the scan. As soon as M's interface requirements are completely satisfied, M is removed from *NeedsImports*. In this way, a set of modules with some imports and some exports in common can be loaded (in any order) and bound together for execution. Note that the loader acts as an incremental binder in the final part of loading any configuration. Another way to view this is that there is a single runtime configuration into which the loader binds modules for execution.

5.5.2.2 A Remote Loader Model

Extending the local loader to perform remote loading duties is not difficult. The basic scheme just replicates the *AvailableExports* list and all remote interface records on all of the machines participating in a distributed load. Only remote interfaces are included in the distributed version of the list, which is called *AvailableRemoteExports*. As in section 5.3.1 on orphan algorithms, communication between machines is by RPC.

The remote loader operates as follows. Call a set of cooperating distributed programs running on potentially separate machines a *company*. Each machine in a company has a list that contains all the company's members; this list of machines is called *CompanyMembers*. Adding new members to a company is discussed below; for now, assume that the members of a company are fixed and specified by *CompanyMembers*. Assume that a new module M is loaded on machine A, and that M exports remote interface E that has remote procedures or other control transfer components. Machine A's loader then transmits E's remote exports to the entire company with a remote call to a *NewRemoteInterface* procedure on each machine in *CompanyMembers*. In each machine, *NewRemoteInterface* adds E to *AvailableRemoteExports* and puts the new, remote transfer descriptors to E's interface record. Each machine then performs a local binding pass to further bind the modules remaining in *NeedsImports*. In this way, all the machines in *CompanyMembers* participate in a remote *Load* on the member machines.

5.5.2.3 Dynamically Mustering a Set of Programs

To start and grow a company, assume that *CompanyMembers* contains a single program, *Captain*, that is in command. *Captain* may simply be the first program loaded, or it may be the machine

that initiates connections with other machines such as servers, which are already loaded. (The captain's role in a company is analogous to the CONTROL module's role in a local configuration: it represents the company for initialization and startup purposes.) To *muster* a company of programs dynamically, the captain first directs his loader to add all the component machines to the company.

To muster a machine *A* into the company, *Captain* makes a remote *Recruit* call to *A*. (How *Captain* names and locates *A* is described in the next section.) *Recruit*[*A*] checks with host *A* to see if its loader is running or if it is otherwise occupied. If not, *Recruit* invokes a primitive booting operation on *A*. This booting operation is implemented by the network or other low-level software that can give the breath of life to hibernating machines (e.g., a one-packet or single-sector bootstrap program). The breath of life starts *A* and leaves its loader in command.

When *Captain's Recruit* call to *A* is finished, he adds *A* to his list of *CompanyMembers* and remotely calls *AddRecruit*[*A*] on all *CompanyMembers* to replicate the list. Once a new member machine is recruited, the *Captain* (or other *CompanyMembers*) can load programs on it by calling *Load* remotely. This *Load* eventually causes *AvailableRemoteExports* to be updated on all *CompanyMembers*.

Whenever a new independent machine *C* wants to enlist the services of the company—e.g., import a file server company's remote client interface—it reports to the *Captain*. The *Captain*'s loader will automatically inform *C* of the company's *AvailableRemoteExports* and send along the necessary remote interface records. Except for this initial dialog with the *Captain*, the members of a company are indistinguishable. Eventually all the programs in a company will know of one another and will be in the steady-state situation described above.

A natural extension of this scheme allows one to add an already existing company to another. Basically, if captain *A* connects with captain *B* of another company, *A* needs only to add *B* to *CompanyMembers*. Essentially, the (sub)captain *A* acts as a gateway between its company and the *B* company to which it is connected. However, remote calls between members of *A*'s and *B*'s companies are made directly because the *AvailableRemoteExports* lists that *A* and *B* exchange specify the actual exporting machines, not the captains.

5.5.3 Static Binding of Configurations

Because of its anarchy and inefficiency, completely dynamic loading has been supplemented with static configuration binding in Mesa. Similarly, dynamically mustering a company of programs can often be pleasantly replaced with more declarative binding. The rewards of binding are several:

> The hierarchical relationships between modules are declared in a language designed to make those relationships explicit and visible: the structure is not hidden in a confusing tangle of *Load* procedure calls. In addition, the binder can locate a great many errors, such as missing or multiply defined imports or exports, that would otherwise cause errors at runtime.
>
> Much of the loader's work can be done in advance of actual execution by the binder. For instance, subconfigurations of modules that cooperatively export an abstraction can always

be bound in advance because the modules are a logical unit. In this case a client of the abstraction never wants to deal with individual modules, just the whole unit.

Static binding, however, naturally tends to impose a static structure on a configuration. In a distributed system, presuming a static structure for a remote configuration can lead to less robust behavior in the presence of failures. This point was discussed earlier in section 5.5.1, but it is worth making again: binding too tightly, in advance, can cause problems with *re*configuration. Programmers must adjust the balance of static binding and dynamic loading according to the characteristics of each application.

Despite this drawback, the rewards of binding are well worth pursuing. To make C/Mesa describe distributed configurations two new abilities are needed:

Remote interface naming. The ability to name a remote interface is needed so that remote imports and exports can be matched and satisfied. Fortunately, Mesa already attaches an internet-wide unique identifier to each local or remote interface. The problem of naming a remote interface thus boils down to locating the machine or company captain that exports a remote interface with a matching identifier. The loader handles the rest.

Service and machine naming. The ability to name services and machines is an important part of distributed resource location. Recognizing that service naming and machine naming are independent but related tasks is essential. The name of a file *service*, for instance, might be Juniper [85], but the Juniper service itself might be distributed over any number of distinct *servers* such as *JuniperA*, *JuniperB*, and *JuniperC*. Each of these server names is in turn bound to the internetwork address of its host machine. As an example, consider a machine *M* that wants to use the Juniper file service. Machine *M* first contacts a *clearinghouse* [67] or *registration* [6] service and inquires about the Juniper service. The clearinghouse looks up Juniper in its data base and returns a list of Juniper's servers, {*JuniperA*, *JuniperB*, *JuniperC*}. Machine *M* then asks the clearinghouse to lookup the addresses of the servers, say, {*A*, *B*, *C*}. (Servers that just map machine names to internet addresses are usually called *name servers*, or *name lookup servers* [8].) At this point *M* contacts the Juniper service directly, using one of the addresses from the clearinghouse. In the usual case *M* will simply use the first server and deal with machine *A*. But *M* can also choose a particular Juniper server on the basis of performance, proximity, or some other criterion. Further elaboration of the crucial role of clearinghouse services and indirect name binding is beyond the scope of this discussion.

There are two prototypical examples of distributed configurations that exercise the naming abilities described above. The following two examples will be used to explain the C/Mesa extensions.

Using a remote server. A client program, *Client*, wants to keep company with a server of the Juniper file service.

Commanding a distributed company. A program, *Captain*, wants to muster a company from a set of programs to be loaded on a number of separate machines.

5.5.3.1 Using a Remote Server

Assume that the client interface for Juniper is defined by *JuniperClientDefs*, which will contain interface components for both local and remote use. This allows some of the server's activities to run on the client machine and some on the Juniper machine. (The reader might wish to glance back at the file server example in section 2.1.2.1 at this time.) The module that provides the local Juniper code and exports the local interface components is called *JuniperClientCode*. The remote Juniper components come from a Juniper server. A programmer might write the following *UseServer* configuration to execute his *ClientCode* that calls Juniper. (The syntax in this and later examples is not intended as a firm proposal for C/Mesa.)

```
MACHINE DIRECTORY
        Juniper: FIND "Juniper";

UseServer: CONFIGURATION
        IMPORTS remoteJuniper: REMOTE JuniperClientDefs ON Juniper
        CONTROL ClientCode =
    BEGIN
    localJuniper: JuniperClientDefs ← JuniperClientCode;
    ClientCode; -- Uses interface components from both localJuniper and remoteJuniper.
    END.
```

Now, *JuniperClientCode* exports the *JuniperClientDefs* interface (calling it *localJuniper*), providing the local interface items. Juniper also exports *JuniperClientDefs* (calling it *remoteJuniper*), providing the remote items. Getting part of the interface from within the *UseServer* configuration and part from the outside is already supported by the binder and loader and requires no extra machinery: the parts of the interface to come from the IMPORTed interface record are all those not provided by *JuniperClientCode*.

FIND in the MACHINE DIRECTORY section means that when *UseServer* is eventually loaded, the loader should use a clearinghouse service to map the service name "Juniper" to some server's internet address. In this case, *UseServer* is willing to use any Juniper server. *UseServer*'s loader then contacts the server's loader to obtain the remote parts of the *JuniperClientDefs* interface as described above.

This simple example illustrates how a program gets in contact with the Juniper file service. The same method is used by a program that requires any number of services: The standard name of each service appears in the MACHINE DIRECTORY, and the desired abstract interface(s) from each service appear in the IMPORTS clause. The loader uses a clearinghouse to locate the services and then imports the interfaces from the servers themselves.

Here is a more complicated example where a local *PrintFile* configuration uses two Juniper services—*DirectoryOps* and *FileOps*—to read and print a file using the Printing service's *PagePrinter* interface. There is no local Juniper or Printing code in this example; all interface components are remote.

```
MACHINE DIRECTORY
        Juniper: FIND "Juniper",
        Printer: FIND "Printing";

PrintFile: CONFIGURATION
        IMPORTS
                REMOTE DirectoryOps, FileOps ON Juniper,
                REMOTE PagePrinter ON Printer;
        CONTROL ReadAndPrintControl =
  BEGIN
  ReadAndPrintControl; -- Imports DirectoryOps, FileOps, and PagePrinter.
  END.
```

5.5.3.2 Commanding a Distributed Company

Assume that *Captain* and *Subordinate* are two programs that want to form a company on two separate machines. Neither *Captain* nor *Subordinate* is a regular server, so their host machines are not known beforehand and are not registered with a clearinghouse. The C/Mesa configurations for the two programs are:

```
MACHINE DIRECTORY
        SubordinateMachine: ASK,
        CaptainMachine: LOCAL;

Captain: CONFIGURATION
        IMPORTS REMOTE SubordinateDefs ON SubordinateMachine
        EXPORTS REMOTE CaptainDefs ON CaptainMachine
        CONTROL CaptainCode =
  BEGIN
  -- Various other modules...
  CaptainCode; -- Imports SubordinateDefs and exports CaptainDefs.
  END;


MACHINE DIRECTORY
        SubordinateMachine: LOCAL,
        CaptainMachine: ASK;

Subordinate: CONFIGURATION
        IMPORTS REMOTE CaptainDefs ON CaptainMachine
        EXPORTS REMOTE SubordinateDefs ON SubordinateMachine
        CONTROL SubordinateCode =
  BEGIN
  -- Various other modules...
  SubordinateCode; -- Imports CaptainDefs and exports SubordinateDefs.
  END.
```

These configurations are similar to the previous server example. The first difference is that *Captain* and *Subordinate* each export remote interfaces. This allows each partner to call the other remotely. (In the Juniper example, of course, the (missing) configuration defining the Juniper program contained the line "EXPORTS REMOTE *JuniperClientDefs* ON *Juniper*.") The second difference is the use of the LOCAL and ASK keywords in the directory. The keyword LOCAL directs the loader to use the machine on which the program is loaded. ASK tells the loader to query the human user for a machine name or network address, but only if the loader needs to. For example,

if the captain machine asks for *SubordinateMachine* and it is supplied by the user, then the subordinate machine will automatically know about the captain after the captain's loader makes its *Recruit* call to *SubordinateMachine*.

There is another useful keyword, CALL, that specifies a client module to invoke at load time to return the desired machine address. For example,

```
MACHINE DIRECTORY
        SubordinateMachine: CALL FindAVolunteer,
        CaptainMachine: LOCAL.
```

The symmetry of these two configurations leads to some redundancy that can be eliminated. C/Mesa requires that all the imports and exports of a configuration, both local and remote, be completely specified. This requirement is for information hiding reasons. The machine directories, however, can certainly be combined, and the ASK operation replaced with one substantially more powerful. The example below presents the combined configurations, which are together called a DISTRIBUTED CONFIGURATION to indicate that each component can execute on a separate machine. Notice that this example uses nested configurations. This is for clarity; the nested configurations could also be separate ones, independently bound, and included externally.

```
MACHINE DIRECTORY
        SubordinateMachine: ANY,
        CaptainMachine: LOCAL;

Company: DISTRIBUTED CONFIGURATION
        CONTROL Captain =
  BEGIN

   Captain: CONFIGURATION
        IMPORTS REMOTE SubordinateDefs ON SubordinateMachine
        EXPORTS REMOTE CaptainDefs ON CaptainMachine
        CONTROL CaptainCode =
    BEGIN
    -- Various other modules...
    CaptainCode;
    END;

   Subordinate: CONFIGURATION
        IMPORTS REMOTE CaptainDefs ON CaptainMachine
        EXPORTS REMOTE SubordinateDefs ON SubordinateMachine
        CONTROL SubordinateCode =
    BEGIN
    -- Various other modules...
    SubordinateCode;
    END;

  Subordinate;
  Captain;
  END.
```

In the *Company* distributed configuration, *Captain* and *Subordinate* are again to execute on *CaptainMachine* and *SubordinateMachine*. *CaptainMachine* is still the local machine, but *SubordinateMachine* is now declared to be ANY machine. ANY means that, at load time, an available

idle machine should be located and recruited. Idle machines are located through standard means such as a network broadcast or a clearinghouse that keeps a list of idle machines. If there are no idle machines, ANY does an ASK operation instead. Once a free machine is contacted, its two loaders communicate and transfer the *Subordinate* configuration from the *CaptainMachine* to the *SubordinateMachine*. This transfer uses a regular FTP package and need not occur if *Subordinate* is already on the remote machine—say, from a previous execution of *Company*. Finally, the loaders actually *Load* the modules in *Company*, exchange remote exports, and start each configuration's CONTROL module.

The example above has only two machines and uses no remote services. In general, a statically bound company can have any fixed number of machines and use arbitrary services such as Juniper and Printing. Constructing these more elaborate distributed configurations is straightforward.

5.5.4 Reflections

The distributed extensions proposed for the the C/Mesa language cover the common cases of using servers and mustering companies quite handily. Furthermore, the remote loader model of section 5.5.2.2 is adequate for all of the necessary runtime operations: locating idle machines and standard services with the help of clearinghouses, and typesafely satisfying all remote interface requirements within a given company. Covering just these two common cases, however, has left some missing details:

> *Binding indefinite companies.* The C/Mesa extensions discussed above do not permit a company to be mustered from an indefinite number of members, a restriction which exists for the local C/Mesa language also. The current solution to this problem in the local case is to instantiate new configurations by calling the loader; this solution is recommended for mustering remote configurations as well. Programmers name and use dynamic remote configurations by qualifying operation names with the configuration handles returned by the *Load* operation; this syntax is akin to that of the Emissary orphan algorithms. Dynamically dispatching new members of a company resembles the action of the worm program discussed in section 2.4.3.
>
> *Location transparency.* The machine naming conventions suggested here give a reasonable amount of location transparency because of the indirection through clearinghouses. For instance, asking for "Juniper" can return any member of Juniper's company. While this works well for starting companies, it gives no location transparency over crashes. If a particular Juniper machine goes down, for example, any programs talking to it will get the *Failed* exception. To handle *Failed*, each client program will probably recontact the clearinghouse service and ask for another member of Juniper's company. Of course, if a Juniper crash also crashes the client, then restarting (reloading) the client program will automatically rebind Juniper. Thus, while some location transparency is available from the remote binder, it gives no robust help with crashes.
>
> *Reconfiguration.* Just as distributed C/Mesa offers no location transparency over crashes, so it incorporates no notion of automatic reconfiguration after crashes either. Reconfiguration is a hard research problem in itself; the C/Mesa extensions given here explicitly avoid this problem by making unembellished single-machine binding facilities available in a distributed environment. Programmers must achieve robust behavior by calling the remote binder and loader at runtime to alter their companies on the fly.

Initialization and finalization. Not much attention has been paid to how distributed configurations are started and destroyed. There are two reasons for this. First, Mesa and C/Mesa have a standard initialization mechanism that uses CONTROL modules. This mechanism works in the distributed case too. Second, for destroying either modules or configurations, Mesa and C/Mesa give no help whatsoever. This is an oversight in the language and language-level solutions should be designed that will extend naturally the remote case. One possible solution is Euclid's INITIALLY and FINALLY procedures [45]. These routines are automatically called when a module is created and destroyed; Mesa module initialization and C/Mesa CONTROL modules are close to the former, but there is no support of the latter. In the absence of these nice declarative methods, programmers can always arrange the initialization and finalization of companies through explicit but tedious procedure calls.

Interface components. This section deals with general interface components and not specific items such as procedures. This is intentional; the remote loader and binder should handle all the control transfer components that the local binder and loader do. All transfer types are supported in C/Mesa except for PORTS (coroutines), and I have suggested a port binding scheme elsewhere [65].

Heterogeneity. Finally, companies with machine-code mercenaries, Fortran legionnaires, and other foreign language troops are possible. This problem was examined in section 4.1.2.5.

The remote binder and loader proposals made in this section are unimplemented high-level designs. Some primitive binding and clearinghouse operations similar to those discussed in section 5.4.4.4 have been implemented for the testing described in chapter 6, but experience with these operations does not seriously evaluate the overall remote binding approach. Only a full design and implementation will uncover remaining problems.

5.5.4.1 Transparency and the Essential Properties

The goal of this section was to develop an approach to remote binding that satisfies the strong typechecking and powerful binding and configuration properties. This goal has been achieved in two steps.

Typechecking. The binder for distributed C/Mesa is just the local binder with naming extensions that are checked either statically by the same binder or dynamically by the remote loader. The implementation of the remote loader is sketched from a model of the local loader; the runtime binding characteristics of the remote loader are unchanged from the local one. Since the remote loader checks the types of interfaces and implementations in precisely the same fashion as the local one, the strong typechecking property is satisfied.

Binding. The distributed C/Mesa configuration language is extended to specify remote configurations. The justification for using the Mesa-specific setting in this work is Mesa's status as a sophisticated, state-of-the-art programming environment. The C/Mesa language changes allow the programmer to specify remote configurations with almost the same ease that he can specify local ones. Since the suggested changes are in the spirit of C/Mesa, the powerful binding and configuration property is satisfied.

5.6 Summary

This chapter presented design approaches for Emissary, a transparent remote procedure mechanism whose semantics satisfy the five essential properties. Emissary's design was split into three parts for this task: orphan algorithms, remote call mechanisms, and distributed binding. The first and third parts, while described in detail, are unimplemented design approaches for performing crash recovery and module binding in distributed systems. The second part, remote call mechanisms, is also unimplemented, but is fully specified and is based on the implementation and evaluation of several operational RPC mechanisms.

Mbengga—Fiji
18° 25′ S 178° 09′ E
In an ancient ritual, Matanggali firewalkers tred red-hot stones harmlessly

6

Performance Evaluation of a Family of Mechanisms

This chapter studies efficiency issues of remote procedure call; it addresses the good performance property.

Empirical performance measurements are obtained by experimenting with a series of related, operational RPC mechanisms. The performance analysis of one mechanism is used to hypothesize optimizations which are then implemented and tested on the next. Each mechanism is programmed in Mesa and executes on a personal computer communicating over the Ethernet. There are five generations in this family of mechanisms: Envoy-Diplomat, Stubs, Liaison, EtherPkt, and EtherPktMC, each much faster than its predecessor. The last three generations have multiple members, and each succeeding member is refined and faster than its older siblings.

The discoveries and techniques used to tune and optimize these mechanisms are a vital part of the thesis. No single, guiding optimization principle emerges. Rather, a set of principles emerges, each of which significantly increases performance when applied appropriately. As a consequence of this, the results of the performance evaluation are presented as a group of general *performance lessons*. These lessons were incorporated into the Emissary design of the previous chapter.

6.1 Family History

Before discussing the performance evaluation, the RPC family is introduced with a brief description of each member. The supporting cast of processors is described as well.

6.1.1 Processors

All of the software tests described in this chapter execute on experimental computers built by research divisions of Xerox Corporation.

The *Alto* [89] is a medium-speed microprogrammed personal computer. It has a high resolution display, 2.5 megabyte local disk, and an Ethernet interface. The Alto has emulators for a number of languages, including Bcpl, Lisp, Mesa, and Smalltalk. No RPC testing was done with Altos; this description is included only for completeness.

The *Dolphin* [60] is a successor of the Alto. The Dolphin processor executes Mesa with performance comparable to the Alto, but it has a larger virtual memory and a display that steals no memory bandwidth from the CPU. Typical Mesa intermodule procedure call times on a Dolphin are as follows: null call, 40 microseconds; one argument/result call, 48 microseconds. Forking a detached process takes 1.8 milliseconds.

The *Dorado* [50] is a high performance successor to the Alto and Dolphin. It uses caching and pipelining to execute Mesa about 8-10 times faster than a Dolphin. Typical Mesa intermodule procedure call times on the Dorado are as follows: null call, 6.5 microseconds; one argument/result call, 7.1 microseconds.

6.1.2 Communication

The prototype *Ethernet* is a 2.94 megabit/second local packet network (the production Ethernet has a 10 megabit bandwidth). It uses carrier sensing and collision detection to achieve very low error rates—less than one percent—in normal operation. The Ethernet is a level 0 packet transport mechanism in the Pup internet hierarchy of figure 2.5. The unreliable datagrams of level 1, bytestreams of level 2, and so forth can all be built using the Ethernet as a base. Familiarity with these levels, and the characteristics of their software interfaces as described in section 2.1.5.3, is assumed in the rest of this chapter.

6.1.3 RPC Mechanisms

This section gives a sketch of the five RPC mechanisms. Each is based on a stub translator and its associated runtime machinery, with Envoy-Diplomat taking the most unusual approach. The names of these mechanisms can be somewhat confusing at first, but the mixture of diplomatic and functional terms has a good historic basis and is superior to an enumerative naming such as *Scheme1*, *Scheme2*, etc. A tabular summary of the mechanisms' important characteristics appears at the end of the section.

6.1.3.1 Envoy

Envoy is a remote procedure call facility very similar to the Distributed Programming System (DPS) of section 3.2.2. Envoy uses the bytestream interface—level 2—of the internet software. It has its own marshaling scheme, described below, that can marshal most Mesa datatypes.

Like DPS, Envoy does not provide syntactic transparency. Instead, Envoy provides clients with very high-level primitive operations that can be used to implement and invoke remote procedures easily. For example, Envoy's remote invocation operation is

CallRemoteProcedure: PROCEDURE [*proc*: *RemoteProcedure*, *args*, *results*: *Parameters*].

Envoy's *RemoteProcedures* are distinguished procedure values (similar to Emissary's *ProcedureDescriptors*) that are defined in an exporting server and obtained at bind time by importing clients.

Envoy's parameter marshaling scheme differs markedly from Emissary's precompiled approach; it dynamically interprets procedurally encoded type descriptions to marshal and unmarshal data. For example, consider a procedure with a pointer argument:

P: PROCEDURE [*i*: INTEGER, *pc*: POINTER TO CARDINAL].

The type information that Envoy needs to marshal *P* is supplied by the following auxiliary description procedure that the programmer must supply:

DescriptionOfP: PROCEDURE [*marshal*: POINTER TO *MarshalingRoutines*] =
{ *marshal*.*MarshalPointer*[*pc*, SIZE[CARDINAL]] }.

DescriptionOfP tells Envoy that *P*'s argument record has one component, *pc*, that needs marshaling in addition to the argument record itself. The description also tells Envoy the size (in words) of *pc*'s referent. Unmarshaling is handled with the same description procedure by binding the *marshal* argument to a record of *UnmarshalingRoutines*, thus mapping the call of *MarshalPointer* into one of *UnmarshalPointer*.

This example is a slight simplification, but it conveys the flavor of the scheme. Nested structures are handled recursively; for instance, if *pc↑* were a record that contained other pointers. Envoy's marshaling scheme is unwieldy and rapidly becomes difficult to read, but it is an alternative to Emissary's approach that can be evaluated.

Envoy executes in an environment significantly different from its other RPC family members. As a result, absolute speed measurements of Envoy are not comparable with most measurements of this chapter. Fortunately, however, Envoy-related statistics are relevant in the marshaling discussion.

6.1.3.2 Diplomat

Diplomat is a stub generator. Instead of generating the low-level stubs suggested by the Emissary design, Diplomat writes an implementation with Envoy as its target machine. Together, the Envoy-Diplomat pair define a reasonably transparent RPC mechanism. Diplomat's stubs use Envoy's marshal operations to perform transparent marshaling of all types except procedure parameters and recursive lists (that is, structures with pointers are marshaled to a bounded depth, but not to an unbounded depth as required for general list structures). The design of this mechanism is much different from Emissary, and its impact on performance is discussed below.

6.1.3.3 Stubs

Stubs is another stub generator. The Stubs translator reads a Mesa interface and writes an implementation that, like Envoy, uses a level 2 bytestream as its communication medium. Stubs uses this bulk data transport layer rather than the lower level that Emissary uses or the higher one that Envoy uses. Stubs's transparency is considerably less than Diplomat's because the marshaling is not very general. Strings and array descriptors are the only complex types that are handled. Marshaling aside, the basic software structures of Envoy-Diplomat and Stubs are remarkably similar because they both use a standard bytestream for communication.

6.1.3.4 Liaison and PktStream

Liaison is a translator refined from Stubs. Its target is a special *PktStream* bytestream implementation that is significantly faster than Pup bytestreams. PktStream accesses the Pup internet through the level 1 socket interface and provides highly RPC-tuned stream operations. Liaison's marshaling is identical to Stubs's.

6.1.3.5 EtherPkt

EtherPkt (Ethernet packets) is a set of RPC stub modules that are manually derived from Liaison counterparts. EtherPkt stubs send level 0 Ethernet packets—not level 1 Pup datagrams—using a special interface to the Pup Ethernet drivers. EtherPkt eliminates all of the Pup overhead for remote calls, but leaves the Pup package running for other clients. The EtherPkt stubs were written by hand, not by a stub translator. Transparency is therefore poor, and marshaling nonexistent.

6.1.3.6 EtherPktMC

EtherPktMC is a variant of EtherPkt whose stubs call microcode (*MC*) routines instead of modified level 0 drivers. EtherPktMC runs on Dorados and, with microcode transliterated into Mesa, on Dolphins. It does not coexist with the Pup package. EtherPktMC is an attempt to explore the optimized case of *RTransfer* microcode (section 2.1.3.1) sending call and return packets directly.

6.1.3.7 Profile of Characteristics

Table 6.1 captures the important characteristics of each family member in tabular form. The first six characteristics are associated with the corresponding essential properties (the fifth property is split into two characteristics—*Standard Concurrency* and *Remote Exceptions*). The last five categories indicate some additional characteristics of the implementations.

Characteristic:	*Mechanism:*					
	Envoy	*Diplomat*	*Stubs*	*Liaison*	*EtherPkt*	*EtherPktMC*
Call Semantics	Exactly-once	Exactly-once	Exactly-once	Last-one	Exactly-once	Exactly-once
Binding Method	Explicit	Explicit	Automatic	Automatic	Automatic	Automatic
Parameter Functionality	Excellent	Very Good	Fair	Fair	Poor	Poor
Uniform Typechecking	No	Yes	Yes	Yes	Yes	Yes
Standard Concurrency	Yes	Yes	Yes	Yes	Yes	Yes
Remote Exceptions	Yes	Yes	No	No	No	No
Client View of RPC	Package	Stubs (Envoy)	Stubs (Mesa)	Stubs (Mesa)	Stubs (Mesa)	Stubs (Mesa)
Communication Software	Bytestream	Uses Envoy	Pup stream	PktStream	Pup driver	Ethernet
Pup Level of Comm. Soft.	2	3	2	1	0	0
Marshaling Method	See text	Uses Envoy	Compiled	Compiled	Manual	Manual
Overall Transparency	None	Very Good	Good	Good	Poor	Poor

Table 6.1: Summary of main RPC family characteristics.

Remarks on this table:

Binding Method. Explicit means that the machines executing the desired remote modules must be explicitly named at runtime. *Automatic* means that the machines executing the desired modules are implicitly located through a name server. The actual binding itself uses the simple stub-oriented scheme described in section 5.4.4.4.

Parameter Functionality and Overall Transparency. The judgments expressed in these categories are subjective. They are based on the previous descriptions and my personal experience.

Remote Exceptions. Yes means that remote exceptions are available at the language level. *No* means that they are not available because of implementation laziness, not because of inherent difficulty.

6.2 Benchmark Description

The family of five RPC mechanisms discussed above was evaluated by measuring the behavior of each mechanism as it executed a standard set of benchmark procedures. Two kinds of measurements were made: remote call timings and program counter (PC) histograms.

6.2.1 Benchmark Procedures

The *ParamTest* program that defines the implementations of the benchmark procedures is presented below. Since the goal of testing was to measure the overhead of a remote call, the body of each procedure does nothing more than return its arguments. Because the *StringDescriptor* procedure must convert between STRING and DESCRIPTOR types, it is actually more complicated than indicated.

```
ParamTest: PROGRAM = {
    Null: PROCEDURE = { NULL };
    One: PROCEDURE [in: CARDINAL] RETURNS [echo: CARDINAL] = { RETURN[in] };
    Four: PROCEDURE [a,b,c,d: CARDINAL] RETURNS [w,x,y,z: CARDINAL] = { RETURN[a,b,c,d] };
    TwentyArray: PROCEDURE [in: ARRAY [0..20) OF CARDINAL]
            RETURNS [echo: ARRAY [0..20) OF CARDINAL] = { RETURN[in] };
    StringDescriptor: PROCEDURE [string: STRING]
            RETURNS [desc: DESCRIPTOR FOR ARRAY OF CHARACTER] = { RETURN[string] }   }.
```

6.2.2 Timing Methods

Two methods were used to time remote calls. For slow calls taking several milliseconds or more, an averaging technique was used. The following timing program was used for slow remote calls of *Null*.

```
-- Slow call measurement program.
start ← Timer[];
THROUGH [0..5000) DO --remote call of-- Null[] ENDLOOP;
averageCallTime ← (Timer[]-start)/5000.
```

Because of network contention—that is, delays introduced by the hardware's carrier sense and collision detection circuitry—all averaged times could change by a few milliseconds (0–2). In general, several trials of between 5,000–20,000 calls were made of each test; the average time from the fastest run is shown in the tables of section 6.3.

For fast calls taking less than a few milliseconds, individual call times were measured so that network contention and other microcode task overhead could be factored out. The following timing program was used for fast remote calls of *Null*.

```
-- Fast call measurement program.
FOR i IN [0..5000) DO
        start ← Timer[];
        Null[];
        callTime[i] ← Timer[]-start;
        ENDLOOP;
SortArray[callTime]. -- See text.
```

The time reported for a fast call test in section 6.3 is the median of the first ten call times in the sorted *callTime* vector. This median represents the fastest time measured.

Two different *Timers* were used in the tests. On the Dolphin, time was measured with a high-precision realtime clock. This clock has 38 microsecond resolution, so the call timings are precise to a tenth of a millisecond. On the Dorado, time was measured with one of the Dorado's special hardware event counters. This counter has 64 nanosecond resolution, so the Dorado call timings are precise to a microsecond.

6.2.3 PC Histograms

The Mesa *Spy* is a software performance monitor that constructs a symbolic execution histogram by sampling the PC at frequent periodic intervals. The Spy's histogram gives information about the relative time spent at the module, procedure, and statement levels. The Spy was the main tool used to formulate the optimizations that changed one RPC family member into a faster sibling.

6.3 Performance Evaluation

This section contains the results of performance testing. The results are quite dense, and readers who feel overwhelmed may wish to skim this section now and return later when individual numbers are discussed in the next (analysis) section.

The performance data presented here are derived from more comprehensive test results [66]. Abbreviated programs for the Envoy-Diplomat, Liaison, and EtherPkt timing tests appear in appendix 2.

6.3.1 Dolphin Remote Call Times

Table 6.2 contains call timing tests that were performed between two distinct Dolphins under low network load during the middle of the night. The Optimized and Original versions of Liaison and Stubs identify important implementation stages in their performance enhancement. The details of each stage are discussed later.

Mechanism (on Dolphins)	*Roundtrip call times (milliseconds) for procedures:*				
	Null	*One*	*Four*	*TwentyArray*	*StringDescriptor*
Stubs Original (software checksums)	28.7	32.7	44.7	37.8	42.1
Stubs Original (byte operations)	25.2	28.8	40.4	30.3	35.2
Stubs Optimized (word operations)	24.3	27.3	37.5	28.8	33.4
Liaison Original (original PktStream)	11.4	11.8	12.8	14.2	16.8
Liaison Optimized (fast PktStream)	10.1	11.0	12.5	13.1	15.3
EtherPkt	2.0	2.1	2.1	—	—
EtherPktMC	0.8	0.9	0.9	—	—
Measurement Overhead	0.1	0.1	0.1	0.3	0.6

Table 6.2: Dolphin remote call times.

Remarks on this table:

Liaison. The Liaison numbers shown are for highly optimized stubs. In some preliminary tests with original stubs and original PktStream, null calls took 15 milliseconds. Also, for comparison, instantaneous Liaison Optimized null call times were 1.2 milliseconds on the Dorado.

EtherPkt and EtherPktMC. The array and string procedures were not timed with these mechanisms because their more complicated parameters could not be marshaled; only procedures *Null*, *One*, and *Four* were tested.

Measurement Overhead. Measurement overhead in this and succeeding tables indicates the overhead incurred by the measurement program on a *local call.* This number can be subtracted from the call times of individual mechanisms to give a more accurate estimate of the remote call time. It *cannot* be used to compute a speedup factor; use the local call times from section 6.1.1 for this calculation.

Figure 6.1 is a graphic presentation of table 6.2. The surprising hump in the Stubs's curves is explained in section 6.4.7.

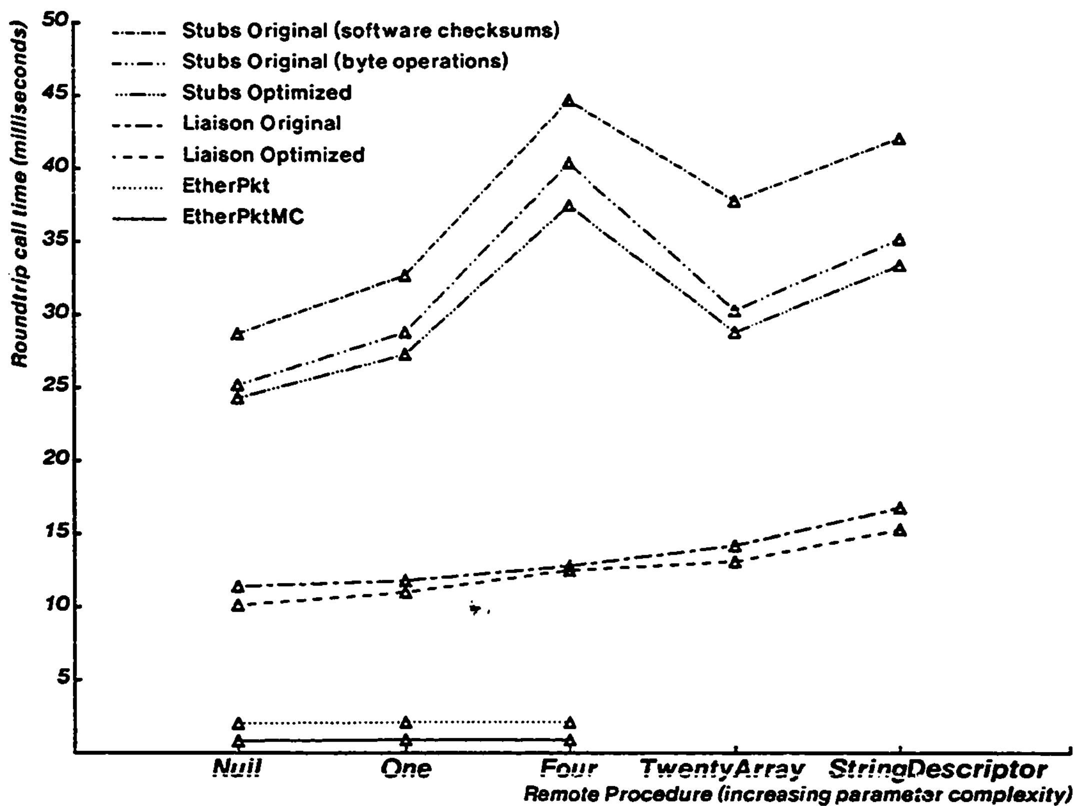


Figure 6.1: Dolphin remote call times.

6.3.2 Dorado Remote Call Times

Table 6.3 gives EtherPkt timings that were performed on both Dolphins and Dorados. Notice that the Dolphin times are repeated from above and are programmed entirely in Mesa (no special microcode). The EtherPktMC implementations identified by BusyWait and ProcessWAIT are discussed later; the AllMesa version is programmed entirely in Mesa and uses no microcode in spite of the *MC* suffix.

Mechanism (on Dorados and Dolphins)	*Roundtrip call times (microseconds) for procedures:*		
	Null	*One*	*Four*
EtherPkt (Dolphin)	2018	2056	2094
EtherPktMC AllMesa (Dolphin)	800	876	876
Measurement Overhead (Dolphin)	38	38	114
EtherPkt (Dorado)	289	303	342
EtherPktMC AllMesa (Dorado)	149	162	200
EtherPktMC ProcessWAIT (Dorado)	145	158	198
EtherPktMC BusyWait (Dorado)	124	137	175
Measurement Overhead (Dorado)	11	11	12

Table 6.3: EtherPkt and EtherPktMC remote call times.

Figure 6.2 is a graphic presentation of the Dorado data in table 6.3. The linearity and nearly identical slope of the four curves indicate that the cost of marshaling and transmitting an additional parameter is the same in all four mechanisms.

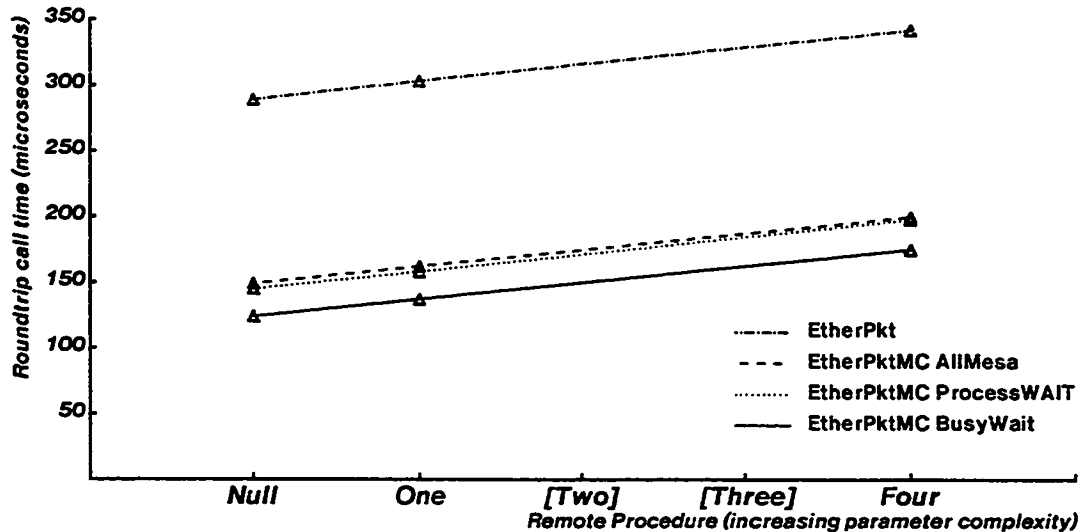


Figure 6.2: Dorado remote call times.

6.3.3 Network Characteristics

Table 6.4 shows how each RPC implementation uses the Pup internetwork. The numbers reflect the *total* activity of a single, roundtrip, steady-state invocation of *Null*. These statistics do not tell the whole story, such as the number of times the data is copied. The versions of Stubs and Liaison that are not shown differed slightly in the number of words sent.

Mechanism	*Internet level and total number of packets & words & xmit time for a Null call:*			
	Pup Level	Packets	Total words in packets: Call (Ack) + Return (Ack)	Xmit time
Stubs Optimized	2	4	61: 17 (14) + 16 (14)	332 μsec.
Liaison Optimized	1	2	30: 15 + 15	163
EtherPkt	0	2	10: 5 + 5	54
EtherPktMC (all versions)	0	2	10: 5 + 5	54
Local Calls	—	0	0	0

Table 6.4: Network characteristics.

Columns of this table:

Total words in packet. This number reflects *all* bits sent on the wire. In particular, for EtherPkt and EtherPktMC it includes Ethernet encapsulation overhead of 3 words for addresses, packet type, and cyclic redundancy check (CRC). For Liaison and Stubs it includes Pup packet header overhead of 11 words (remember the checksum) plus the 3 word encapsulation. (See figure 2.4.) For Stubs Optimized, the numbers in parentheses are the size of the acknowledgement packets sent by the bytestream implementation.

Xmit time. This is the on-the-wire transmission time for *Total words* assuming that there is no deference (i.e., no waiting for packets already on the wire).

6.3.4 Process Utilization

Table 6.5 shows how the Mesa process machinery [47,62] is used by each mechanism. The data are for steady-state calls and do not include the cost of binding and other initial transients. As in table 6.4 on network characteristics, this one also presents biased statistics because the number of process switches gives no indication of how much work each process performs. In Mesa, a process switch occurs when one process WAITing in a monitor is NOTIFYed by another process.

Mechanism	*Total number of FORKs and WAIT-NOTIFY pairs for a single call:*	
	FORKs	Process switches: Client (Ack processing) + Server (Ack)
Stubs Optimized	1	18: 5 (4) + 5 (4)
Liaison Original	1	12: 6 + 6
Liaison Optimized	0	12: 6 + 6
EtherPkt	0	4: 2 + 2
EtherPktMC ProcessWAIT	0	2: 1 + 1
EtherPktMC BusyWait	0	0
Local Calls	0	0

Table 6.5: Process machinery utilization.

Remarks on this table:

Stubs. The number of process switches in Stubs is primarily governed by those performed by the Pup communication software. In particular, on input there are five switches per packet: two in level 0 (one for the naked NOTIFY, or Mesa interrupt [47], and one for the interrupt-to-router handshake), one in level 1 (router-to-socket handshake), and one in level

2 (socket-to-stream transition). Acknowledgement packets, shown in parentheses, involve one less switch because acknowledgements do not have the final socket-to-stream transition.

Liaison. Liaison's six process switches are split evenly between the Pup and PktStream implementations. The three in Pup are described below (levels 0 and 1). Of the three in PktStream, only one is necessary; the other two happen because a nonstandard buffer requeuing procedure must be called on output.

EtherPkt. EtherPkt's two extra switches are the price of cooperation with the Pup package. The Pup driver is activated by a naked NOTIFY, which immediately switches to EtherPkt when an RPC packet is received.

EtherPktMC. The ProcessWAIT version switches on the hardware's naked NOTIFY. The BusyWait version performs no switches because it "knows" the call is a short one and thus spins in a loop rather than blocking.

6.3.5 PC Histograms

Table 6.6 contains PC histogram results for a remote call of benchmark procedure *TwentyArray*. Unlike all previous tests, in this one the client and server are run on the same physical machine so that both client and server execution data can be gathered simultaneously with the Spy. This change in testing procedure perturbs the measurements slightly, but not enough to affect the results. In addition, execution profiles change slightly from run to run since the Spy uses statistical sampling. Finally, because of the wide range in absolute call times, both relative and absolute times are given for each table entry.

TwentyArray: PROCEDURE [*in*: ARRAY [0..20) OF CARDINAL] RETURNS [*echo*: ARRAY [0..20) OF CARDINAL]

Mechanism (on same Dolphin)		*Execution profile of a TwentyArray call:*						
	Time	Total	User	Stubs	RPC Impl	Levels 0&1	Level 2	Other
Envoy-Diplomat	Absolute	*no times*						
	Relative	100.0%	0.9%	2.9%	21.0%	42.7%	13.6%	18.9%
Stubs Optimized	Absolute	28.8 msec.	0.46	2.33	—	10.92	14.08	1.01
	Relative	100.0%	1.6%	8.1%	—	37.9%	48.9%	3.5%
Liaison Optimized	Absolute	13.1 msec.	0.25	0.76	—	6.35	5.46	0.28
	Relative	100.0%	1.9%	5.8%	—	48.5%	41.7%	2.1%
EtherPkt (EtherPktMC)	Absolute	2.0 msec.	0.03	1.75	—	0.20	—	0.02
	Relative	100.0%	1.6%	87.5%	—	9.9%	—	1.0%
Local Calls	Absolute	0.3 msec.	0.24					0.06
	Relative	100.0%	80.0%					20.0%

Table 6.6: *TwentyArray* execution profiles.

Columns of this table:

User. User time is time spent in modules that call and implement the *TwentyArray* procedure. In the *Local Calls* case this time is ideally 100%, but it is reduced by the overhead of maintaining the driver program's small display and log file.

Stubs. Stub time is time spent in the client and server stub modules that map local calls to remote calls.

RPC Impl. Implementation time is time spent in the RPC mechanism when it is a separate package like Envoy. For EtherPkt, Liaison, and Stubs there is no separate RPC implementation and thus the RPC time is included in the *Stubs* column. Implementation time includes relevant time spent in the storage allocator.

Levels 0&1. Level 0 and level 1 time is spent in the driver and socket software.

Level 2. Level 2 time is spent in the bytestream communications software. For Liaison, level 2 time is spent in the PktStream code.

Other. Other time is a measure of Mesa's overhead to support the disk and display for the test program.

Remarks on this table:

Envoy-Diplomat. Envoy-Diplomat times are not shown here because, as noted previously, Envoy's execution environment is significantly different from that of the other family members. This makes absolute time comparisons meaningless.

EtherPkt (& EtherPktMC). The EtherPkt execution profile measures different test conditions, namely, just the client half of an EtherPkt *One* call on a Dorado. The only important thing to notice is the 87.5% stub time, which, because EtherPkt deals directly with the Ethernet driver, includes level 1 and 2 communication time. EtherPktMC has a similar profile.

Figure 6.3 is a graphic presentation of table 6.6. The unusual behavior of the relative-time EtherPkt curve is explained in the remarks above.

6.4 Performance Lessons

By combining knowledge of the various RPC implementations along with the timing and histogram results, we can draw some strong conclusions about the way RPC mechanisms should conduct business. These lessons are presented below. A graphic summary appears in section 6.4.10.

6.4.1 Bytestreams are bad.

Bytestreams are an improper communication model for RPC. The excess data movement and multiple process switches of a standard stream mechanism are much too expensive. In the Pup world, these problems are exacerbated because the standard stream implementation is explicitly streamlined for file transfer operations, which is at the far end of the latency-bandwidth spectrum from RPC.

The best example of this poor performance is given by Liaison Original and Stubs Optimized, where figure 6.1 shows a factor of 2–3 improvement in abandoning Pup bytestreams. This speedup is largely due to the fact that Liaison's PktStream protocol sends two packets per remote call, whereas Pup bytestreams sends four. The extra process switching (table 6.5) and flow control overhead of Pup streams accounts for much of the remaining time.

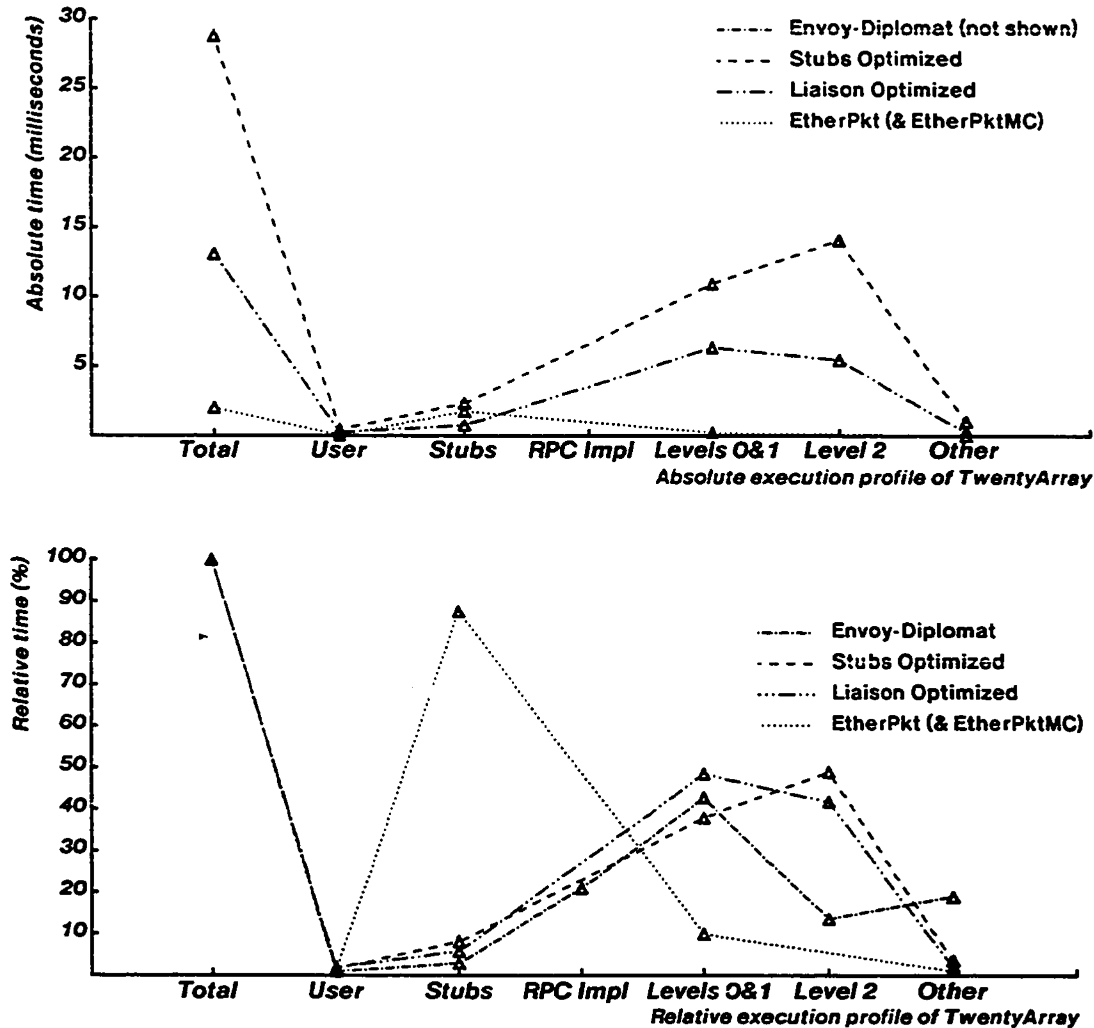


Figure 6.3: *TwentyArray* execution profiles.

6.4.2 Watch for hidden protocol costs.

Stubs measurements uncovered two bytestream areas with unexpectedly bad performance penalties. Any RPC implementation building upon high-level protocols must investigate and consider such costs extremely carefully.

> (*Software*) *checksums.* The significant cost of software checksums is demonstrated by the Stubs Checksums and Byte Operations tests. The only difference between the two is that Pup checksums were turned on in the former. Notice the large 25% and 20% penalties paid in the *TwentyArray* and *StringDescriptor* big parameter cases (table 6.2 and figure 6.1). Less costly microcode-computed checksums are, of course, the way of the future. While abandoning checksums for end-to-end use in an internetwork is foolish, the reliability of the Ethernet's CRC urges abandoning them for EtherPkt-like local network calls.

Byte operations. The penalty of byte-aligned operations on word-oriented machines is demonstrated by the Stubs Byte Operations and Word Operations tests (figure 6.1). The space saved by dealing with bytes is clearly negated by the cost of the slow *ByteBlt* (byte block transfer) when large parameters are misaligned. The lesson is clear—pad out the odd bytes as needed.

6.4.3 Special-purpose protocols are good.

As discussed above, a special-purpose request-response protocol like PktStream is essential for performance that is modest to good. For good to excellent performance, EtherPkt-like optimizations are required for simple local network calls.

While Liaison's performance is much better than Stubs's, Liaison's 10 millisecond *Null* call time is still large. A look inside the Liaison Optimized implementation for the *Null* call case—not the *TwentyArray* case shown in table 6.6 and figure 6.3—shows that 10% of the time is spent in the level 0 Ethernet drivers, 53% in the level 1 Pup datagram interface, and 34% in the level 2 PktStream implementation. This 87% level 1 and 2 overhead—8.8 milliseconds—is a substantial price to pay for stream and internet functionality that is infrequently used when most calls fit into one packet that is sent to a server on the local network. Of course, Liaison's generality is still needed for multipacket and internet calls, but optimizing the most frequent case is important.

EtherPkt and EtherPktMC explore these optimizations by reliably sending and sequencing their *own* call packets directly over the Ethernet. By discarding both levels 1 and 2—87% in Liaison—EtherPkt eliminates 8.1 of 8.8 wasted milliseconds and achieves a fivefold improvement over Liaison Optimized (figure 6.1). This reclaims all of the 53% internet overhead and most of the 34% PktStream overhead (similar to *TwentyArray* in figure 6.3). EtherPktMC, which takes over the Ethernet interface rather than sharing it with Pup as EtherPkt does, manages to do nearly three times better: By cutting the number of process switches in half and further reducing overhead, EtherPktMC achieves a twelvefold increase over Liaison.

In a working implementation, cooperation between the RPC mechanism and the internet software is probably required. This can be obtained without sacrificing much of EtherPktMC's speed by incorporating a microcoded Pup and RPC packet demultiplexor in the Ethernet interface (i.e., implementing two logical Ethernet interfaces). Thus, while EtherPkt and EtherPktMC code take some shortcuts that would be unacceptable in a real implementation, such level 0 local network optimizations are clearly necessary for excellent performance.

6.4.4 Use microcode for exceptional performance.

Moving more of these local network optimizations into microcode will yield extreme efficiency. EtherPktMC experience on the Dorado indicates that call times of close to 100 microseconds are possible if stubs are abandoned and the compiled Emissary approach is used (table 6.3 and figure 6.2). The Emissary stub scheme may be acceptable, too, if more of the stubs' activities are microcoded (EtherPktMC microcodes only *Send* and *Receive*).

On the Dolphin, the 800–900 microsecond EtherPktMC Mesa "microcode" times can be considerably reduced by microcoding. Using a conservative factor of five speedup for the Mesa-to-Dolphin microcode change, Dolphin times of around 200 microseconds are possible. (This scaling is not linear—that is, not 800/5—because the Ethernet does not scale at all: Sending the ten words that EtherPkt transmits for a complete null call takes 54 microseconds in any case. The ten words (five for the *call* and five for the *return*) are [*destinationAndSource*, *packetType*, *serialNumber*, *procedureDescriptor*, *etherCRC*].)

This 200 microsecond estimate is reinforced by Spector's remote read/write memory work, described in chapter 3. On the Alto, which is comparable to the Dolphin, he performs simple "remote calls" of read and write in 155 microseconds with a completely microcoded implementation. Spector's remote memory operations are simpler than our remote calls, but his times easily establish the 200 microsecond ballpark.

Whatever protocol and optimizations are used by an RPC mechanism, it must always take account of these two issues:

> *No "optimized" semantics.* Any RPC optimizations must have semantics that are identical to those of the standard mechanism (e.g., the one used for internet RPC). A microcoded call failure must recoverably trap back to the Liaison (or equivalent) layer.
>
> *Minimum-state idle connections.* Attention must be paid to minimizing the information that one machine needs to retain about its idle importers and exporters. RPC mechanisms will probably not use bytestream-like connections, and whatever style of connection is used must be designed to have a small quiescent state. For example, one hundred already-bound but infrequently used remote interfaces should have modest storage demands.

6.4.5 Caches are very important.

The speedup of Liaison Optimized over Liaison Original (figure 6.1) is primarily due to caching two expensive objects: processes and stream objects. In Liaison Original a stream object and corresponding process were created for each call; these operations took 4% and 17% of the *Null* call time, respectively. In the optimized version these were reused when possible, reducing the time to less than 1%.

EtherPkt also uses a process cache. In addition, it keeps a cache of fixed packet buffers because the Pup buffer queue machinery is too expensive, costing over 10% at times. EtherPktMC uses a similar strategy that involves one less copy operation.

The caching concepts can, of course, be applied to any layer of the system. One way to get EtherPkt-like speedups in Liaison is to keep a cache of prerouted packet buffers. This gives full internet flexibility for slightly greater software cost and an extra 120 microseconds of transmission time for the Pup header on the wire. In this case, a remote call pays for any gateway delay (several milliseconds, when needed) but not for the 53% route-and-dispatch overhead on *every* packet. Suitable cache invalidation schemes must be used, but even the worst case is not too bad: If a retransmission is necessary, perform it through the normal internet layer on the assumption that

routing changed. It has been suggested that gateways might assume more of the internet RPC routing responsibilities, relieving the local machine of this overhead and thus speeding up this case without local caching. This case will be difficult to press until remote procedure calls become *very* popular.

6.4.6 Marshal by compiling inline, not interpreting out-of-line.

Generating inline parameter marshaling code, as Emissary, Liaison, and Stubs do, is much more time-efficient than the interpretive procedure-driven marshaling scheme that Envoy uses.

The marshaling data in table 6.7 are for a remote call of the *StringDescriptor* benchmark procedure. The figures include the time spent allocating and deallocating the string and descriptor. EtherPkt and EtherPktMC are excluded because they do not handle complicated parameters.

StringDescriptor: PROCEDURE [*string*: STRING] RETURNS [*desc*: DESCRIPTOR FOR ARRAY OF CHARACTER]

Mechanism (on Dolphin)	*Relative time for a StringDescriptor call:* Marshaling
Envoy-Diplomat (Standard Protocol)	11.4%
Envoy-Diplomat (Tuned Protocol)	10.7
Stubs Optimized	8.5
Liaison Optimized	8.5

Table 6.7: *StringDescriptor* marshaling times.

These relative times are actually deceiving: Although Envoy-Diplomat's execution environment is very different from Liaison's, some informal *absolute* comparisons show that Liaison's *marshaling* is 3–5 times faster than Envoy-Diplomat's. This significant difference is not surprising since Envoy is interpreting, at runtime, a marshaling description that Liaison efficiently compiles in advance. In particular, the difference between the inline and out-of-line marshaling times is due to two factors:

The interpretive Envoy description scheme invokes tens of procedure calls (costing at least 50 microseconds each) that Liaison performs inline or not at all.

The Envoy approach defends itself against user errors that Liaison does not check because it cannot commit them as long as its translation (compilation) is robust.

On the other hand, the interpretive out-of-line approach does have these advantages:

It can use significantly less code space, which is very important for some applications.

It is probably comparably efficient for complicated marshaling, such as a list whose nodes are variant records that must be individually allocated anew in the remote machine.

6.4.7 Marshal large blocks, not small ones.

Stubs and Liaison handle one-word parameters individually by calling the stream's *GetWord* and *PutWord* operations once for each parameter. This takes 11% of the time even in the Liaison Optimized *One* call case. Envoy, on the other hand, sends the entire parameter record with one call to *PutBlock* no matter what is inside it. This explains why the Liaison times increase so much—from 10.1 to 12.5 milliseconds—in the first three columns of table 6.2. The corresponding Stubs times increase from 24.3 to 37.5—an incredible 13.2 milliseconds—because the standard stream's *GetWord* and *PutWord* are extraordinarily more expensive than the streamlined PktStream operations that Liaison uses. (Look at the hump in the Stub's curves of figure 6.1). The lesson here is to watch data partitioning very carefully. Stream operations are expensive; if they must be used, move large blocks so the overhead is acceptable.

6.4.8 Select your data protocol carefully.

The two Envoy-Diplomat entries in table 6.7 indicate that using a data transmission protocol tuned for homogeneous language RPC is faster than using a standard protocol for heterogeneous use. This is an expected result, but the reason is informative and gives some hints on increasing performance. The fundamental issue is the *order* in which parameters must be marshaled. Different orders lead to different flattened representations—data protocols—and these representations can be handled with different efficiencies.

Consider, for example, the parameter record

parameterRecord: RECORD [*array*: ARRAY ..., *string*: STRING, *record*: RECORD [...]].

Envoy's standard protocol, which is very similar to DPS's, requires a *strictly* depth-first enumeration of the parameter structure. Because a Mesa string is implemented with a pointer, i.e.,

```
STRING: TYPE = POINTER TO --StringBody:-- RECORD [
    length, maxlength: CARDINAL,
    text: PACKED ARRAY [0..0) OF CHARACTER ],
```

Envoy-Diplomat Standard must traverse *parameterRecord* depth-first and send three separate fragments: *array*, *string*↑, and *record*. Envoy-Diplomat Tuned, on the other hand, chooses to traverse breadth-first and thus sends just the two fragments *parameterRecord*, *string*↑. The tuned implementation therefore saves a call to *PutBlock* (or the setup cost of a third copy operation) for the negligible expense of sending *string*'s pointer in *parameterRecord* needlessly. Because *StringBody* must be dynamically allocated in the receiver during unmarshaling, *string's* pointer must always be changed to point to the new storage. The standard protocol implementation does this adjustment at the same time as it allocates *StringBody*; the tuned protocol version simply retrofits the pointer into *parameterRecord* after both *parameterRecord* and *StringBody* are in place.

At first glance the additional fragmentation incurred by the standard protocol appears costly only because of the overhead for the extra *PutBlock* or copy. Inside the standard implementation, however, there is another penalty: the type information that programmers (or Diplomat) give to

Envoy via the type description procedures is barely adequate to perform remote unmarshaling. Envoy-Diplomat Standard must build and search a nontrivial mapping structure that the tuned version, with its breadth-first marshaling protocol, avoids entirely. (The implementation reasons are beyond the scope of this discussion: see the last point, below.) I offer three observations here:

> This *parameterRecord* example is a very simple one. More deeply embedded structures increase this overhead exponentially, such as if *record* contains records of strings.
>
> Standard representations and protocols are absolutely necessary for heterogeneous clients; abandoning them in favor of homogeneously tuned ones such as Envoy-Diplomat Tuned's is often not acceptable. A hybrid scheme that negotiates the data protocol—for example, standard for interlanguage calls, tuned for intralanguage calls—is a higher-performance solution. This negotiation should probably occur at remote interface binding time.
>
> Finally, the previous *inline* lesson is well applied in this case: Inline marshaling, with its statically compiled rather than dynamically interpreted type descriptions, eliminates all of the standard protocol's mapping overhead no matter how many fragments are sent. Negotiation can still be used to advantage, but it will be somewhat expensive in code space since both kinds of marshaling need to be generated and an appropriate dispatch made.

The lesson about data protocols is this: Select a marshaling method and its corresponding protocol only after careful consideration of both the external heterogeneity of the distributed environment and the internal data representations used by the environment's programming languages.

6.4.9 Avoid copying whenever possible.

Marshaling large blocks is not effective unless the blocks are manipulated with as little copying as possible. Similarly, the nonparameter parts of an RPC mechanism's *call* and *return* packets should be handled with little or no copying.

EtherPktMC performs exceptionally well in this regard because it constructs, sends, and receives packets with no copying at all: parameters are moved (assigned) from the local frame of the stub directly to and from the final packet. (This could, of course, be called one copy. To eliminate it the compiler would have to push remote call arguments directly into the call packet and pop results from the return packet. This approach was used by Emissary in chapter 5.) Further, the constant parts of the packets are initialized only once; this saves additional time. EtherPkt uses a similar technique, but input packets are copied once, from the Pup driver's input buffer into EtherPkt's fixed input buffer.

An intermediate approach is used by Envoy. The header portions of *call* and *return* messages are preallocated records. This record, when complete (requiring one copy), is sent with a single *PutBlock* that is followed by a *PutBlock* of the entire parameter record (requiring no copying). Any parameters requiring additional marshaling are sent immediately thereafter. The *PutBlock* and *GetBlock* operations themselves also involve a single copy into the packet. This is acceptable for large blocks.

The worst copying approach is used by Liaison and Stubs. Both the RPC header and the parameter body are sent piecewise with *PutWord* and *PutBlock*. The data is copied only once, but the overhead is unacceptable. Stubs also copies the argument record one additional time because of an extra level of call-by-value procedure call.

The copying lesson is this: Almost always trade the increased storage cost of a whole buffer for reduced computation. If possible, make the buffer be the final packet itself. Barring this, build an image of the final buffer in the stub and then send it as a block. In either case, let this buffer *be* the desired structure and perform all operations directly upon it.

6.4.10 Summary of Performance Lessons

Here is a recapitulation of the nine performance lessons. They are not ordered by importance.

1. Bytestreams are bad.
2. Watch for hidden protocol costs.
3. Special-purpose protocols are good.
4. Use microcode for exceptional performance.
5. Caches are very important.
6. Marshal by compiling inline, not interpreting out-of-line.
7. Marshal large blocks, not small ones.
8. Select your data protocol carefully.
9. Avoid copying whenever possible.

A graphic illustration of the performance lessons' impact on Dolphin RPC appears in figure 6.4. The data are from the *Null* call times of table 6.2.

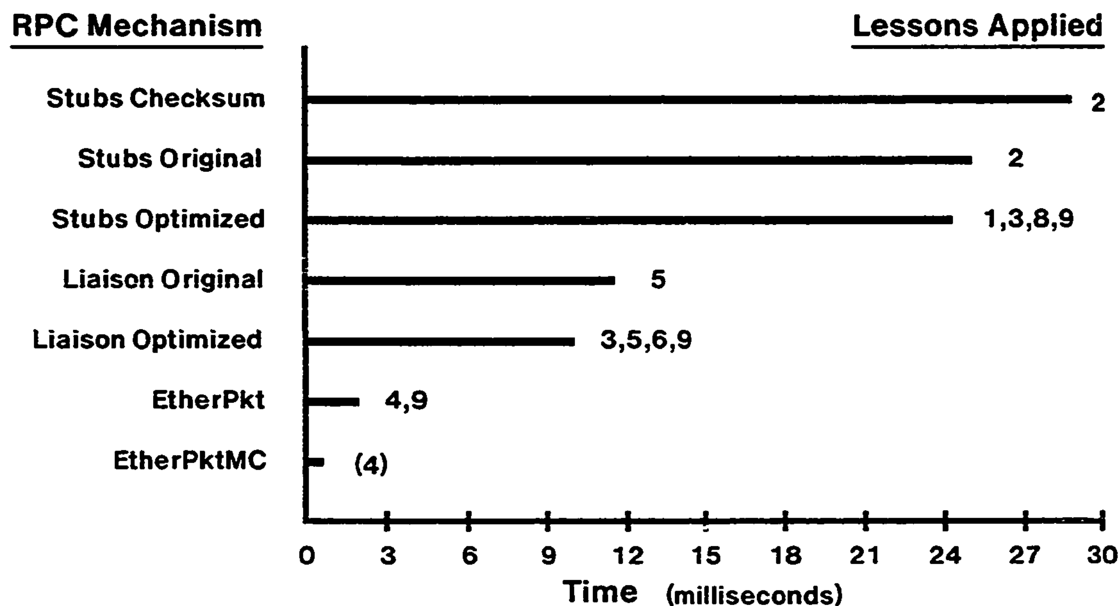


Figure 6.4: Performance comparison of the evaluated mechanisms.

The details of each performance step in figure 6.4 were discussed earlier. Looking at the whole picture, a conclusion worth emphasizing is that each step is very important. For example, the 900 microsecond decrease between Stubs Original and Stubs Optimized *appears* inconsequential when compared with its neighbor 12.9 and 3.5 millisecond decreases. But when compared against the final 800 microsecond EtherPktMC complete call time, the 900 microsecond decrease is significant indeed. To achieve a speed increase of an order of magnitude or more—in this case, a factor of 35—attention must be paid to even the smallest details.

Considered together, these performance lessons define a set of rules that can be used to optimize any communication mechanism that uses a narrow communication channel—and this certainly includes remote messages as well as remote procedures. In a different light, these lessons can be viewed as specific instances of general program optimization techniques. Bentley [4] describes a comprehensive set of these techniques in his excellent study on writing efficient code. The interested reader is referred there for more information.

6.5 Functionality Lessons

In the course of implementing and using these RPC mechanisms, four important functional considerations emerged in addition to the previous performance lessons. These considerations, presented here, serve as experimental verification of some of the essential remote procedure properties.

6.5.1 Named binding is important.

A major difference between Liaison and Envoy-Diplomat is that Liaison's binding routines implement an automatic binding between client and server. This is accomplished with a special Pup name lookup service that operates in the serving RPC machine. A *client* RPC machine attempting to import interface *I* asks the network name lookup services to locate a resource whose name is Mesa's unique ID for *I*. The stripped-down lookup service in the RPC *server* answers this question with its network address, completing the automatic binding without human assistance.

Envoy-Diplomat does not tackle automatic binding, and programmers must specify explicit network addresses when they import or export a remote interface. This typically means that humans must type in the addresses of host machines, although it is possible to wire-in the addresses of well-known servers as long they do not change location. This leads to extreme inconvenience during system development and checkout; even Liaison's very primitive binding scheme is superior to Envoy-Diplomat's static method.

The lesson here is that the powerful binding and configuration property really is essential. Full and flexible binding facilities require clearinghouse services so that *names*, rather than *addresses*, can be used. Both the Liaison and Envoy-Diplomat schemes are inferior to the distributed binding approach discussed in section 5.5.

6.5.2 Clients want full parameter functionality.

A lesson quickly learned from Envoy-Diplomat is that RPC clients want complete support for all Mesa datatypes. For example, while Envoy-Diplomat is designed to handle all datatypes, the current implementation marshals everything *except* procedure types and recursive list structures. One important client, however—a database system—is unable to use Diplomat exactly because the database's remote interfaces make extensive use of both procedure parameters and tree-structured data. There is no question that procedure types should be fully handled, although there is more uncertainty about automatically flattening and transmitting list structures—an extremely expensive operation, even for a database system.

At the other end of the spectrum, the Liaison and Stubs translators have intentionally severe marshaling restrictions because their primary focus is not on marshaling. Serious client use of Liaison is impossible because of these restrictions.

6.5.3 Remote interfaces must be carefully designed.

When remote procedure call looks like local procedure call, most programmers reasonably expect the semantics and functionality to be the same. On the other hand, performance considerations dictate that programmers be aware of locality distinctions. Powerful marshaling tempts some programmers into poor performance by handling complex structures without complaint; this is an area of extreme danger for the hapless remote interface designer (consider the previous database example). The upshot is that a few marshaling restrictions are acceptable. Marshaling methods—and their client guidelines—are best derived only after more client experience and implementor negotiation.

6.5.4 Call-by-reference problems are tricky.

Limited client experience with these working Mesa RPC mechanisms has uncovered an unexpected amount of trouble with parameter allocation and deallocation. The trouble boils down to misuse of call-by-reference (pointer) parameters. While the following discussion of this problem may seem mundane to some, it is nonetheless important because it underlines the problem of retrofitting an existing language with a uniform RPC mechanism. This task will be easy for some languages and impossible for others. Mesa tends toward the former, but the lack of explicit VAR parameters, coupled with the frequent use of pointers to pass multiword parameters, causes headaches in remote interface design. By considering the problem and solution for this Mesa-specific example, light can be shed on overcoming similar idiosyncracies in other languages.

We introduce the problem by considering the current local and ideal remote implementations of the Mesa string routine

Append: PROCEDURE [*string, suffix*: STRING].

The Mesa system implementation of this procedure causes *string*↑ to be modified even though *string* itself is a call-by-value pointer. The (efficiency) advantage of this scheme is that the implementor does not have to allocate any new strings. He simply modifies the existing ones. However, this local *Append* fails miserably in the remote case—*string* will indeed be modified in the remote machine, but the side effects never get reflected back in the caller.

As we have seen, the most satisfactory solution to this problem is to add VAR parameters to Mesa. In this case our *Append* routine looks quite similar:

Append: PROCEDURE [*string*: VAR STRING, *suffix*: STRING].

This *Append* works properly whether it is called locally or remotely. Locally, VAR can be implemented as call-by-reference for good performance. Remotely, the actual call-by-value-result semantics of VAR cause the remotely modified *string* to be copied back to the caller, overwriting *string*↑. Notice that there is no new storage allocated in the caller.

Unfortunately, Mesa does not have VAR parameters, and we can illustrate the problems this causes by considering a likely interface declaration for a remote *Append* routine:

Append: PROCEDURE [*baseString*, *suffix*: STRING] RETURNS [*catenation*: STRING].

A typical programmer concerned with efficiency would probably implement this interface with the code

{ *MesaSystem.Append*[*baseString*, *suffix*]; RETURN[*baseString*] }.

This implementation will work both locally and remotely. Furthermore, the programmer would even be quite happy with himself: He created no new storage and even built upon an existing routine. His smile would fade, however, after a few minutes of debugging.

The problem he would encounter stems from the requirement of identical local and remote call semantics. The cheap implementation above does not satisfy this property because, in the local case, *baseString* and *catenation* refer to the same object whereas in the remote case *catenation* refers to a completely new object allocated in the caller by the RPC machinery. This is the fundamental problem with the cheap implementation. The thorns are felt when deallocation is considered:

> If clients of this *Append* routine uniformly program in the local fashion they will expect that *Append* performs no allocation. This assumption is fine in the server machine. In the client machine, however, each remote call of *Append* allocates space for *catenation* and will eventually exhaust the allocator since no *catenations* are ever freed.
>
> If clients uniformly program in a remote fashion they will assume that *Append* allocates storage for *catenation*. Now everything is fine on the client machine. In the server machine, the RPC machinery will now (programming in the remote fashion) attempt to deallocate *catenation*. *Catenation* was never allocated, however, and the *basestring* argument has presumably already been deallocated, so some sort of allocator exception will result.

This is a simple example, and the reader must be careful not to conclude that the solution is for the RPC machinery to be smart. Even if the RPC mechanism *was* smart, a client who used *Append*

for local calls would get the same exception. The only way an *Append* client can avoid a fault is to know whether a call is being performed locally (return-by-reference so do not deallocate) or remotely (return-by-value so do deallocate). This is, of course, exactly the situation we are trying to avoid. The *real* problem remains the fundamental semantic inconsistency discussed above, and RPC implementations cannot dance around it very easily.

Designers of remote interfaces must understand call-by-reference. An object's implementor must not perform side effects on objects that might live outside his virtual memory. Changes on such objects must simply return new objects whenever the VAR or HANDLE paradigms are not used.

Note that this lesson is unaffected by the presence of a garbage collector. A collector does, of course, remove explicit deallocation responsibilities. This is a truly significant advantage, and the clever reader may even convince himself that garbage collection enables the cheap *Append* implementation to work properly. This is indeed true to the extent that no allocator exceptions will occur, but it is false to the extent that identical local and remote semantics are guaranteed: *baseString* and *catenation* still have different values in the local and remote cases.

In summary, unless VAR parameters are available or the object-handle paradigm is used, designers of remote interfaces must be especially careful about allocation responsibilities and call-by-reference parameters. An unavoidable copying penalty will always be paid to operate on remote objects, and this penalty must be considered relative to the computational cost of the operation.

Application of this rule makes *Append*'s implementation more costly, but also gives it correct semantics at last:

```
Append: PROCEDURE [baseString, suffix: STRING] RETURNS [catenation: STRING] = {
    catenation ← MesaSystem.NewString[baseString.length + suffix.length];
    MesaSystem.Append[catenation, baseString];
    MesaSystem.Append[catenation, suffix] }.
```

Of course, any remote implementation of *Append* is probably an unreasonable operation because it is so inexpensive to perform locally.

6.5.5 Summary of Functionality Lessons

Here is a recapitulation of the empirical functionality lessons.

1. Named binding is important.
2. Clients want full parameter functionality.
3. Remote interfaces must be carefully designed.
4. Call-by-reference problems are tricky.

Lessons 1 and 2 verify the importance of the powerful binding and excellent parameter functionality properties. Lessons 3 and 4 verify the need for sound remote interface design. Lesson 4 also reinforces the importance of excellent parameter functionality.

6.6 Retrospective

This chapter focuses primarily and very narrowly on efficiency issues of remote procedure call. While attention to these lessons is crucial for the success of a viable RPC scheme, keeping the essential and pleasant properties in perspective is important. Good performance is a pleasant property, not an essential one. To meet the overall goal of transparency none of the essential properties must be compromised to achieve superior performance. Fortunately, chapter 5 presented a mechanism that satisfies all of the essential properties and has excellent performance too.

Considered individually, the lessons in this chapter give good but not outstanding results. Acting in concert, however, they result in a speedup of 35 times with no significant change in functionality. A *quantitative* performance improvement of this magnitude makes a *qualitative* difference in the value of remote procedure call as a language-level communication primitive. For instance, based on the background of chapter 3 and the Stubs performance numbers in this chapter, assume that an existing RPC mechanism has a 25-millisecond remote call overhead. If a distributed system designer is willing to pay 10% for communication costs, then the remote operations of these slow calls must take 250 milliseconds. This is a long time, longer than the time needed to use even a very slow disk. If a remote call takes 1 millisecond, on the other hand, then 10-millisecond operations are feasible. In this case, remote operations that involve only computation and no disk activity are attractive. Emissary's exceptionally large quantitative performance improvement makes a pronounced qualitative difference in the way that remote procedures can be used.

Aitutaki—Cooks
18°52' S 159°46' W
The Pacific's best dancers whirl on sparkling sands

7

Conclusion

Emissary's remote procedure mechanism is now complete. The basic approach, derived from the essential properties, was reinforced with orphan algorithms, promoted by a distributed binder, and confirmed with a performance evaluation.

7.1 Reviewing the Goals

The thesis of this dissertation is that remote procedure call is a satisfactory and efficient programming language primitive for constructing distributed systems. Three goals were established in the introduction to demonstrate this thesis. These goals have been met as follows.

> *Desirability.* Chapter 2 divided remote procedure applications into three classes—resource sharing, load splitting, and conversation. The utility of language-level RPC in each class was shown with specific examples. All of the examples could be programmed with message primitives as well as with remote procedures: there was no claim that RPC is a panacea for communication in distributed systems. Instead, RPC emerged as one natural way to write distributed programs in procedural languages. Chapter 3 continued the general discussion of desirability by examining the strengths—and weaknesses—of some existing RPC schemes.

> *Transparency—theory.* Chapter 4 explored the semantic and syntactic ramifications of transparency in great detail. For homogeneous language systems, there are five *essential properties* that must be satisfied by any RPC mechanism that is fully integrated into a programming language. These five properties are uniform call semantics, powerful binding and configuration, strong typechecking, excellent parameter functionality, and standard concurrency control and exception handling. In addition to the essential properties, there are six *pleasant properties* that ease the work of constructing real distributed systems. The pleasant properties are good performance, sound remote interface design, atomic transactions, respect for autonomy, type translation, and remote debugging.

> *Transparency—practice.* Chapter 5 used the semantic groundwork of chapter 4 to develop *Emissary*, a Mesa-based remote procedure mechanism that satisfied all five essential

properties. The Emissary design has three major components: orphan algorithms to handle crashes, call mechanisms to perform steady-state calls, and a distributed binder to link the modules of distributed programs. Although unimplemented, the heart of Emissary—the call machinery presented in algorithm 5.3—is based on the actual implementation and testing of a series of operational RPC mechanisms.

Efficiency. Chapter 6 contained a performance evaluation of five working RPC mechanisms, three of which I implemented: Envoy-Diplomat, Stubs, Liaison, EtherPkt, and EtherPktMC. The results of the evaluation were a set of general performance lessons that decreased the roundtrip time for a remote call by a remarkable factor of 35. These lessons were incorporated into the Emissary design of chapter 5; Emissary therefore satisfied the good performance property in addition to the essential properties.

The upshot of meeting the desirability, transparency, and efficiency goals is this: Remote procedures can and should be routinely used in applications where they have been previously regarded as an extravagant luxury. Transparent RPC is appropriate for constructing distributed systems because it is a good model for many distributed computations, is comfortable and familiar to programmers, and has excellent performance when properly implemented.

7.2 Critical Evaluation

7.2.1 The Value of Transparency

This thesis made a very early commitment to focus on transparent remote procedure schemes, that is, schemes with semantics neither weaker nor stronger than local procedure semantics. The reasons for this decision have been expressed in many ways, but perhaps none is as important as the following:

> Transparent RPC provides a level of abstraction at which the programmer can ignore all the details of both unreliable communication and crashes on multiple nodes. The programmer deals with a uniform system that has completely familiar properties, including last-one semantics during crashes.

While the qualitative truth of this statement is clear, the quantitative story can be substantially different. For instance, the elapsed time taken by an *inexpensive* remote operation in a distributed system can be an order of magnitude longer than its local counterpart. Furthermore, *crash* exceptions really do force the programmer to deal with crashes and last-one semantics in situations where the booting was once an acceptable resort. Transparency, then, is a mixed blessing: it lies in the middle of a semantic spectrum with at-least-once semantics at one end, and at-most-once semantics at the other.

To their credit, transparent mechanisms such as Emissary offer all the performance advantages of at-least-once schemes, but none of their semantic inconsistencies—as long as there are no crashes. This makes Emissary a wonderful agent for building experimental distributed systems: modules can be moved around, machines can be shuffled at will, and the whole business of constructing a prototype system with decent performance can be extremely simplified. Furthermore, applications that assume responsibility for their own data consistency can go even further: They can use the

last-once guarantee to program application-specific recovery operations just as they would for nondistributed systems.

To their detriment, however, transparent semantics offer little assistance to production programmers building a highly reliable system. Reasoning about crashes is very complex, and a level of abstraction higher than transparent RPC can be very helpful. For these highly reliable applications, at-most-once semantics—with built-in transactions yielding stronger than local procedure semantics—are undoubtedly what programmers want. In these cases, the performance of Emissary-like RPC schemes is willingly traded for the crash-proof atomicity of extremely robust RPC mechanisms. Of course, an intermediate approach to atomicity is using transparent remote procedures for communication between crashes and transaction mechanisms for reliability across crashes.

Emissary's position in the middle of the semantic spectrum has been rejected by some designers. Until more distributed applications are built, however, using a variety of RPC and transaction mechanisms, there will be insufficient experience to make firm conclusions about any preferred points in the spectrum.

7.2.2 The Need for Orphan Algorithms

Emissary's RPC mechanism, while transparent, must be used in conjunction with an independent transaction mechanism for highly reliable applications. This method has the attractive property that applications can define and use atomic operations—local or remote—in just the required situations. Unfortunately, in a distributed system the use of both transparent RPC and a multinode transaction mechanism usually leads to wasted motion in crash recovery: In guaranteeing atomicity, distributed transaction schemes effectively eliminate all of their own orphans as a part of transaction-specific crash recovery. In this case, using separate transactions to guarantee atomicity makes the work of Emissary's orphan algorithms completely redundant. Notice, however, that if both regular and atomic remote operations are used, then there is no redundancy for the regular operations because they fall outside the sphere of the transaction scheme.

This redundancy raises a question about the need for orphan algorithms. In the previous semantic spectrum, if real applications tend to lie at both ends, then orphan algorithms are not needed at all: when programmers use RPC for transparent between-crash semantics, orphan algorithms are an unwanted expense; when programmers use transactions for *all* remote operations, orphan algorithms are unnecessary.

All distributed applications do not fit precisely into these two extremes. Distributed programs with modest reliability requirements will probably fit between the two, and for these applications orphan algorithms and their last-one semantics are vital. Once again, more experience is required before conclusive judgments can be made.

7.2.3 *The Role of Performance*

In chapters 5 and 6, performance issues played a critical role in Emissary's design. The decision to give performance considerations equal parity with abstract functionality is an important one. The wisdom of this decision—or the folly of it—lays in the following remark by Bob Sproull [83]:

> The structuralists are currently in vogue, and the performers are out of power now. But middle-of-the-road types, using the techniques of both, are the better designers.

At their polarized extremes, structuralists are designers who insist on general purpose, highly layered design and implementation at *all* levels of a system; performers are designers who always sacrifice as much functionality and structure as necessary to optimize an implementation to the fullest extent. Sproull's thesis is, of course, that the best designers strike a balance between clean structure and good performance. This thesis is quite evident in the following two areas of Emissary's design.

> *Syntactic transparency.* At the language level, absolute syntactic (structural) transparency is blemished by the REMOTE attribute. But in yielding this transparency in declarations—and only in declarations—Emissary's compiler is able to perform substantial space and time optimizations for remote calls.
>
> *Semantic transparency.* At the runtime level, Emissary's remote call mechanism is optimized only for *procedure* calls. At the language level, however, the complete range of control transfers is available to the programmer. Emissary invisibly maps the uncommon transfers into procedures, giving full generality with a spectrum of performance.

The demands of structure and performance are carefully balanced in Emissary's design to result in a practical and usable mechanism.

7.2.4 *The Trials of Implementation*

Two critical components of Emissary's design are not implemented: the orphan algorithms and distributed binder presented in chapter 5. If the implementation and evaluation of either of these pieces finds serious flaws, the integrity of the Emissary's approach will suffer. Thus, while the orphan and binding approaches are developed in some detail, the trials of their implementation will deliver the final verdict. Emissary's call mechanism is excluded from this discussion because its detailed but unimplemented algorithm is based on several operational prototypes. These closely related call mechanisms were successful.

7.2.5 *The Nature of Processes*

The Emissary design depends on Hoare-like monitors for interprocess communication. This style of process interaction uses shared memory, although the sharing is captured in monitor data structures and is completely hidden from clients behind the monitor's abstract operations. The monitor model is well represented in languages like Ada and Mesa, which use inexpensive shared-memory tasks and processes (*lightweight* processes). In systems with processes that have separate address spaces—for example, Unix and Multics—the remote procedure model is less natural unless these robust processes (*heavyweight* processes) have a layer of lightweight process structure within

them. In this latter case, the heavyweight processes have the characteristics of logically distinct nodes, with RPC supplying the communication between them. This characterization is also confirmed by experience, for in the absence of lightweight processes, heavyweight processes usually communicate with message passing—just as the physically distinct nodes in an internetwork do. Some operating systems even offer both kinds of processes. For example, a Thoth *team* or a Medusa *task force* is a heavyweight process in which lightweight shared-memory processes are supported (Medusa calls these lightweight processes *activities*).

Concluding that systems with heavyweight processes are unsuitable for RPC is incorrect. A vital ingredient of Emissary's design is its *language-level* approach, and, in state-of-the-art procedural languages, monitors and lightweight processes are language-level abstractions also. The benefit of having all these features in a language is that the language's runtime environment is easily encapsulated in a heavyweight process, as described above. Thus machine architectures and operating systems that provide only heavyweight processes can easily support programming environments that use remote procedures. The only drawback is that performance will suffer if the heavyweight processes have cumbersome interprocess communication. However, this same performance argument is true whether communication is between virtual nodes in one processor, physical nodes in an internetwork, or some combination of the two.

7.3 Future Directions for RPC

In the short term, RPC mechanisms need detailed design and implementation. Since Emissary's detailed design is complete (algorithm 5.3), the first step is to implement Emissary's (or an Emissary-like) call mechanism. With this language-level communication backbone in place, detailed design and construction of a distributed binder can follow, as can implementation of the orphan algorithms. Both the Mesa and Ada languages are well suited for transparent language-level RPC adaptation.

A full scale implementation of remote procedures in a homogeneous language environment allows the RPC framework of the essential properties to be evaluated in an ideal context. This evaluation is vital for two reasons: First, to test the ideas of this thesis in a more realistic setting than has been attempted heretofore. Second, to discover what other directions language-level distributed communication primitives should take. I believe speculation about the latter point is inappropriate until the former has been resolved.

In the medium term, RPC mechanisms must acquire the pleasant properties. Heterogeneous language and processor systems, rich distributed programming environments, and extremely robust applications all require communication primitives that satisfy the pleasant properties. Fortunately, independent work addressing some of these properties is either finished or in progress—for example, Herlihy's type translation work for Clu, and Liskov's atomic transaction mechanism for Guardians. Of the three remaining pleasant properties, sound remote interface design and remote debugging present primarily engineering challenges, and questions of node autonomy are best answered on a system-by-system basis.

In the long term, RPC mechanisms need to offer increased reliability and fault tolerance. The use of transaction mechanisms to maintain data consistency in a distributed system is fairly well understood. However, problems of network partitioning, location transparency over crashes, and automatic system reconfiguration are less clear. Whether or not solving these reliability problems will cause fundamental alterations in RPC mechanisms—and it seems certain that binding approaches will change—RPC will be valuable in constructing systems that test any proposed solutions.

Finally, RPC mechanisms must accommodate one trend of distributed systems themselves—growing smaller. The physical example used for internode communication in most of this dissertation is the Ethernet. With VLSI technology, however, tens or even hundreds of physically distributed processors will soon be packed onto a single circuit board, or perhaps even onto an integrated circuit. This change in scale will undoubtedly change interprocessor transport mechanisms, but, as long as individual processors do not share memory—or are strongly discouraged from doing so—RPC will continue to be a viable language-level communication strategy. This will remain true for both on- and off-chip communication whenever high-level procedural languages are a programming medium.

7.4 Contribution to Computer Science

This dissertation is a step toward the time when distributed computing is commonplace. Its contribution is a genesis of two distinct lines of evolution: communication in distributed systems and high-level programming languages.

The past fifteen years have seen networks and other communication media of distributed systems move from myth to madness to moderation. Moderation came with protocol hierarchies that now permit programmers to completely ignore underlying transport mechanisms and their myriad characteristics. These protocol layers provide a wonderfully flexible and unassuming communication medium. But there's the rub—no assumed structure.

These same fifteen years—and the twenty before that—watched programming language evolution move through these same stages: myth, madness, and moderation. Programming languages attained moderation by layering data and control structures atop a myriad of machine architectures. These structuring schemes are now very formal and abstract: the type, data, and control notions of modern programming languages are far removed from the characteristic concepts of most intercomputer communication.

My work in remote procedure call joins these evolving lines. The dissertation describes the additional layering—and, for efficiency, transparent delayering—needed to extend the disciplined semantics of abstractly programmed machines into a universe populated with such machines. Remote procedure call is one transparent *ether* for distributed program communication.

Islas de Juan Fernández—Chile
33°30′ S 78°55′ W
Shipwreck legends help Defoe spin the yarn of Robinson Crusoe, *1719*

Appendix 1

Some Mesa Details

This appendix contains some details about the Mesa language [62]. Its purpose is to help readers familiar with Pascal and similar procedural languages understand some unusual Mesa-specific constructs used throughout the thesis, especially in chapter 5. The descriptions given here are not comprehensive; only the necessary details are given.

Identifiers

Case is significant in Mesa identifiers. For example, in the two declarations *squareBox*: *SquareBox* and *squarebox*: *SquareBox*, *squareBox* and *squarebox* are distinct variables of the same *SquareBox* type.

The CARDINAL Type

A CARDINAL is an unsigned number. An n-bit cardinal can represent values in the interval $[0..2^n)$.

Statement Brackets

In addition to BEGIN and END, Mesa allows { and } to enclose a series of statements. Do not confuse this with Pascal's use of { and } to delimit comments (Mesa uses -- to begin a comment).

Block Exits

Blocks can be exited with GOTO statements. The optional part of a block where the exit clauses are declared is introduced with EXITS. For example:

```
BEGIN
 statements;
 ... GOTO Label;
 statements;
 lastStatement;
EXITS -- This line and the next are optional.
 Label => statement;
END;
nextStatement.
```

In the absence of GOTOs, a block exits by jumping from *lastStatement* directly to *nextStatement*.

Loop Statements

Loops have a number of control and termination options. They are illustrated in this example:

```
FOR variable IN interval UNTIL stopCondition DO -- WHILE goCondition is possible too.
        loopStatements;
        ... LOOP; -- This causes control to return immediately to the FOR.
        loopStatements;
        ... EXIT; -- This causes control to resume at nextStatement, below.
        loopStatements;
   REPEAT -- This line and the next are optional.
        FINISHED => normalTerminationStatement; -- See text.
   ENDLOOP;
nextStatement.
```

The FOR and UNTIL (or WHILE) clauses are optional and can have different forms from the ones shown here. The only unobvious one is THROUGH, which is exactly like FOR except that it has no control *variable*. The special LOOP statement starts the next iteration without finishing the current one. The EXIT statement terminates the loop immediately, transferring control to *nextStatement*. The optional REPEAT keyword ends the iterative part of the loop and introduces a series of exit clauses, similar to block EXITS. The FINISHED label is a special exit; its *normalTerminationStatement* is executed if and only if the loop terminates normally. Normal termination occurs when either the FOR steps completely through *interval*, or when *stopCondition* is TRUE (or *goCondition* is FALSE).

Case Statements

Mesa's case statements are known as SELECT and have a fairly traditional syntax. For example:

```
SELECT expression FROM
        0,3,5 => statement; -- 0,3,5 is shorthand for =0,=3,=5.
        IN [27..bound] => statement;
        <-10 => statement;
        ENDCASE => finalStatement.
```

The first (and only the first) arm that matches *expression* is executed. The optional *finalStatement* is executed if and only if none of the arms match *expression*.

Default Values for Parameters

In Mesa, the formal parameters of a procedure—both arguments *and* results—are local variables of the procedure. These arguments and results can be assigned default values. For example:

Lookup: PROCEDURE [*name*: STRING, *exactCaseMatch*: BOOLEAN←FALSE] RETURNS [*match*: STRING←""].

When *Lookup* is called, *exactCaseMatch* is FALSE unless the programmer supplies an explicit value (e.g., *Lookup*["BZM"] is the same as *Lookup*["BZM", FALSE]). The return variable *match* has the null string as its initial value.

MONITORS and CONDITION Variables

Monitors are a powerful language-level synchronization tool [47]. The following discussion is *not* an introduction to monitors and gives only an overview of Mesa's mechanism, which is similar to the one proposed by Hoare [37].

A MONITOR provides synchronized, mutually excluded operations on the monitor's shared global data by associating a *lock* (semaphore) with the shared data. Monitors have three kinds of procedures:

> Normal PROCEDURES are unsynchronized, do not acquire the monitor's lock, and can overlap each other in time.
>
> ENTRY PROCEDURES are synchronized, must acquire the monitor's lock to run, and therefore execute serially in time. (Concurrent calls to entry procedures are automatically queued until the lock is released.)
>
> INTERNAL PROCEDURES are synchronized, implicitly have the monitor's lock because they can be called *only* from entry or internal procedures, and hence execute serially.

Fine grain monitor synchronization is provided by CONDITION variables and the WAIT and NOTIFY operations. The basic sequence of actions is this: When an entry (or internal) procedure executing in process p WAITs on condition c in monitor M, p is blocked and M's lock is released. Later, an independent process q calls some entry procedure in M that NOTIFYs c. This NOTIFY unblocks process p and schedules p to run, possibly as soon as process q releases M's lock (by either exiting the entry procedure or WAITing). This is illustrated in the skeleton monitor M below, where procedure P is called by process p, and procedure Q by process q.

```
M: MONITOR = {
    synchronizedData: Type = ...;
    c: CONDITION;
  P: ENTRY PROCEDURE [...] = {... WAIT c; ...};
  Q: ENTRY PROCEDURE [...] = {... NOTIFY c; ...};
  ... }.
```

The NOTIFY operation is not queued, so that if a NOTIFY precedes a WAIT, the WAITing process will block until a subsequent NOTIFY.

SIGNALs and Exception Handling

Exceptions are a powerful error-handling tool [54]. Once again, the following discussion is *not* an introduction to exceptions and gives only an overview of Mesa's mechanism.

An exception *E* is declared like a procedure without a body, e.g., *E*: SIGNAL =.... (Exceptions can have arguments and results, but parameters are not considered here.) An exception is signalled (raised) in the current process by writing SIGNAL *E*; it is raised in process *p* by writing SIGNAL *E* IN *p*. When an exception such as *E* is signalled, a search back up the call stack in the target process looks for a procedure call or a block that is *enabled* to *catch E*. For each procedure or block that is enabled, the corresponding catch statement is executed. Catches are written as follows:

```
P[arguments ! E => catchStatement]; -- Catch on a procedure call.
nextStatement.

BEGIN ENABLE E => catchStatement; -- Catch on a block, can also be on a loop's DO.
 statements;
END;
nextStatement.
```

A *catchStatement* is an arbitrary statement, including a block. There are several special actions that can be used in a catch to control how the exception is handled:

> RESUME causes *E* to be ignored at the point it was raised. Execution continues immediately after the SIGNAL *E*.
>
> RETRY causes the procedure or block to be reexecuted. In the example above, *P* is recalled, or the block is restarted at BEGIN.
>
> CONTINUE continues execution at the statement after the catch. In the example, execution continues at the respective *nextStatements*.

The following program is a poor but illustrative example of simple exception handling. It attempts to deliver a telephone message until the line is not busy.

```
Busy: SIGNAL = ...;
Dial: PROCEDURE [number: PhoneNumber] =
        {...; IF noAnswer THEN SIGNAL Busy; ...};
DeliverPhoneMessage: PROCEDURE [number: PhoneNumber, message: Message] =
        {...; Dial[number]; SendAudioMessage[message]; ...};
-- Main program:
DeliverPhoneMessage[NASA, ShuttleCongratulations ! Busy => RETRY].
```

Upolu—Samoa
13°55' S 171°40' W
Natives bury Robert Louis Stevenson on Mount Vaea, 3 December 1894

Appendix 2

Examples of Envoy-Diplomat, Liaison, and EtherPkt

The family of remote procedure mechanisms evaluated in chapter 6 has five generations. Abridged stub programs for three generations—Envoy-Diplomat, Liaison, and EtherPkt—appear here. Examples of the remaining two generations are omitted because they resemble in character, if not in performance, one of the included schemes: Stubs resembles Liaison, and EtherPktMC resembles EtherPkt. Brief overviews of all these mechanisms appear in section 6.1.3.

The remote *ParamTest* interface implemented by each of these schemes is shown below. *ParamTest* is shortened for clarity in this example by editing the full performance-testing version.

The three stub implementations have an Emissary-like stub structure: a client-resident stub module exports *ParamTest* and transparently transmits calls to a server stub module; the server-resident stub actually calls *ParamTest*'s implementation. A more comprehensive description accompanies each example.

```
-- ParamTest.mesa last edited by BZM on October 21, 1980 9:14 AM.
-- This is the primary test program. The implementation of each procedure
--   should echo its arguments back as results.

ParamTest: DEFINITIONS = BEGIN

Array20: TYPE = ARRAY [0..20) OF CARDINAL;

Null: PROC;
One: PROC [one: CARDINAL] RETURNS [a: CARDINAL];
Four: PROC [one,two,three,four: CARDINAL] RETURNS [a,b,c,d: CARDINAL];
TwentyArray: PROC [in: Array20] RETURNS [out: Array20];
StringDescriptor: PROC [string: STRING] RETURNS [desc: DESCRIPTOR FOR ARRAY OF CHARACTER];
-- Procedures Two, TwoArray, FourArray, TenArray, and FortyArray have been deleted.

END.
```

A2.1 Envoy-Diplomat

This example contains Envoy-Diplomat's four generated stub programs for the simple *ParamTest* interface: *ParamTestEnvoy*, *ParamTestEnvoyClient*, *ParamTestEnvoyServer*, and *ParamTestEnvoyUtility*.

To aid the reader's understanding of this code, here is the intermodule control flow for a single steady-state call of *ParamTest.Null*.

> On the client machine, *ParamTestDriver*, which performs the timing measurements, calls *ParamTest.Null*.
>
> *ParamTestEnvoyClient*, the client module exporting *Null*, catches the call and feeds it into *Call*, which calls *Envoy.CallRemoteProcedure*.
>
> In the client machine, Envoy marshals the arguments (if any—*ParamTest* has none) using the type descriptions in *ParamTestEnvoyUtility* and sends a *call* message to the server machine.
>
> On the server machine, Envoy receives the *call* message and invokes the *Dispatch* routine in *ParamTestEnvoyServer*.
>
> *Dispatch* unmarshals the arguments and invokes *ParamTest.Null*.
>
> On the server machine, *ParamTestImpl*, which exports the real implementation of *Null*, performs the work of the call and returns to *Dispatch* in *ParamTestEnvoyServer*.
>
> *Dispatch* marshals the results and returns to Envoy.
>
> In the server machine, Envoy handles the completed call by transmitting a *return* message back to the client machine.
>
> On the client machine, Envoy receives the *return* message, unmarshals the results, and returns from *CallRemoteProcedure*.
>
> *ParamTestEnvoyClient* returns from *Call*, and then from its implementation of *Null*.
>
> *ParamTestDriver*, which originally called *Null*, resumes at long last and completes its timing of the call.

```
-- File ParamTestEnvoy.mesa was generated on 23-Oct-80 16:08:05 by Diplomat of 13-Oct-80 16:18:37.
-- Source interface ParamTest came from file paramtest.bcd, created on 23-Oct-80 14:50:38 (3# 145 #)
        from source of 21-Oct-80 9:17:05.

DIRECTORY Envoy, ParamTest;

ParamTestEnvoy: DEFINITIONS
 SHARES ParamTest
 = PRIVATE BEGIN

-- Runtime Error Exceptions
DiplomatRuntimeError: PUBLIC SIGNAL;

-- Remote Binding Interface

RemoteInterfaceVersion: PUBLIC Envoy.Version = 22607660056B; --Compilation time of source interface.

InitAndExportRemoteInterface: PUBLIC PROCEDURE [
    interfaceVersion: Envoy.Version ← RemoteInterfaceVersion,
    interfaceImplementor: Envoy.Implementor ← 0  ];

ImportAndBindRemoteInterface: PUBLIC PROCEDURE [
    exportingHost: Envoy.SystemElement,
    interfaceVersion: Envoy.Version ← RemoteInterfaceVersion,
    interfaceImplementor: Envoy.Implementor ← 0,
    callTimeoutInSeconds: LONG CARDINAL ← LAST[LONG CARDINAL]  ];

-- Remote Procedure and Error Definitions

RemoteProcedures: TYPE = ARRAY RemoteProcedureIndex OF Envoy.RemoteProcedure;
RemoteErrors: TYPE = ARRAY RemoteErrorIndex OF Envoy.RemoteError;

RemoteProcedureIndex: TYPE = {Filler, Null, One, Two, Four, TwentyArray, FortyArray, StringDescriptor};
RemoteErrorIndex: TYPE = {Filler};

-- Parameter Description Definitions

GetRemoteProcedureDescriptions: PROCEDURE RETURNS [procedures: RemoteProcedureDescriptionsHandle];
GetRemoteErrorDescriptions: PROCEDURE RETURNS [errors: RemoteErrorDescriptionsHandle];

RemoteProcedureDescriptionsHandle: TYPE = POINTER TO READONLY RemoteProcedureDescriptions;
RemoteErrorDescriptionsHandle: TYPE = POINTER TO READONLY RemoteErrorDescriptions;
RemoteProcedureDescriptions: TYPE = ARRAY RemoteProcedureIndex OF RECORD [arguments, results:
Envoy.Description];
RemoteErrorDescriptions: TYPE = ARRAY RemoteErrorIndex OF RECORD [arguments: Envoy.Description];

-- Parameter Record Definitions

MaxArgumentRecordSize: CARDINAL = MAX[1, SIZE[NullArguments], SIZE[OneArguments], SIZE[TwoArguments],
SIZE[FourArguments], SIZE[TwoArrayArguments], SIZE[FourArrayArguments], SIZE[TenArrayArguments],
SIZE[TwentyArrayArguments], SIZE[FortyArrayArguments], SIZE[StringDescriptorArguments]];

MaxResultRecordSize: CARDINAL = MAX[1, SIZE[NullResults], SIZE[OneResults], SIZE[TwoResults],
SIZE[FourResults], SIZE[TwoArrayResults], SIZE[FourArrayResults], SIZE[TenArrayResults],
SIZE[TwentyArrayResults], SIZE[FortyArrayResults], SIZE[StringDescriptorResults]];

GenericArgumentRecord: TYPE = ARRAY [0..MaxArgumentRecordSize) OF CARDINAL;
GenericResultRecord: TYPE = ARRAY [0..MaxResultRecordSize) OF CARDINAL;

NullArguments: TYPE = RECORD [];
NullResults: TYPE = RECORD [];
OneArguments: TYPE = RECORD [one: CARDINAL];
OneResults: TYPE = RECORD [a: CARDINAL];
FourArguments: TYPE = RECORD [one: CARDINAL, two: CARDINAL, three: CARDINAL, four: CARDINAL];
FourResults: TYPE = RECORD [a: CARDINAL, b: CARDINAL, c: CARDINAL, d: CARDINAL];
TwentyArrayArguments: TYPE = RECORD [in: ParamTest.Array20];
TwentyArrayResults: TYPE = RECORD [out: ParamTest.Array20];
StringDescriptorArguments: TYPE = RECORD [string: STRING];
StringDescriptorResults: TYPE = RECORD [desc: DESCRIPTOR FOR ARRAY CARDINAL OF CHARACTER];

END.
```

```
-- File ParamTestEnvoyClient.mesa was generated on 23-Oct-80 16:08:18 by Diplomat of 13-Oct-80 16:18:37.
-- Source interface ParamTest came from file paramtest.bcd, created on 23-Oct-80 14:50:38 (3 # 145 #)
        from source of 21-Oct-80 9:17:05.

DIRECTORY Envoy, ParamTestEnvoy, ParamTest;


ParamTestEnvoyClient: PROGRAM
 IMPORTS ParamTest, Envoy, ParamTestEnvoy
 EXPORTS ParamTest, ParamTestEnvoy
 SHARES ParamTest, ParamTestEnvoy
 = PRIVATE BEGIN OPEN ParamTestEnvoy;


-- Remote Binding

alreadyBound: BOOLEAN ← FALSE;
remoteProcedures: RemoteProcedures;
ourServerProgram: Envoy.RemoteProgram;

InitAndExportRemoteInterface: PUBLIC PROCEDURE [
    interfaceVersion: Envoy.Version ← RemoteInterfaceVersion,
    interfaceImplementor: Envoy.Implementor ← 0 ] = {SIGNAL DiplomatRuntimeError};

ImportAndBindRemoteInterface: PUBLIC PROCEDURE [
    exportingHost: Envoy.SystemElement,
    interfaceVersion: Envoy.Version ← RemoteInterfaceVersion,
    interfaceImplementor: Envoy.Implementor ← 0,
    callTimeoutInSeconds: LONG CARDINAL ← LAST[LONG CARDINAL] ] =
 BEGIN
 remoteProgram: ARRAY [0..0] OF Envoy.RemoteProgram;
 remoteProgramID: ARRAY [0..0] OF Envoy.ProgramID;
 IF alreadyBound THEN RETURN ELSE alreadyBound ← TRUE;
 remoteProgramID ← [ [interface: "ParamTest", version: interfaceVersion, implementor: interfaceImplementor] ];
 Envoy.ImportRemotePrograms [
    systemElement: exportingHost,
    programIDs: DESCRIPTOR[remoteProgramID],
    remotePrograms: DESCRIPTOR[remoteProgram] ];
 ourServerProgram ← remoteProgram[0];
 callTimeout ← callTimeoutInSeconds;
 FOR procedure: RemoteProcedureIndex IN RemoteProcedureIndex DO
       remoteProcedures[procedure] ← Envoy.ComposeRemoteProcedure [
     remoteProgram: ourServerProgram,
     procedureNumber: LOOPHOLE[procedure, Envoy.ProcedureNumber] ];
       ENDLOOP;
 END;


-- Remote Interface Errors


-- Remote Interface Procedures

Null: PUBLIC PROCEDURE [] RETURNS [] =
 BEGIN
 arguments: NullArguments;
 results: NullResults;
 Call[RemoteProcedureIndex[Null], @arguments, @results];
 RETURN[];
 END;

One: PUBLIC PROCEDURE [one: CARDINAL] RETURNS [a: CARDINAL] =
 BEGIN
 arguments: OneArguments ← [one];
 results: OneResults;
 Call[RemoteProcedureIndex[One], @arguments, @results];
 RETURN[results.a];
 END;
```

```
Four: PUBLIC PROCEDURE [one: CARDINAL, two: CARDINAL, three: CARDINAL, four: CARDINAL] RETURNS [a:
CARDINAL, b: CARDINAL, c: CARDINAL, d: CARDINAL] =
 BEGIN
 arguments: FourArguments ← [one, two, three, four];
 results: FourResults;
 Call[RemoteProcedureIndex[Four], @arguments, @results];
 RETURN[results.a, results.b, results.c, results.d];
 END;

TwentyArray: PUBLIC PROCEDURE [in: ParamTest.Array20] RETURNS [out: ParamTest.Array20] =
 BEGIN
 arguments: TwentyArrayArguments ← [in];
 results: TwentyArrayResults;
 Call[RemoteProcedureIndex[TwentyArray], @arguments, @results];
 RETURN[results.out];
 END;

StringDescriptor: PUBLIC PROCEDURE [string: STRING] RETURNS [desc: DESCRIPTOR FOR ARRAY CARDINAL
OF CHARACTER] =
 BEGIN
 arguments: StringDescriptorArguments ← [string];
 results: StringDescriptorResults;
 Call[RemoteProcedureIndex[StringDescriptor], @arguments, @results];
 RETURN[results.desc];
 END;


-- Remote Call Handler

procedureDescriptions: RemoteProcedureDescriptionsHandle = GetRemoteProcedureDescriptions[];
errorDescriptions: RemoteErrorDescriptionsHandle = GetRemoteErrorDescriptions[];

callTimeout: LONG CARDINAL ← LAST[LONG CARDINAL];

Call: PROCEDURE [procedure: RemoteProcedureIndex, argumentList, resultList: POINTER ] =
 BEGIN
        ENABLE BEGIN
        -- Remote exception handling goes here.
        END; -- ENABLE

 IF ~alreadyBound THEN ImportAndBindRemoteInterface[Envoy.LocalSystemElement[]];
 Envoy.CallRemoteProcedure [
        remoteProcedure: remoteProcedures[procedure],
        arguments: [location: argumentList, description: procedureDescriptions↑[procedure].arguments],
        results: [location: resultList, description: procedureDescriptions↑[procedure].results],
        timeoutInSeconds: callTimeout   ];

 END; -- Call


-- Module Initialization

-- See ImportAndBindRemoteInterface.

END.
```

```
-- File ParamTestEnvoyServer.mesa was generated on 23-Oct-80 16:08:15 by Diplomat of 13-Oct-80 16:18:37.
-- Source interface ParamTest came from file paramtest.bcd, created on 23-Oct-80 14:50:38 (3#145#)
        from source of 21-Oct-80 9:17:05.


DIRECTORY Envoy, ParamTestEnvoy, ParamTest;


ParamTestEnvoyServer: PROGRAM
 IMPORTS ParamTest, Envoy, ParamTestEnvoy
 SHARES ParamTest, ParamTestEnvoy
 = PRIVATE BEGIN OPEN ParamTestEnvoy;


-- Remote Binding

alreadyExported: BOOLEAN ← FALSE;
remoteErrors: RemoteErrors;

ImportAndBindRemoteInterface: PUBLIC PROCEDURE [
      exportingHost: Envoy.SystemElement,
      interfaceVersion: Envoy.Version ← RemoteInterfaceVersion,
      interfaceImplementor: Envoy.Implementor ← 0,
      callTimeoutInSeconds: LONG CARDINAL ← LAST[LONG CARDINAL]   ] = {SIGNAL DiplomatRuntimeError};

InitAndExportRemoteInterface: PUBLIC PROCEDURE [
      interfaceVersion: Envoy.Version ← RemoteInterfaceVersion,
      interfaceImplementor: Envoy.Implementor ← 0   ] =
 BEGIN
 remoteProgram: Envoy.RemoteProgram;
 IF alreadyExported THEN RETURN ELSE alreadyExported ← TRUE;
 remoteProgram ← Envoy.FabricateRemoteProgram[localProgram: Dispatch];
 Envoy.ExportRemoteProgram [
      programID: [interface: "ParamTest", version: interfaceVersion, implementor: interfaceImplementor],
      localProgram: Dispatch   ];
 FOR error: RemoteErrorIndex IN RemoteErrorIndex DO
        remoteErrors[error] ← Envoy.ComposeRemoteError [
      remoteProgram: remoteProgram,
      errorNumber: LOOPHOLE[error, Envoy.ErrorNumber]   ];
        ENDLOOP;
 END;


-- Dispatching to Local Procedures

procedureDescriptions: RemoteProcedureDescriptionsHandle = GetRemoteProcedureDescriptions[];
errorDescriptions: RemoteErrorDescriptionsHandle = GetRemoteErrorDescriptions[];

Dispatch: Envoy.LocalProgram --[procedureNumber: ProcedureNumber, arguments, results: PROC[Parameters]]-- =
 BEGIN
        ENABLE BEGIN
        -- Exception handling goes here.
        END; --ENABLE

 procedure: RemoteProcedureIndex = LOOPHOLE[procedureNumber];
 argumentList: POINTER TO GenericArgumentRecord = @argumentListRecord;
 argumentListRecord: GenericArgumentRecord;
 resultList: POINTER TO GenericResultRecord = @resultListRecord;
 resultListRecord: GenericResultRecord;

 arguments[[location: argumentList, description: procedureDescriptions↑[procedure].arguments]];
 SELECT procedure FROM
        Null => {
                OPEN arg: LOOPHOLE[argumentList, POINTER TO NullArguments],
                        res: LOOPHOLE[resultList, POINTER TO NullResults];
                ParamTest.Null[];  };
        One => {
                OPEN arg: LOOPHOLE[argumentList, POINTER TO OneArguments],
                        res: LOOPHOLE[resultList, POINTER TO OneResults];
                [res.a] ← ParamTest.One[arg.one];  };
```

```
        Four => {
                OPEN arg: LOOPHOLE[argumentList, POINTER TO FourArguments],
                        res: LOOPHOLE[resultList, POINTER TO FourResults];
                [res.a, res.b, res.c, res.d] ← ParamTest.Four[arg.one, arg.two, arg.three, arg.four];   };
        TwentyArray => {
                OPEN arg: LOOPHOLE[argumentList, POINTER TO TwentyArrayArguments],
                        res: LOOPHOLE[resultList, POINTER TO TwentyArrayResults];
                [res.out] ← ParamTest.TwentyArray[arg.in];   };
        StringDescriptor => {
                OPEN arg: LOOPHOLE[argumentList, POINTER TO StringDescriptorArguments],
                        res: LOOPHOLE[resultList, POINTER TO StringDescriptorResults];
                [res.desc] ← ParamTest.StringDescriptor[arg.string];   };
        ENDCASE => ERROR DiplomatRuntimeError;
 results[[location: resultList, description: procedureDescriptions↑[procedure].results]];

 END; --Dispatch

-- Module Initialization

-- See InitAndExportRemoteInterface.

InitAndExportRemoteInterface[
      interfaceVersion: ,
      interfaceImplementor: ,
      possibleParameterOverlap:    ];

END.
```

```
-- File ParamTestEnvoyUtility.mesa was generated on 23-Oct-80 16:08:09 by Diplomat of 13-Oct-80 16:18:37.
-- Source interface ParamTest came from file paramtest.bcd, created on 23-Oct-80 14:50:38 (3#145#)
        from source of 21-Oct-80 9:17:05.


DIRECTORY Envoy, ParamTestEnvoy, ParamTest;


ParamTestEnvoyUtility: PROGRAM
 EXPORTS ParamTestEnvoy
 SHARES ParamTest, ParamTestEnvoy
 = PRIVATE BEGIN OPEN ParamTestEnvoy;


-- Runtime Error Exceptions

DiplomatRuntimeError: PUBLIC SIGNAL = CODE;


-- Interface To Descriptions

GetRemoteProcedureDescriptions: PUBLIC PROCEDURE RETURNS[procedures:
RemoteProcedureDescriptionsHandle] =
                { RETURN [@procedureDescriptions] };
GetRemoteErrorDescriptions: PUBLIC PROCEDURE RETURNS [errors: RemoteErrorDescriptionsHandle] =
                { RETURN [@errorDescriptions] };

procedureDescriptions: RemoteProcedureDescriptions ← [
        [NULL, NULL],
        [NullArgumentsDescription, NullResultsDescription],
        [OneArgumentsDescription, OneResultsDescription],
        [TwoArgumentsDescription, TwoResultsDescription],
        [FourArgumentsDescription, FourResultsDescription],
        [TwoArrayArgumentsDescription, TwoArrayResultsDescription],
        [FourArrayArgumentsDescription, FourArrayResultsDescription],
        [TenArrayArgumentsDescription, TenArrayResultsDescription],
        [TwentyArrayArgumentsDescription, TwentyArrayResultsDescription],
        [FortyArrayArgumentsDescription, FortyArrayResultsDescription],
        [StringDescriptorArgumentsDescription, StringDescriptorResultsDescription]   ];

errorDescriptions: RemoteErrorDescriptions ← [
        [NULL]   ];


-- Description Procedures

NullArgumentsDescription: Envoy.Description --[notes: Notes]-- =
 BEGIN OPEN notes;
 parameters: LONG POINTER TO NullArguments = noteSize[size: SIZE[NullArguments]];
 END;

NullResultsDescription: Envoy.Description --[notes: Notes]-- =
 BEGIN OPEN notes;
 parameters: LONG POINTER TO NullResults = noteSize[size: SIZE[NullResults]];
 END;

OneArgumentsDescription: Envoy.Description --[notes: Notes]-- =
 BEGIN OPEN notes;
 parameters: LONG POINTER TO OneArguments = noteSize[size: SIZE[OneArguments]];
 END;

OneResultsDescription: Envoy.Description --[notes: Notes]-- =
 BEGIN OPEN notes;
 parameters: LONG POINTER TO OneResults = noteSize[size: SIZE[OneResults]];
 END;

FourArgumentsDescription: Envoy.Description --[notes: Notes]-- =
 BEGIN OPEN notes;
 parameters: LONG POINTER TO FourArguments = noteSize[size: SIZE[FourArguments]];
 END;
```

```
FourResultsDescription: Envoy.Description --[notes: Notes]-- =
 BEGIN OPEN notes;
 parameters: LONG POINTER TO FourResults = noteSize[size: SIZE[FourResults]];
 END;

TwentyArrayArgumentsDescription: Envoy.Description --[notes: Notes]-- =
 BEGIN OPEN notes;
 parameters: LONG POINTER TO TwentyArrayArguments = noteSize[size: SIZE[TwentyArrayArguments]];
 END;

TwentyArrayResultsDescription: Envoy.Description --[notes: Notes]-- =
 BEGIN OPEN notes;
 parameters: LONG POINTER TO TwentyArrayResults = noteSize[size: SIZE[TwentyArrayResults]];
 END;

StringDescriptorArgumentsDescription: Envoy.Description --[notes: Notes]-- =
 BEGIN OPEN notes;
 parameters: LONG POINTER TO StringDescriptorArguments = noteSize[size: SIZE[StringDescriptorArguments]];
  BEGIN OPEN rec: parameters;
   noteString[site: @rec.string, possibleOverlap: possibleOverlap];
  END; --OPEN parameters--
 END;

StringDescriptorResultsDescription: Envoy.Description --[notes: Notes]-- =
 BEGIN OPEN notes;
 parameters: LONG POINTER TO StringDescriptorResults = noteSize[size: SIZE[StringDescriptorResults]];
  BEGIN OPEN rec: parameters;
   noteArrayDescriptor[site: @rec.desc, elementSize: SIZE[CHARACTER], possibleOverlap: possibleOverlap];
  END; --OPEN parameters--
 END;

-- No Module Initialization

END.
```

A2.2 Liaison

This example contains Liaison's two generated stub programs for the simple *ParamTest* interface: *ParamTestClientStubsImpl* and *ParamTestServerStubsImpl*. The use of these programs in a single-machine test environment is illustrated in the *ParamTestCombined* configuration (next page). Liaison's remote binding interface, *LiaisonBinder*, is included as well.

To aid the reader's understanding of this code, here is the intermodule control flow for a single steady-state call of *ParamTest.Null*.

> On the client machine, *ParamTestDriver*, which performs the timing measurements, calls *ParamTest.Null*.
>
> *ParamTestClientStubsImpl*, the client module exporting *Null*, catches the call and transmits a *call* message to the server machine.
>
> On the server machine, *ParamTestServerStubsImpl* receives the *call* message and invokes *ParamTest.Null*.
>
> *ParamTestImpl*, which exports the real implementation of *Null* on the server, performs the call and returns.
>
> *ParamTestServerStubsImpl* handles the completed call by transmitting a *return* message back to the client machine.
>
> In the client machine, *ParamTestClientStubsImpl* receives the *return* message and returns from *its* implementation of *Null*.
>
> *ParamTestDriver*, which originally called *Null*, resumes at long last and completes its timing of the call.

```
-- ParamTestCombined.config edited by BZM on October 25, 1980 12:07 PM.
-- Single machine test program that loops back in the Pup Package.
-- Run this with the Spy by typing 'Mesa ISpy ParamTestCombined'.

ParamTestCombined: CONFIGURATION
 IMPORTS DisplayDefs, Inline, InlineDefs, IODefs, SpyDefs, Time,
                  TimeDefs, SystemDefs, StreamDefs, String, Process, MiscDefs,
                  ImageDefs, ProcessDefs, FrameDefs,
                  SegmentDefs, StringDefs
 CONTROL ParamTestServerStubsImpl, ParamTestDriver
 = BEGIN

-- Network support.
TinyPup;
PktStreamImpl;
LiaisonBinderImpl;

-- Performance monitoring.
--ISpy;

-- Server modules.
serverParamTest: ParamTest ← ParamTestImpl[];
ParamTestServerStubsImpl[ serverParamTest, LiaisonBinder, PktStreamDefs, String, SystemDefs];

-- Client modules.
clientParamTest: ParamTest ← ParamTestClientStubsImpl[];
ParamTestDriver[ DisplayDefs, Inline, IODefs, clientParamTest, SegmentDefs, SpyDefs, StreamDefs, String,
                  SystemDefs, Time, TimeDefs];

END.
```

```
-- LiaisonBinder.Mesa last edited by BZM on October 23, 1980 12:16 PM

DIRECTORY
 BcdDefs USING [VersionStamp],
 PupDefs USING [PupAddress, PupSocketID];

LiaisonBinder: DEFINITIONS =
 BEGIN

 -- Types

 UniqueID: TYPE = BcdDefs.VersionStamp;
 MsgType: TYPE = {noop, call, return, error, signal};
 Message: TYPE = RECORD [type: MsgType ← noop, info: [0..7777B] ← 0];


 -- ERRORs and SIGNALs

 Problem: ERROR [reason: ErrorCode];
 ErrorCode: TYPE = {importFailed};


 -- Procedures

 ExportRemoteInterface: PROC [interfaceID: UniqueID]
  RETURNS [serverListenersSocket: PupDefs.PupSocketID];

 ImportRemoteInterface: PROC [interfaceID: UniqueID]
  RETURNS [serversAddress: PupDefs.PupAddress];

 END.
```

```
-- Stub file ParamTestClientStubsImpl.mesa was generated on 27-Oct-80 9:19:09
--          by Liaison of 27-Oct-80 9:11:25.
-- Source interface ParamTest came from file ParamTest.bcd,
--          created on 27-Oct-80 9:16:10 (3#145#) from source of 21-Oct-80 9:17:05.

DIRECTORY
 ParamTest,
 LiaisonBinder USING [ImportRemoteInterface, Message , Problem],
 PktStreamDefs USING [Call, Complaining, Complaint, EndCall, GetBlock,
                      GetWord, Handle, PutBlock, PutWord, SendNow],
 PupDefs USING [PupAddress],
 String USING [WordsForString],
 SystemDefs USING [AllocateHeapNode, AllocateHeapString];


ParamTestClientStubsImpl: MONITOR
 IMPORTS LiaisonBinder, PktStreamDefs, String, SystemDefs
 EXPORTS ParamTest
 SHARES ParamTest
  = BEGIN OPEN RPC: LiaisonBinder, Stream: PktStreamDefs;


-- Network stream trouble handler; do nothing for now.

StreamComplaint : PROC [complaint: Stream.Complaint] = { ERROR };


-- Stubbed public procedures.

Null: PUBLIC PROC [] RETURNS [] =
 BEGIN -- of procedure 1.
 ENABLE BEGIN
          Stream.Complaining => StreamComplaint[complaint];
          END;
 ParamTestServer: Stream.Handle = GetCallHandle[];
 paramSize: CARDINAL←NULL;
 -- Code to send call message to server.
 Stream.PutWord[ParamTestServer, RPC.Message[call, 1--our procedure #--]];
 -- Code to send arguments (if any).
 -- Tell server to execute the call.
 Stream.SendNow[ParamTestServer];
 SELECT LOOPHOLE[Stream.GetWord[ParamTestServer], RPC.Message].type FROM
          return => NULL;
          ENDCASE => ERROR;
 -- Code to get results (if any).
 ReturnCallHandle[ParamTestServer];
 RETURN [];
 END;

One: PUBLIC PROC [one: CARDINAL] RETURNS [CARDINAL] =
 BEGIN -- of procedure 2.
 ENABLE BEGIN
          Stream.Complaining => StreamComplaint[complaint];
          END;
 ParamTestServer: Stream.Handle = GetCallHandle[];
 paramSize: CARDINAL←NULL;
                 a: CARDINAL;
 -- Code to send call message to server.
 Stream.PutWord[ParamTestServer, RPC.Message[call, 2--our procedure #--]];
 -- Code to send arguments (if any).
                 Stream.PutWord[ParamTestServer, one];
 -- Tell server to execute the call.
 Stream.SendNow[ParamTestServer];
 SELECT LOOPHOLE[Stream.GetWord[ParamTestServer], RPC.Message].type FROM
          return => NULL;
          ENDCASE => ERROR;
 -- Code to get results (if any).
                 a ← Stream.GetWord[ParamTestServer];
 ReturnCallHandle[ParamTestServer];
 RETURN [a];
 END;
```

```
Four: PUBLIC PROC [one: CARDINAL, two: CARDINAL, three: CARDINAL, four: CARDINAL] RETURNS [CARDINAL,
CARDINAL, CARDINAL, CARDINAL] =
 BEGIN -- of procedure 4.
 ENABLE BEGIN
          Stream.Complaining => StreamComplaint[complaint];
          END;
 ParamTestServer: Stream.Handle = GetCallHandle[];
 paramSize: CARDINAL←NULL;
                    a: CARDINAL;
                    b: CARDINAL;
                    c: CARDINAL;
                    d: CARDINAL;
 -- Code to send call message to server.
 Stream.PutWord[ParamTestServer, RPC.Message[call, 4--our procedure # --]];
 -- Code to send arguments (if any).
                    Stream.PutWord[ParamTestServer, one];
                    Stream.PutWord[ParamTestServer, two];
                    Stream.PutWord[ParamTestServer, three];
                    Stream.PutWord[ParamTestServer, four];
 -- Tell server to execute the call.
 Stream.SendNow[ParamTestServer];
 SELECT LOOPHOLE[Stream.GetWord[ParamTestServer], RPC.Message].type FROM
          return => NULL;
          ENDCASE => ERROR;
 -- Code to get results (if any).
                    a ← Stream.GetWord[ParamTestServer];
                    b ← Stream.GetWord[ParamTestServer];
                    c ← Stream.GetWord[ParamTestServer];
                    d ← Stream.GetWord[ParamTestServer];
 ReturnCallHandle[ParamTestServer];
 RETURN [a, b, c, d];
 END;

TwentyArray: PUBLIC PROC [in: ParamTest.Array20] RETURNS [ParamTest.Array20] =
 BEGIN -- of procedure 8.
 ENABLE BEGIN
          Stream.Complaining => StreamComplaint[complaint];
          END;
 ParamTestServer: Stream.Handle = GetCallHandle[];
 paramSize: CARDINAL←NULL;
                    out: ParamTest.Array20;
 -- Code to send call message to server.
 Stream.PutWord[ParamTestServer, RPC.Message[call, 8--our procedure # --]];
 -- Code to send arguments (if any).
                    Stream.PutBlock[ParamTestServer,
                              [@in, 0, 2*SIZE[ParamTest.Array20]]];
 -- Tell server to execute the call.
 Stream.SendNow[ParamTestServer];
 SELECT LOOPHOLE[Stream.GetWord[ParamTestServer], RPC.Message].type FROM
          return => NULL;
          ENDCASE => ERROR;
 -- Code to get results (if any).
                                        []←Stream.GetBlock[ParamTestServer,
                                          [@out, 0,
                                                 2*SIZE[ParamTest.Array20]]];
 ReturnCallHandle[ParamTestServer];
 RETURN [out];
 END;

StringDescriptor: PUBLIC PROC [string: STRING] RETURNS [DESCRIPTOR FOR ARRAY CARDINAL OF
CHARACTER] =
 BEGIN -- of procedure 10.
 ENABLE BEGIN
          Stream.Complaining => StreamComplaint[complaint];
          END;
 ParamTestServer: Stream.Handle = GetCallHandle[];
 paramSize: CARDINAL←NULL;
                    desc: DESCRIPTOR FOR ARRAY CARDINAL OF CHARACTER;
 -- Code to send call message to server.
 Stream.PutWord[ParamTestServer, RPC.Message[call, 10--our procedure # --]];
```

```
 -- Code to send arguments (if any).
                  IF string = NIL
                           THEN Stream.PutWord[ParamTestServer, LAST[CARDINAL]]
                           ELSE Stream.PutBlock[ParamTestServer,
                                    [string, 0,
                                      2*String.WordsForString[string.length]]];
 -- Tell server to execute the call.
 Stream.SendNow[ParamTestServer];
 SELECT LOOPHOLE[Stream.GetWord[ParamTestServer], RPC.Message].type FROM
         return => NULL;
         ENDCASE => ERROR;
 -- Code to get results (if any).
                  paramSize ← Stream.GetWord[ParamTestServer];
                  IF paramSize = 0
                    THEN desc ← DESCRIPTOR[NIL,0]
                    ELSE {desc ← DESCRIPTOR[
                           SystemDefs.AllocateHeapNoc  paramSize*SIZE[CHARACTER]],
                           paramSize];
                                    []←Stream.GetBlock[ParamTestServer,
                                      [BASE[desc], 0,
                                             2*paramSize*SIZE[CHARACTER]]]};
 ReturnCallHandle[ParamTestServer];
 RETURN [desc];
 END;


-- Call handle management.

cacheEmpty: BOOLEAN;
cachedCallHandle: Stream.Handle;

InitCallHandleCache: PRIVATE PROC =
 INLINE BEGIN
 cachedCallHandle ← Stream.Call[
                  serversAddress
                  ! Stream.Complaining => StreamComplaint[complaint] ];
 cacheEmpty ← FALSE;
 END;

GetCallHandle: PRIVATE ENTRY PROC RETURNS [Stream.Handle] =
 INLINE BEGIN
 IF ~cacheEmpty
         THEN {cacheEmpty ← TRUE; RETURN[cachedCallHandle]}
         ELSE RETURN[Stream.Call[
                           serversAddress
                           ! UNWIND => NULL ]];
 END;

ReturnCallHandle: PRIVATE ENTRY PROC [callHandle: Stream.Handle] =
 INLINE BEGIN
 IF callHandle = cachedCallHandle
         THEN cacheEmpty ← FALSE
         ELSE Stream.EndCall[
                           callHandle
                           ! UNWIND => NULL ];
 END;


-- Start server module by importing a call handle for the remote interface.

serversAddress: PupDefs.PupAddress = RPC.ImportRemoteInterface[
                           interfaceID: [3,101,2518967770]
                           ! RPC.Problem => REJECT ];

InitCallHandleCache;

END.
```

```
-- Stub file ParamTestServerStubsImpl.mesa was generated on 27-Oct-80 9:19:16
--        by Liaison of 27-Oct-80 9:11:25.
-- Source interface ParamTest came from file ParamTest.bcd,
--        created on 27-Oct-80 9:16:10 (3 # 145 #) from source of 21-Oct-80 9:17:05.

DIRECTORY
 ParamTest,
 LiaisonBinder USING [ExportRemoteInterface, Message, Problem],
 PktStreamDefs USING [Complaining, GetBlock, GetWord, Handle,
                      Listen, PutBlock, PutWord, SendNow],
 String USING [WordsForString],
 SystemDefs USING [AllocateHeapNode, AllocateHeapString,
                   FreeHeapNode, FreeHeapString];

ParamTestServerStubsImpl: PROGRAM
 IMPORTS ParamTest, LiaisonBinder, PktStreamDefs, String, SystemDefs
 SHARES ParamTest
 = BEGIN OPEN RPC: LiaisonBinder, Stream: PktStreamDefs;

ReceiveClientCalls: PROC [client: Stream.Handle] =
 BEGIN
 ENABLE BEGIN
        -- Fill in your own Stream exception handler.
        Stream.Complaining => REJECT;
        END;
  msg: RPC.Message = Stream.GetWord[client];
  SELECT msg.type FROM
        call =>
         SELECT msg.info--procedure #-- FROM

           1 =>BEGIN
                paramSize: CARDINAL←NULL;
                -- Perform the call.
                [] ← ParamTest.Null[];
                -- Send RPC control message for the return.
                Stream.PutWord[client, RPC.Message[return]];
                Stream.SendNow[client];
                END; -- of procedure 1.

           2 =>BEGIN
                paramSize: CARDINAL←NULL;
                a0: CARDINAL;
                r0: CARDINAL;
                -- Get arguments (if any).
                a0 ← Stream.GetWord[client];
                -- Perform the call.
                [r0] ← ParamTest.One[a0];
                -- Send RPC control message for the return.
                Stream.PutWord[client, RPC.Message[return]];
                -- Send results (if any).
                Stream.PutWord[client, r0];
                Stream.SendNow[client];
                END; -- of procedure 2.

           4 =>BEGIN
                paramSize: CARDINAL←NULL;
                a0: CARDINAL;
                a1: CARDINAL;
                a2: CARDINAL;
                a3: CARDINAL;
                r0: CARDINAL;
                r1: CARDINAL;
                r2: CARDINAL;
                r3: CARDINAL;
                -- Get arguments (if any).
                a0 ← Stream.GetWord[client];
                a1 ← Stream.GetWord[client];
                a2 ← Stream.GetWord[client];
                a3 ← Stream.GetWord[client];
                -- Perform the call.
                [r0, r1, r2, r3] ← ParamTest.Four[a0, a1, a2, a3];
                -- Send RPC control message for the return.
```

```
                Stream.PutWord[client, RPC.Message[return]];
                -- Send results (if any).
                Stream.PutWord[client, r0];
                Stream.PutWord[client, r1];
                Stream.PutWord[client, r2];
                Stream.PutWord[client, r3];
                Stream.SendNow[client];
                END; -- of procedure 4.

        8 =>BEGIN
                paramSize: CARDINAL←NULL;
                a0: ParamTest.Array20;
                r0: ParamTest.Array20;
                -- Get arguments (if any).
                                []←Stream.GetBlock[client,
                                  [@a0, 0,
                                        2*SIZE[ParamTest.Array20]]];
                -- Perform the call.
                [r0] ← ParamTest.TwentyArray[a0];
                -- Send RPC control message for the return.
                Stream.PutWord[client, RPC.Message[return]];
                -- Send results (if any).
                Stream.PutBlock[client,
                        [@r0, 0, 2*SIZE[ParamTest.Array20]]];
                Stream.SendNow[client];
                -- Free string and descriptor arguments (if any).
                END; -- of procedure 8.

        10 =>BEGIN
                paramSize: CARDINAL←NULL;
                a0: STRING;
                r0: DESCRIPTOR FOR ARRAY CARDINAL OF CHARACTER;
                -- Get arguments (if any).
                paramSize ← Stream.GetWord[client];
                IF paramSize = LAST[CARDINAL]
                        THEN a0 ← NIL
                        ELSE {a0 ← SystemDefs.AllocateHeapString[
                                        Stream.GetWord[client]];
                                a0.length ← paramSize;
                                []←Stream.GetBlock[client,
                                  [@a0.text, 0,
                                        2*(String.WordsForString[paramSize]-2)]]};
                -- Perform the call.
                [r0] ← ParamTest.StringDescriptor[a0];
                -- Send RPC control message for the return.
                Stream.PutWord[client, RPC.Message[return]];
                -- Send results (if any).
                Stream.PutWord[client, LENGTH[r0]];
                IF LENGTH[r0] # 0
                        THEN Stream.PutBlock[client,
                                [BASE[r0], 0,
                                  2*LENGTH[r0]*SIZE[CHARACTER]]];
                Stream.SendNow[client];
                -- Free string and descriptor arguments (if any).
                IF a0 # NIL THEN SystemDefs.FreeHeapString[a0];
                END; -- of procedure 10.

        ENDCASE => ERROR;
      ENDCASE => ERROR;
END; -- ReceiveClientCalls.


-- Start server stub module by declaring the remote interface for binding.

Stream.Listen [
  listeningSocket: RPC.ExportRemoteInterface[
                                interfaceID: [3,101,2518967770]
                                ! RPC.Problem => REJECT ],
  listeningProc: ReceiveClientCalls
  ! Stream.Complaining => REJECT ];

END.
```

A2.3 EtherPkt

This example contains EtherPkt's two stub programs for the *ParamTest* interface: *EPParamTestClientStubsImpl* and *EPParamTestServerStubsImpl*. The *EtherPkt* interface, which is the stubs' fast trapdoor into the Pup Ethernet driver, is included below as well (below).

The reader is advised to understand the preceding Liaison example before proceeding with this one. The flow of control through EtherPkt's stubs is analogous to Liaison's, so reading the step-by-step description there—making the obvious module name substitutions—will be of great help here.

```
-- File EtherPkt.mesa last edited by BZM on November 7, 1980 10:31 PM.

EtherPkt: DEFINITIONS =
BEGIN

Packet: TYPE = POINTER TO PacketObject;
  -- Packet: [dest,,source ; etherPktType ; dataWords... ; etherCRC]

PacketType: TYPE = RECORD [WORD];
 RPCPktType: PacketType = [30303B];
 VoidPktType: PacketType = [1B];

PacketHeaderSize: CARDINAL = SIZE[PacketObject] - PacketDataSize;
PacketDataSize: CARDINAL = 10;

PacketObject: TYPE = MACHINE DEPENDENT RECORD [
          dest: [0..377B],
          source: [0..377B],
          type: PacketType,
          serial: CARDINAL,
          data: ARRAY [0..PacketDataSize) OF UNSPECIFIED ];

Control: TYPE = POINTER TO ControlObject;

ControlObject: TYPE = RECORD [
  pktType: CARDINAL ← VoidPktType,
  input: RECORD [
          done: BOOLEAN ← FALSE,
          packet: Packet ← NIL,
          size: CARDINAL ← 0,
          maxSize: CARDINAL ← 0,
          condition: POINTER TO CONDITION ← NIL,
          accepting: BOOLEAN ← FALSE ],
  output: RECORD [
          done: BOOLEAN ← TRUE,
          packet: Packet ← NIL,
          size: CARDINAL ← 0 ] ];

control: PUBLIC Control;

GetControl: PROC [
           pktType: CARDINAL,
           inputCondition: POINTER TO CONDITION,
           inputBuffer: Packet,
           inputBufferMaxSize: CARDINAL ← SIZE[EtherPkt.PacketObject] ]
          RETURNS [control: Control];

ReleaseControl: PROC [control: Control] RETURNS [controlNIL: Control];

Send: PROC [pkt: Packet, pktSize: CARDINAL] RETURNS [busyTryAgain: BOOLEAN];

END.
```

```
-- Stub file EPParamTestClientStubsImpl.mesa generated on 27-Oct-80 9:19:09
--        by Liaison of 27-Oct-80 9:11:25.
-- Source interface ParamTest came from file paramtest.bcd,
--        created 27-Oct-80 9:16:10 (3#145#) from source of 21-Oct-80 9:17:05.
-- Changed to do EtherPkt protocol by BZM on November 11, 1980 1:53 AM.

DIRECTORY
 ParamTest,
 EtherPkt USING[Control, GetControl, PacketHeaderSize,
                PacketObject, PacketType, RPCPktType, Send],
 LiaisonBinder USING [ImportRemoteInterface, Message, Problem],
 Process USING[MsecToTicks, SetTimeout],
 PupDefs USING [PupAddress, PupNameLookup];


EPParamTestClientStubsImpl: MONITOR
 IMPORTS EtherPkt, LiaisonBinder, Process, PupDefs
 EXPORTS ParamTest
 SHARES ParamTest
 = BEGIN OPEN RPC: LiaisonBinder;


-- These are managed by SendRequestAndReceiveReply;
inPkt, outPkt: EtherPkt.PacketObject ← [dest:, source:, type:, serial:, data:];


-- Stubbed public procedures.

Null: PUBLIC PROC [] RETURNS [] =
 BEGIN -- of procedure 1.
 -- Code to send call message to server.
 outPkt.data[0] ← RPC.Message[call, 1--our procedure #--];
 -- Code to send arguments (if any).
 -- Tell server to execute the call.
 SendRequestAndReceiveReply[dataWords: 1];
 SELECT LOOPHOLE[inPkt.data[0], RPC.Message].type FROM
        return => NULL;
        ENDCASE => ERROR;
 -- Code to get results (if any).
 RETURN [];
 END;

Two: PUBLIC PROC [one: CARDINAL, two: CARDINAL] RETURNS [CARDINAL, CARDINAL] =
 BEGIN -- of procedure 3.
 -- Code to send call message to server.
 outPkt.data[0] ← RPC.Message[call, 3--our procedure #--];
 -- Code to send arguments (if any).
 outPkt.data[1] ← one;
 outPkt.data[2] ← two;
 -- Tell server to execute the call.
 SendRequestAndReceiveReply[dataWords: 3];
 SELECT LOOPHOLE[inPkt.data[0], RPC.Message].type FROM
        return => NULL;
        ENDCASE => ERROR;
 -- Code to get results (if any).
 RETURN [inPkt.data[1], inPkt.data[2]];
 END;
```

```
-- Handle transport of remote call messages.

inputWait: CONDITION;
ether: EtherPkt.Control;
ourAddress: PupDefs.PupAddress;
reTransmitTimeInMsec: CARDINAL = 10;
retriesUntilFailure: CARDINAL ← 5--minutes--*(60*1000/reTransmitTimeInMsec);


getStats: BOOLEAN = FALSE;
totalCalls: LONG CARDINAL ← 0;
totalRetransmissions: LONG CARDINAL ← 0;
Bump: PROC [counter: POINTER TO LONG CARDINAL] =
 INLINE { IF getStats THEN counter↑ ← counter↑ + 1 };


NoCallResponse: ERROR = CODE;

InitSendRequestAndReceiveReply: PROC =
 BEGIN
 PupDefs.PupNameLookup[@ourAddress, "ME"L];
 Process.SetTimeout[@inputWait, Process.MsecToTicks[reTransmitTimeInMsec]];
 outPkt ← [
        dest: serversAddress.host,
        source: ourAddress.host,
        type: EtherPkt.RPCPktType,
        serial: 0,
        data: ];
 ether ← EtherPkt.GetControl[EtherPkt.RPCPktType, @inputWait, @inPkt];
 ether.input.accepting ← TRUE;
 END;


SendRequestAndReceiveReply: ENTRY PROC [dataWords: CARDINAL] =
 INLINE BEGIN
 Bump[@totalCalls];
 outPkt.serial ← outPkt.serial + 1;
 THROUGH [0..retriesUntilFailure) DO
          ether.input.done ← ether.output.done ← FALSE;
          WHILE EtherPkt.Send[ @outPkt,
                 EtherPkt.PacketHeaderSize + dataWords].busyTryAgain DO ENDLOOP;
          WAIT inputWait;
          IF ether.input.done AND inPkt.serial = outPkt.serial THEN EXIT;
          Bump[@totalRetransmissions];
        REPEAT
          FINISHED => ERROR NoCallResponse;
        ENDLOOP;
 END;


-- Start server module by importing a call handle for the remote interface.

serversAddress: PupDefs.PupAddress = RPC.ImportRemoteInterface[
                  interfaceID: [3,101,2518967770]
                  ! RPC.Problem => REJECT ];

InitSendRequestAndReceiveReply;

END.
```

```
-- Stub file EPParamTestServerStubsImpl.mesa generated on 27-Oct-80 9:19:16
--         by Liaison of 27-Oct-80 9:11:25.
-- Source interface ParamTest came from file paramtest.bcd,
--         created 27-Oct-80 9:16:10 (3 # 145 #) from source of 21-Oct-80 9:17:05.
-- Changed to do EtherPkt protocol by BZM on November 11, 1980 1:53 AM.

DIRECTORY
 ParamTest,
 LiaisonBinder USING [ExportRemoteInterface, Message, Problem],
 EtherPkt USING[Control, GetControl, PacketHeaderSize, PacketObject,
                PacketType, RPCPktType, Send],
 Process USING[DisableTimeout],
 PupDefs USING [PupAddress, PupNameLookup];


EPParamTestServerStubsImpl: MONITOR
 IMPORTS ParamTest, EtherPkt, LiaisonBinder, Process, PupDefs
 SHARES ParamTest
 = BEGIN OPEN RPC: LiaisonBinder;


-- These are managed by ReceiveRequest and SendReply;
inPkt, outPkt: EtherPkt.PacketObject ← [dest:, source:, type:, serial:, data:];


ReceiveClientCalls: PROC =
BEGIN
DO --Forever.
 ReceiveRequest[];
 SELECT LOOPHOLE[inPkt.data[0], RPC.Message].type FROM
        call =>
         SELECT LOOPHOLE[inPkt.data[0], RPC.Message].info--procedure #-- FROM

          1 =>BEGIN
                [] ← ParamTest.Null[];
                -- Send RPC control message for the return.
                -- Done at Init: outPkt.data[0] ← RPC.Message[return];
                -- Send results (if any).
                SendReply[1];
                END; -- of procedure 1.

          3 =>BEGIN
                -- Get arguments (if any).
                -- Perform the call.
                [outPkt.data[1], outPkt.data[2]]
                        ← ParamTest.Two[inPkt.data[1], inPkt.data[2]];
                -- Send RPC control message for the return.
                -- Done at Init: outPkt.data[0] ← RPC.Message[return];
                -- Send results (if any).
                SendReply[3];
                END; -- of procedure 3.

          2,4 => NULL; --Not included in this example

          5,6,7,8,9,10 => ERROR;

          ENDCASE => ERROR;
        ENDCASE => ERROR;
 ENDLOOP;
 END; -- ReceiveClientCalls.
```

```
-- Handle transport of remote call messages.

inputWait: CONDITION;
ether: EtherPkt.Control;
ourAddress: PupDefs.PupAddress;
lastSerial: CARDINAL;

getStats: BOOLEAN = FALSE;
requestsAccepted: LONG CARDINAL ← 0;
requestsRejected: LONG CARDINAL ← 0;
repliesResent: LONG CARDINAL ← 0;
Bump: PROC [counter: POINTER TO LONG CARDINAL] =
 INLINE { IF getStats THEN counter↑ ← counter↑ + 1 };


InitReceiveRequestAndSendReply: PROC =
 BEGIN
 PupDefs.PupNameLookup[@ourAddress, "ME"L];
 Process.DisableTimeout[@inputWait];
 outPkt ← [
        dest: ,
        source: ourAddress.host,
        type: EtherPkt.RPCPktType,
        serial: ,
        data: [RPC.Message[return], ........] ];
 lastSerial ← 0;
 ether ← EtherPkt.GetControl[EtherPkt.RPCPktType, @inputWait, @inPkt];
 ether.input.accepting ← TRUE;
 END;

ReceiveRequest: ENTRY PROC =
 INLINE BEGIN
 DO
        ether.input.done ← FALSE;
        UNTIL ether.input.done DO WAIT inputWait ENDLOOP;
        SELECT inPkt.serial FROM
          (lastSerial←lastSerial + 1) => EXIT;
          (lastSerial←lastSerial-1) =>
                          BEGIN
                          SendReply[ether.output.size-EtherPkt.PacketHeaderSize];
                          Bump[@repliesResent];
                          END;
          ENDCASE => Bump[@requestsRejected];
        ENDLOOP;
 Bump[@requestsAccepted];
 END;

SendReply: PROC [dataWords: CARDINAL] =
 INLINE BEGIN
 outPkt.dest ← inPkt.source;
 outPkt.serial ← lastSerial;
 ether.output.done ← FALSE;
 WHILE EtherPkt.Send[ @outPkt,
                EtherPkt.PacketHeaderSize + dataWords].busyTryAgain DO ENDLOOP;
 END;


-- Start server stub module by declaring the remote interface for binding.

[] ← RPC.ExportRemoteInterface[
                interfaceID: [3,101,2518967770]
                ! RPC.Problem => REJECT ];

InitReceiveRequestAndSendReply;

ReceiveClientCalls;

END.
```

Kusaie—East Carolines
5° 19' N   163° 02' E
The westernmost outpost for American whalers and missionaries in the mid 1800s

References

[1] Gene Ball.
Alto as Terminal.
Internal memorandum, Carnegie-Mellon University, Computer Science Department, April, 1980.
A complete description of CMU's Vax-Alto RPC environment.

[2] Forest Baskett, John H. Howard, and John T. Montague.
Task communication in Demos.
Operating Systems Review 11(5):23–31, November, 1977.
Designed for the Cray-1, Demos passes messages over links for all communication. The scheme is impure because mechanisms for sharing memory—e.g., I/O buffers—are incorporated as well.

[3] F. L. Bauer and H. Wössner.
The *Plankalkül* of Konrad Zuse: a forerunner of today's programming languages.
Communications of the ACM 15(7):678–85, July, 1972.
An overview of the two-dimensional language that Zuse—who is credited with building the first (nonelectronic) stored program computer—designed after the war.

[4] Jon Louis Bentley.
Writing Efficient Code.
Technical Report, Carnegie-Mellon University, Computer Science Department, April, 1981.
A thorough presentation of language-independent programming optimization techniques. Measured speedups of 5–25 times are obtained for operational programs.

[5] A. D. Birrell and R. M. Needham.
A universal file server.
IEEE Transactions on Software Engineering SE-6(5):450–53, September, 1980.
A brief description of the file server used in the Cambridge Ring.

[6] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder.
Grapevine.
Communications of the ACM, should appear early in 1982.
Also available as a Xerox Palo Alto Research Center technical report.
A complete description of the Grapevine distributed mail transport system, including the registration service.

[7] Daniel G. Bobrow and Douglas W. Clark.
Compact encodings of list structure.
Transactions on Programming Languages and Systems 1(2):266–86, October, 1979.
The benefits of compact encodings are discussed and several existing schemes are examined.

[8] David R. Boggs, John F. Shoch, Edward A. Taft, and Robert M. Metcalfe.
Pup: an internetwork architecture.
IEEE Transactions on Communications Com-28(4):612–24, April, 1980.
The simplicity of the Pup architecture and its fundamental end-to-end datagram approach is touted. Pup's encapsulation within various networks—including the Ethernet, Arpanet, and Packet Radio—are discussed.

[9] David R. Boggs.
Internetwork Broadcasting.
PhD thesis, Stanford University, Electrical Engineering Department, 1981.
Boggs's thesis discusses the merits and problems of broadcasting in packet-switched internets. A number of excellent examples are given and the problems of *reliable* broadcasting are presented.

[10] Per Brinch Hansen.
Operating System Principles.
Prentice-Hall, 1973.
Among its many achievements, this book introduces the notions of monitor and critical region, and has an excellent discussion of the RC4000 message-based operating system.

[11] Per Brinch Hansen.
The programming language Concurrent Pascal.
IEEE Transactions on Software Engineering SE-1(2):199–207, June, 1975.
Brinch Hansen's multiprogramming extensions to Pascal include monitors with *queues*, which are condition variables that exactly one process can wait on.

[12] Per Brinch Hansen.
Distributed processes: a concurrent programming concept.
Communications of the ACM 21(11):934–41, November, 1978.
Brinch Hansen's language proposal for distributed processes—which share no memory—includes a clean RPC-like mechanism.

[13] CMU Spice Committee.
Proposal for a Joint Effort in Personal Scientific Computing.
Technical Report, Computer Science Department, Carnegie-Mellon University, August, 1979.
This proposal outlines CMU's ambitious plans for a department-wide distributed environment of personal and powerhouse computers.

[14] Vinton G. Cerf and Robert E. Kahn.
A protocol for packet network interconnection.
IEEE Transactions on Communications Com-22(5):637–48, May, 1974.
This paper founds the idea of an internet and introduces *gateways* as their interconnection portals.

[15] David R. Cheriton, Michael A. Malcolm, Lawrence S. Melen, and Gary R. Sager.
Thoth, a portable real-time operating system.
Communications of the ACM 22(2):105–115, February, 1979.
Thoth is an elegant message-passing system written in a descendent of Bcpl. It has a small kernel and good performance.

[16] Danny Cohen.
On holy wars and a plea for peace.
IEEE Computer 14, to appear, 1981.
Also available as USC Information Sciences Institute Internal Note IEN 137.

An entertaining discussion of bit transmission and data representation issues.

[17] Robert P. Cook.
*MOD—a language for distributed programming.
In *Proceedings of the First International Conference on Distributed Computing Systems*, pages 233–41. IEEE, October, 1979.
StarMod (as *MOD has since been renamed) is derived from Wirth's Modula. It extends Modula's shared-memory processes and interface modules (monitors) with *network modules* of *processor modules* that do not share memory. The implementation that Cook describes runs on a single machine, not in a physically distributed system.

[18] Robert P. Cook.
Abstractions for distributed computing.
This is an unpublished position paper for the SIGPLAN-SIGOPS *Workshop on Fundamental Issues in Distributed Computing*, held December 14–17, 1980. Cook is with the Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 53706.
Cook describes *ports* and *regions*, which are message-oriented extensions to StarMod. The concept of *scheduler modules* is developed as well.

[19] Stephen D. Crocker, John F. Heafner, Robert M. Metcalfe, and Jonathan B. Postel.
Function-oriented protocols for the ARPA computer network.
In *AFIPS Conference Proceedings of the Spring Joint Computer Conference*, pages 271–79. AFIPS Press, May, 1972.
In this early paper the word *server* is explicitly used to describe telnet and remote job entry services.

[20] William R. Crowther.
Private communication, February, 1981.
Crowther describes an IC layout program that could be easily optimized with language-level RPC.

[21] Yogen K. Dalal and Robert M. Metcalfe.
Reverse path forwarding of broadcast packets.
Communications of the ACM 21(12):1040–48, December, 1978.
Dalal and Metcalfe review five existing methods of broadcasting in a packet-switched network and propose a new method of their own. The ability of each method to perform a reliable broadcast is evaluated.

[22] Jerome A. Feldman and Robert F. Sproull.
System support for the Stanford hand-eye system.
In *Proceedings of the Second International Joint Conference on Artificial Intelligence*, pages 183–89. IJCAI, London, September, 1971.
Sproull and Feldman talk about extensions to Sail and Tops10 which allowed them to do IPC via *message procedures*. While not really RPC in the true sense, their scheme did allow a remote call to have apparently normal syntax.

[23] Jerome A. Feldman.
High level programming for distributed computing.
Communications of the ACM 22(1):353–68, June, 1979.
Feldman discusses Plits, a Sail-based language with modules and messages. While control is message- rather than procedure-based, the problems of parameter functionality, data translation, and machine failures are briefly mentioned.

[24] Samuel Fuller, John Ousterhout, Levy Raskin, Paul Rubinfeld, Pradeep Sindhu, and Richard Swan.
Multi-microprocessors: an overview and working example.
Proceedings of the IEEE 66(2):216–28, February, 1978.
A discussion of the Cm* hardware architecture.

[25] David K. Gifford.
Information Storage in a Decentralized Computer System.
PhD thesis, Stanford University, Electrical Engineering Department, 1981.
Gifford's thesis develops a general transaction-oriented distributed file system with uniform naming.

[26] Xerox Learning Research Group.
The Smalltalk-80 System.
BYTE 6(8):36–48, August, 1981.
This issue of *BYTE* is devoted to Smalltalk; only the introductory article is referenced here.
A discussion of Smalltalk, an object-oriented, message-passing programming language and environment.

[27] James N. Gray.
A Discussion of Distributed Systems.
Research Report RJ2699(34594), IBM Research, San Jose, August, 1979.
Gray's discussion includes mention of the CICS remote procedure capability.

[28] David Gries and Gary Levin.
Assignment and procedure call proof rules.
Transactions on Programming Languages and Systems 2(4):564–79, October, 1980.
Gries and Levin include a good discussion of aliasing and its impact on verification.

[29] Loretta Rose Guarino.
Control and Communication in Programmed Systems.
PhD thesis, Carnegie-Mellon University, Computer Science Department, September, 1980.
Guarino's thesis develops a very general model of control which subsumes both procedure calling and message passing.

[30] Griffith Hamlin Jr.
Configurable Applications for Satellite Graphics.
PhD thesis, Computer Science Department, University of North Carolina at Chapel Hill, 1975.
Hamlin's early Cages system includes novel uses of remote procedures in a special two-processor PL1 environment. He handles remote procedures, exceptions (ON-conditions), and global variables.

[31] Jack Haverty.
Thoughts on Interactions in Distributed Services.
Request For Comments 722, Network Working Group, SRI Augmentation Research Center, September, 1976.
Haverty discusses network servers and the great utility of the request-response discipline for many classes of user-server interactions.

[32] Frank E. Heart, Robert E. Kahn, Severo M. Ornstein, William R. Crowther, and David C. Walden.
The interface message processor for the ARPA computer network.
In *AFIPS Conference Proceedings of the Spring Joint Computer Conference*, pages 551–67. AFIPS Press, May, 1970.
This is the classic Arpanet reference. Discussion focuses almost exclusively on the IMP.

[33] Maurice Peter Herlihy.
Transmitting abstract values in messages.
Master's thesis, MIT Laboratory for Computer Science, April, 1980.
Herlihy's thesis discusses the transmission of strongly typed objects in messages, including shared and cyclic structures.

[34] Maurice Herlihy, Gerald Leitner, and Karen Sollins (editors).
Report on the workshop on fundamental issues in distributed computing.
SIGPLAN Notices 15(3):9–38, July, 1981.
At this workshop, researchers in systems and programming languages discussed their views on atomicity, protection, naming, communications, and other practical issues in real distributed systems.

[35] Carl Hewitt.
Viewing control structures as patterns of passing messages.
Artificial Intelligence 8(3):323–64, June, 1977.
Hewitt provides an introduction to actors in this accessible but wordy paper.

[36] C. A. R. Hoare.
An axiomatic basis of computer programming.
Communications of the ACM 12(10):576–80, October, 1969.
Hoare's paper defines the standard $\{P\}\ S\ \{Q\}$ verification notation for preconditions and postconditions in Algol-like programs.

[37] C. A. R. Hoare.
Monitors: an operating system structuring concept.
Communications of the ACM 17(10):549–57, October, 1974.
Hoare's classic paper elaborates Brinch Hansens's concept of *monitor* into that of a synchronized abstract data type. Signals and condition variables are introduced for fine-grain synchronization.

[38] C. A. R. Hoare.
Communicating sequential processes.
Communications of the ACM 21(8):666–77, August, 1978.
Hoare presents a simple language-level message-passing model for interprocess communication.

[39] Intel Corporation.
Introduction to the iAPX 432 Architecture.
Intel Corporation, Santa Clara, CA 95051, 1981.
Manual order no. 171821-001.
An overview of the 432 architecture that only briefly describes the packet-switched *interconnect bus.*

[40] International Business Machines.
Customer Information Control System/Virtual Storage, Version 1, Release 5.
International Business Machines, White Plains, New York, 1980.
Form No. SC33-0068-2.
This CICS document includes a description of its brand of RPC in chapter 7.2.

[41] Anita K. Jones, Robert J. Chansler Jr., Ivor Durham, Karsten Schwans, and Steven R. Vegdahl.
StarOS, a multiprocessor operating system for the support of task forces.
Operating Systems Review 13(5):117–27, December, 1979.
An overview of StarOS including its capability mechanism and message-passing primitives.

[42] Stephen R. Kimbleton, Helen M. Wood, and M. L. Fitzgerald.
Network operating systems—an implementation approach.
In *AFIPS Conference Proceedings of the National Computer Conference*, pages 773–82. AFIPS Press, June, 1978.
The end of this paper includes a description of remote record access and translation via a third-party network translation server.

[43] Leslie Lamport.
Time, clocks, and the ordering of events in a distributed system.
Communications of the ACM 21(7):558–65, July, 1978.
An excellent discussion of logical and physical clocks. Lamport gives a novel algorithm for synchronizing a set of distributed clocks without setting any of them backward in time.

[44] Butler W. Lampson, James G. Mitchell, and Edwin H. Satterthwaite.
On the transfer of control between contexts.
In G. Goos and J. Hartmanis, editor, *Lecture Notes in Computer Science*, pages 181–203. Springer-Verlag, 1974.

This paper develops a general, low-level model of control transfer using the *Transfer* primitive. An efficient implementation for procedure call is demonstrated.

[45] Butler W. Lampson, James. J. Horning, Ralph L. London, James G. Mitchell, and Gerald J. Popek.
Report on the programming language Euclid.
SIGPLAN Notices 12(2):1–79, February, 1977.
The rather complicated specification for Euclid, a strongly typed system programming language.

[46] Butler W. Lampson and Howard E. Sturgis.
Crash recovery in a distributed data storage system.
Communications of the ACM, to appear—accepted for publication on 8 July 1977.
The underground classic in all its glory. Lampson and Sturgis define a model for a distributed system and show how to build *atomic actions* that operate on *stable storage* (crash-proof storage). Lampson published an early version of this paper in *Distributed Systems: Architecture and Implementation, an Advanced Course* (below).

[47] Butler W. Lampson and David D. Redell.
Experience with processes and monitors in Mesa.
Communications of the ACM 23(2):105–117, February, 1980.
A discussion of the design of Mesa's concurrency machinery.

[48] Butler W. Lampson.
Atomic transactions.
In Butler W. Lampson, editor, *Distributed Systems: Architecture and Implementation, an Advanced Course*, chapter 11. Springer-Verlag, 1981.
Lampson's general discussion of transactions defines stable storage and uses remote procedures for the implementation.

[49] Butler W. Lampson.
Applications and protocols.
In Butler W. Lampson, editor, *Distributed Systems: Architecture and Implementation, an Advanced Course*, chapter 14. Springer-Verlag, 1981.
Section 14.9 covers remote procedures.
Lampson discusses a last-of-many semantics RPC algorithm. An orphan discussion is included.

[50] Butler W. Lampson, Douglas W. Clark, Gene A. McDaniel, Severo M. Ornstein, and Kenneth A. Pier.
The Dorado, A High-Performance Personal Computer: Three Papers.
Technical Report CSL-81-1, Xerox Palo Alto Research Center, January, 1981.
The Dorado processor, instruction-fetch unit, and memory system papers.

[51] Keith A. Lantz.
Uniform Interfaces for Distributed Systems.
PhD thesis, Computer Science Department, University of Rochester, May, 1980.
Lantz's thesis focuses on a number of issues in the RIG system design. An appendix covers the hierarchy of message-passing primitives.

[52] Hugh C. Lauer and Roger M. Needham.
On the duality of operating system structures.
Operating Systems Review 13(2):3–19, April, 1979.
Under some loose assumptions, messages and procedures are shown to have the same power for operating system communication. The authors claim that the choice between these primitives should be based on considerations of the programming environment.

[53] Roy Levin, John McQuillan, and Richard Schantz.
Distributed systems.
Operating Systems Review 11(1):14–19, January, 1977.

The authors, discussing needed research in distributed systems, call for standardization of data and control descriptions for internetwork communication.

[54] Roy Levin.
Program Structures for Exceptional Condition Handling.
PhD thesis, Carnegie-Mellon University, Computer Science Department, June, 1977.
Levin's thesis talks about the spectrum of exception-handling issues. Special attention is paid to the problems of passing exceptions among cooperating processes.

[55] Paul H. Levine.
Facilitating interprocess communication in a heterogeneous network environment.
Master's thesis, MIT Department of Electrical Engineering and Computer Science, July, 1977.
Levine's thesis explores methods of translating data between communicating heterogeneous processors. Three main methods are discussed and a "standard intermediate representation" approach is recommended.

[56] Barbara Liskov, Alan Synder, Russell Atkinson, and Craig Schaffert.
Abstraction mechanisms in Clu.
Communications of the ACM 20(8):564–76, August, 1977.
The authors discuss Clu and some implementation considerations.

[57] Barbara Liskov.
Primitives for distributed computing.
Operating Systems Review 13(5):33–42, December, 1979.
An extension to Clu called *Guardians* is defined. Guardians communicate via messages, not remote procedures, but Clu does provide strong typechecking for messages. At-least-once semantics are favored.

[58] Barbara Liskov.
Remote Procedure Call.
DSG Note 64, MIT Laboratory for Computer Science, June, 1980.
Liskov discusses at-least-once, last-one (sequentiality), and at-most-once (atomic) invocation semantics. She rejects last-one and favors at-most-once.

[59] Barbara Liskov.
Linguistic Support for Distributed Programs: A Status Report.
Computation Structures Group Memo 201, MIT Laboratory for Computer Science, October, 1980.
Liskov changes Guardian invocation semantics to at-most-once, adding a built-in transaction mechanism.

[60] Edward M. McCreight.
The Xerox Research personal computers.
This was a Distinguished Lecture about the Alto, Dolphin, and Dorado computers given at the CMU Computer Science Department in January, 1981.
McCreight describes the detailed architecture and performance of the three processors.

[61] Robert M. Metcalfe and David R. Boggs.
Ethernet: distributed packet switching for local computer networks.
Communications of the ACM 19(7):395–404, July, 1976.
Metcalfe and Boggs explain one of the first high-speed local networks.

[62] James G. Mitchell, William Maybury, and Richard Sweet.
Mesa Language Manual.
Technical Report CSL-79-3, Xerox Palo Alto Research Center, April, 1979.
Mesa is a strongly typed implementation language descended from Pascal. Chapters 1–6 cover the basic language; chapter 7 discusses modules, configurations, and binding; chapter 8 covers exceptions; chapter 10 talks about concurrency.

[63] Roger M. Needham and Michael D. Schroeder.
Using encryption for authentication in large networks of computers.
Communications of the ACM 21(12):993–99, December, 1978.
An excellent investigation of encryption techniques for the security of network communications.

[64] Roger Needham.
Private communication, December, 1979.
Needham describes an addition to the Cambridge Ring wherein a local procedure interface to the local file system will be (transparently) replaced with a remote procedure interface to a file server.

[65] Bruce Nelson.
Thesis problems in remote procedure call.
Internal memorandum, Xerox Palo Alto Research Center, August, 1979.
This is a preliminary thesis study. It has a long section on local and remote Mesa ports.

[66] Bruce Nelson.
RpcLogs, the computer results of an RPC performance evaluation, December, 1980.
These are the unprocessed call timings and PC histograms from chapter 6's performance evaluation.

[67] Derek C. Oppen and Yogen K. Dalal.
The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment.
Technical Report OPD-T8103, Xerox Office Products Division, August, 1981.
You can write to Oppen and Dalal at Xerox OPD, 3333 Coyote Hill Road, Palo Alto, California 94304.
Dalal and Oppen develop a general approach to naming and locating objects in internetwork environments.

[68] John K. Ousterhout, Donald A. Scelza, and Pradeep S. Sindhu.
Medusa: an experiment in distributed operating system structure.
Communications of the ACM 23(2):92–105, February, 1980.
An overview of Medusa including a brief discussion of its exception facilities.

[69] James L. Peterson.
Notes on a workshop on distributed computing.
Operating Systems Review 13(3):18–30, July, 1979.
The notes are interesting reading, especially the parts on procedures, transparency, and reliability.

[70] Jonathan B. Postel and James E. White.
Procedure Call Protocol Documents, Version 2.
Request For Comments 674, Network Working Group, SRI Augmentation Research Center, December, 1974.
This report describes other documents that define the procedure call protocol developed by an SRI team for the National Software Works. Particularly notable is *The Low-Level Debug Package*, which describes a remote debugger.

[71] Jonathan B. Postel.
Internetwork protocol approaches.
IEEE Transactions on Communications Com-28(4):604–11, April, 1980.
An overview of leading internet protocols that contrasts datagrams and virtual circuits.

[72] Richard F. Rashid.
An Interprocess Communication Facility for Unix.
Technical Report, Carnegie-Mellon University, Computer Science Department, June, 1980.
Rashid's IPC will be available in the CMU internet, especially on the Vaxen (which run Unix) and the Perqs.

[73] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell.
Pilot: an operating system for a personal computer.
Communications of the ACM 23(2):81–92, February, 1980.
A description of a multiprogramming operating system for a single-user machine. Pilot is written in Mesa and provides virtual memory, files, streams, and internet communication.

[74] Larry Roberts and Barry Wessler.
Computer network development to achieve resource sharing.
In *AFIPS Conference Proceedings of the Spring Joint Computer Conference*, pages 543–49. AFIPS, June, 1970.
Roberts discusses the motivation and goals of the Arpanet.

[75] Jerome H. Saltzer.
Research problems of decentralized systems with largely autonomous nodes.
Operating Systems Review 12(1):43–52, January, 1978.
Saltzer provides a good discussion of distributed computing issues in the context of the object model.

[76] Richard E. Schantz.
A Commentary on Procedure Calling as a Network Protocol.
Request For Comments 684, Network Working Group, SRI Augmentation Research Center, April, 1975.
Schantz attacks RPC—specifically, RFC674—as an inadequate and improper model for IPC. He promotes a message-based model in its place.

[77] Stephen A. Schuman, Edmund M. Clarke Jr., and Christos N. Nikolaou.
Programming Distributed Applications in Ada: A First Approach.
Technical Report CADD-8103-3102, Massachusetts Computer Associates, March, 1981.
A description of how to add remote procedures to Ada by making no changes to the language itself. The result is a nice exposition, but it lacks much practical interest because the language-level mapping of local to remote calls is burdened with excessive use of Ada's task machinery.

[78] John F. Shoch and Jon A. Hupp.
Notes on the "Worm" programs—some early experience with a distributed computation.
Technical Report SSL-80-3, Xerox Palo Alto Research Center, May, 1980.
A good discussion of a self-replicating program structure for distributed computations.

[79] Marvin H. Soloman and Raphael A. Finkel.
The Roscoe distributed operating system.
Operating Systems Review 13(5):108–114, December, 1979.
Roscoe uses pure message passing for *all* communication between both kernel and user processes.

[80] Alfred Z. Spector.
Extending local network interfaces to provide more efficient interprocessor communication facilities.
In *Distributed Processing—New Directions for a New Decade*, pages 6–13. ACM, November, 1980.
Spector talks about synchronous remote memory operations, including his remote read/write work on the Alto.

[81] Alfred Z. Spector.
Performing Remote Operations Efficiently on a Local Computer Network.
Technical Report STAN-CS-80-831, Computer Science Department, Stanford University, December, 1980.
In this report Spector introduces his remote reference model. The end of the paper includes the timing data for remote instructions on the Alto.

[82] Robert F. Sproull and Dan Cohen.
High-level protocols.
Proceedings of the IEEE 66(11):1371–86, November, 1978.
A good discussion of the relationship between high-level languages and high-level protocols. Remote procedures are advocated for many internet interactions. The notion that an HLP definition is an HLL interface is *almost* developed. Binding and linking are discussed for both.

[83] Robert F. Sproull.
Private communication, December, 1980.
Sproull made his structuralists versus performers remark while commenting on the bulkiness of Liaison's stubs (appendix 2).

[84] Bjarne Stroustrup.
An intermodule communication system for a distributed computer system.
In *Proceedings of the First International Conference on Distributed Computing Systems*, pages 412–18. IEEE, October, 1979.
Stroustrup describes an RPC-like remote invocation mechanism for use in a distributed system with uniform capability addressing. Parameters are migrated between machines as necessary. A simulation of an operating system for a (simulated) ring network is used to make some performance estimates.

[85] Howard E. Sturgis, James G. Mitchell, and Jay E. Israel.
Issues in the design and use of a distributed file system.
Operating Systems Review 14(3):55–69, July, 1980.
An overview of the Juniper file server that does not have many real details.

[86] Ivan E. Sutherland, Charles E. Molnar, Robert F. Sproull, and J. Craig Mudge.
The TriMosBus.
In Charles L. Seitz, editor, *Proceedings of the Caltech Conference on Very Large Scale Integration*, pages 395–427. Caltech Computer Science Department, January, 1979.
The end of the paper discusses the resemblance of TriMosBus messages to high-level network protocols.

[87] Daniel Swinehart, Gene McDaniel, and David Boggs.
WFS: a simple shared file system for a distributed environment.
Operating Systems Review 13(5):9–17, December, 1979.
WFS is a high performance server that obtains much of its efficiency by using connectionless protocols.

[88] Warren Teitelman.
Interlisp Reference Manual.
Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304, 1978.
The complete lisp tome.

[89] Charles P. Thacker, Edward M. McCreight, Butler W. Lampson, Robert F. Sproull, and David R. Boggs.
Alto: a personal computer.
In Dan Siewiorek, C. Gordon Bell, and Allen Newell, editor, *Computer Structures: Readings and Examples, Second Edition*, chapter 33. McGraw-Hill, 1981.
Also published as Xerox Palo Alto Research Center report CSL-79-11.
The principal characteristics of Xerox's Alto personal computer are discussed.

[90] Robert Thomas and Stuart Schaffner.
MSG: The Interprocess Communication Facility for the National Software Works.
Technical Report 3483, Bolt, Beranek, and Newman, December, 1976.
A discussion of the comprehensive message-based IPC used in the NSW implementation.

[91] United States Department of Defense.
Reference Manual for the Ada Programming Language.
United States Department of Defense, 1980.
Ada is intended to be a powerful and universal language. Time will tell if these good intentions come to pass—constructing an efficient implementation for the full language is a formidable task.

[92] David C. Walden.
A system for interprocess communication in a resource-sharing computer network.
Communications of the ACM 15(4):221–30, April, 1972.
One of the earliest descriptions of an IPC facility. Walden's pioneering scheme was an extension of the Arpanet's Initial Connection Protocol.

[93] Peter J. L. Wallis.
External representations of objects of user-defined type.
Transactions on Programming Languages and Systems 2(2):137–52, April, 1980.
Some problems of external representations are discussed in the context of the PPL portable programming language.

[94] Richard W. Watson and John G. Fletcher.
An architecture for support of network operating system services.
Computer Networks 4(1):33–49, February, 1980.
Watson and Fletcher develop a general framework for constructing network operating systems. The high-level goals of network operating systems are covered, but the paper's emphasis is on the low-level internetwork and interprocess communication layers rather than on transparent high-level services.

[95] James E. White.
A high-level framework for network-based resource sharing.
In *AFIPS Proceedings of of the National Computer Conference*, pages 561–70. AFIPS, June, 1976.
White develops a request-response protocol model (with arguments and results) as an alternative to the application-specific protocols used in the Arpanet. RPC is seen as a natural and straightforward extension of these request-response protocols.

[96] James E. White.
Elements of a distributed programming system.
Journal of Computer Languages 2(4):117–34, April, 1977.
In this sequel to the NCC paper, White enhances his model to include a specific remote procedure call protocol (PCP). Advanced problems such as communicating global variables, providing packages (interfaces), and performing nonprocedural control transfers are discussed. Some important areas for further research are listed.

[97] Niklaus Wirth.
Modula: a language for modular multiprogramming.
Software—Practice and Experience 7(1):3–35, January, 1977.
Wirth's excellent description of Modula includes a discussion of *interface modules* and *signals*, which are Modula's version of monitors and condition variables. Companion papers in the same issue have examples of Modula and language implementation hints.

[98] Hubert Zimmermann.
OSI reference model—the ISO model of architecture for open systems interconnection.
IEEE Transactions on Communications Com-28(4):425–32, April, 1980.
The OSI internet architecture is discussed. Of special interest is the OSI protocol hierarchy.

Remote Procedure Call

A Potpourri and Thesis Summary

Bruce Jay Nelson

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania

4 May 1981

1 Introduction

The rapid growth of intercomputer communication has outpaced the development of high-level language primitives that would put this communication ability directly in the hands of the programmer. One way to bridge this gap is with *remote procedure call* (RPC). Remote procedures are logically equivalent to the local procedures found in algorithmic programming languages—e.g., Ada, Algol, Bliss, C, Lisp, Mesa, Pascal, and PL1. A remote procedure call, however, is between programs in two distinct (virtual) machines that share no memory, not between two programs in the same machine or address space. The advantage of making local and remote procedures indistinguishable—that is, making them semantically and syntactically *transparent*—is that the programmers of distributed systems can operate at a comfortable level of communication abstraction: the convenient level of procedure call. Programmers do not have to deal with the thorny issues of unreliable communication and multiple crashes that are so common in distributed systems, but are almost always absent—or ignored—in conventional systems.

Scope and Goals

My dissertation investigates the details of remote procedure mechanisms for homogeneously programmed distributed systems using state-of-the-art programming languages. It explores all of the issues and presents an implementation that is tempered with the results of an empirical performance evaluation of operational RPC schemes. The target setting for the design is based on the Alto-Mesa programming environment and Ethernet network. This particular choice of a concrete environment does not compromise the validity of the design or results for similar distributed systems, say CMU's Spice environment. Some design decisions will certainly change for environments with different characteristics, notably communication properties and process switching times. But the basic design principles and performance lessons of this dissertation are valid across a spectrum of distributed systems.

To prove its thesis the dissertation attacks three major goals.

> *Desirability of remote procedure call.* Chapters 2 and 3 show that remote procedure call is a desirable and satisfactory primitive for distributed program control and communication.
>
> *Transparency of remote procedure call.* Chapter 2 introduces the important RPC issues and emphasizes the need for a clear statement of "remote" procedure semantics in the presence of machine crashes. Chapter 4 addresses these issues in detail and extracts a set of properties that must be satisfied by any RPC mechanism that has local and remote transparency. Chapter 5 presents Emissary, a high-performance RPC implementation that has these properties, showing that remote procedures are a viable primitive for distributed programming.
>
> *Efficiency of remote procedure call.* Chapter 6 contains a performance evaluation of a large family of operational RPC mechanisms. Empirical measurements are accompanied by an analysis that yields a series of important implementation performance lessons.

The next few sections present some highlights from the body of the thesis. This potpourri of technical material is necessarily incomplete, and the serious reader is urged to put this summary

down immediately and read (or even skim) the dissertation instead. Those who read on: smile and be forewarned!

2 Definitions, Examples, Primitives, and Models

Remote Procedure Call

> Remote procedure call is the synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel.

The terms *remote procedure* and *remote procedure call* have been used by members of Arpa's Network Working Group to describe calling procedures through the Arpanet for many years.

Autonomy

A key notion of "remoteness" is that the narrow channel is the primary medium programs use to communicate. This is why the definition explicitly excludes traditional communication channels such as shared memory. Programs communicating with remote procedures are best viewed as isolated, autonomous entities whose preferred communication is along narrow and clearly defined pathways—communication shortcuts are neither allowed nor desirable. This autonomous node environment of distinct virtual memories and channeled communication is frequently called a *distributed system.*

Because remote procedure calls take place between autonomous programs, a failure in one program is usually detectable in another (e.g., when a remote call does not return). This isolation and detectability of failures is a fundamental property of distributed programs, and it distinguishes distributed programs from local programs (using local calls) where a failure usually stops everything (e.g., both caller and callee).

Unreliable Communication

The notion of inherently *unreliable* communication is essential in distributed systems where communication is often through error-prone channels. For example, one communication medium frequently used in distributed systems is what Cerf and Kahn call an *internetwork* (*IEEE Transactions on Communication* 22(5):637-48, May, 1974), a highly but not totally reliable medium whose transmissions are subject with nonzero probabilities to losses, duplications, errors, and delays. This unreliability can distinguish remote procedure implementations from local procedure implementations. For instance, procedure call in uniprocessors and shared-memory multiprocessors is almost always supported by the hardware and is implicitly assumed to be totally reliable. In loosely coupled distributed systems, on the other hand, there is no usually no hardware support for intermachine procedure calls. The software or firmware RPC implementation must cope with unreliable communication, not error-free shared-memory (which the hardware makes reliable). But

while overcoming unreliable communication is often an important consideration for an RPC implementation, it need not be a factor when communication *is* reliable. For example, programs conversing with RPC can be executing in autonomous virtual processors on the same physical machine. If an underlying kernel uses message-passing to provide interprocessor communication, then not only is a narrow channel is being used, but the channel is probably totally reliable if the kernel uses memory-to-memory message copying.

It is important to observe that the reliability of communication usually has no qualitative impact on RPC. Whether the underlying RPC implementation uses a reliable or unreliable channel, the autonomous programs at the ends of the channel can still fail independently. Thus the key aspects of distributed programs remain autonomy and isolation of failure, although it is likely that communication characteristics—specifically, speed and reliability—will have a marked *quantitative* impact on RPC performance.

In summary, the sole requirement of remoteness is that the primary communication medium used by autonomous programs be a narrow channel. Furthermore, because the unreliability of communication is a central theme in distributed systems, the thesis uses the internetwork (internet) as the standard channel model.

Coroutines, Exceptions, and Other Control Transfers

A third important notion in the RPC definition is that the control transfer implied by RPC is not limited to procedure call itself. While procedure call is the dominant control transfer discipline in procedural languages, there are usually less frequently used language-level primitives as well. These can include coroutine transfers, exceptions, and module initialization and finalization. Thus a general RPC scheme must support arbitrary synchronous transfers such as those proposed by Lampson, Mitchell, and Satterthwaite in their paper "On the Transfer of Control between Contexts" (*Lecture Notes in Computer Science*, Springer-Verlag, 1974). Always remember that while remote procedure call is a popular and intuitive name, RPC is actually a misnomer that covers the entire spectrum of remote transfers in procedural languages.

Synchrony and Concurrency

The last essential notion in the definition is that *synchronous* transfers are required because high-level languages usually have only synchronous local communication primitives. For example, calling a procedure, transferring to a coroutine, and raising an exception normally do not start a concurrent activity that executes the caller and callee in parallel. The limitation to synchronous remote transfers does not hinder overall concurrency. A language's synchronous transfer primitives—including remote procedures, remote coroutines, and so forth—can always be composed with the language's *independent* concurrency operators—COBEGIN, FORK, and so on—into parallel activities. Of course, if a language does not have concurrency operators, then the synchronous restriction prevents concurrency—just as the original language does.

Remote Procedure Example

The prevalence of procedure call as a control primitive in most programming languages makes RPC a very intuitive concept once one understands that the remote call is simply going through a narrow channel such as a network. This is illustrated in figure 1 with a suggestive Mesa syntax. In the example, procedure P_C in module M_C on machine C is calling procedure P_S in module M_S on server machine S. C transmits P_S's arguments to S, where S computes P_S and transmits P_S's results back to C.

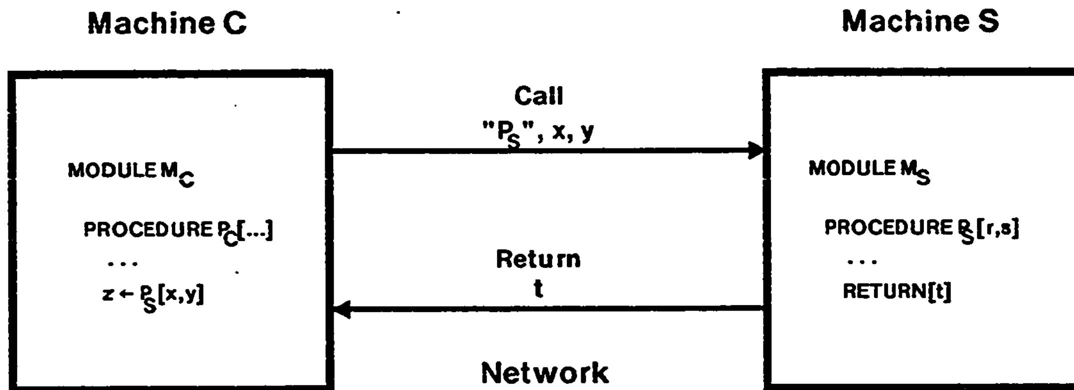


Figure 1: A simple remote procedure call from client C to server S.

The beauty of this example is that the programmers of modules M_C and M_S *do not know there is a network between them.* RPC, when done in the style proposed in the thesis, is transparent to the programmers of both the client *and* server modules. The programmer does not have to leave his usual programming environment to deal with the details of designing and implementing an application-specific protocol each time his program is distributed over more than one machine.

A File Server Example

A *file server* is a node that provides information storage services to clients. (Users are often called *clients* when they can be either people or programs.) File servers usually store their information in files, and clients access data through an abstract interface of operations such as *CreateFile*, *WritePage*, *CloseFile*, and so forth. Because file servers typically execute in isolated machines that are accessed remotely through a network, their operations are ideal candidates for remote procedure call.

To illustrate this further, consider a concrete instance of figure 1 where C is a client reading files and S is a file server offering its operations through remote interface M_S. Client procedure P_C now calls service procedure P_S, where P_S is the server's remote *Read* procedure, e.g.,

Read: PROCEDURE [*file*: *File*, *start*, *stop*: *BytePosition*] RETURNS [*data*: *Buffer*].

Of course, if P_C processes only one byte at a time, his call of *Read*[*file*, *position*, *position*+1] will have rather large overhead. In this case it is likely that the server, S, will provide the client with

some *local* procedures that execute in C and access the server as necessary. For example, the following local *ReadByte* routine could maintain a buffer in C and call *Read* remotely in S as necessary to refresh it:

ReadByte: PROCEDURE [*file*: *File*, *position*: *BytePosition*] RETURNS [*byte*: BYTE].

It is important to observe that the client application on C need not know where *Read* and *ReadByte* reside. The implementors of the file server declare these operations to be local or remote depending on their own policies for providing reasonable service to the client. Implementors can change these policies at any time without programming impact on the client because the client's abstract file interface is the same whether all, some, or none of the interface procedures are remote.

Benefits of Remote Procedures

Remote procedure call has many desirable properties. One way to characterize these properties divides remote procedure applications into three classes: resource sharing, or accessing a common resource such as the file server above; load splitting, or partitioning work for space or time advantages; and conversation, or distributed interaction among closely coupled processes. The boundaries between these classes are elastic and many RPC applications fit naturally into more than one. The dissertation presents several examples of each.

An Abstract Machine RPC Primitive

The Transfer paper's description of local program control is important because it defines a low-level *Transfer* primitive, where *Transfer* is an abstract machine operation that is general enough to express a rich collection of synchronous control disciplines. At the programming language level—as the paper discusses at length—this collection usually includes at least procedure calls, coroutine transfers, and exception handlers. In searching for a *remote* transfer primitive, then, the desire for language-level uniformity requires looking beyond the intuitive but slightly misnamed notion of remote *procedure* call for an abstract machine operation at least as powerful as *Transfer*. Fortunately, *Transfer* is flexible enough to extend naturally into the distributed domain.

Define a *context* to be a program with some local storage (including PC) and a binding rule mapping the program's names into storage addresses. Let a *port* be the name of a context. Then as defined in the Transfer paper, the *Transfer* operation

Transfer (*destinationInport*, *returnOutport*, *argumentPtr*)

specifies that execution of the current context is suspended and that control, *returnOutport*, and *argumentPtr* are delivered to the context named by *destinationInport*. This new context begins executing at its previously suspended PC and retrieves arguments using *argumentPtr*. When the new context finishes its work, it (usually) initiates another *Transfer* of control to the *returnOutport* context with a new *argumentPtr*. The thesis gives an example of how *Transfer* is used to implement procedure call.

Remote Transfer

Extending *Transfer* for remote use is straightforward. *RTransfer* is defined by making these simple changes to *Transfer*'s arguments.

> *Inport* and *Outport* must now identify contexts in the distributed environment rather than just within one node. This identification is extended by prefixing each existing context's name with the name of its host node. This presumes that contexts do not migrate, which is a continuing assumption here. (If contexts do migrate, adding a level of indirection to port names remedies the problem.)
>
> *ArgumentPtr* receives the same treatment. Because it is a pointer, however, dereferencing it in the destination address space does not work. This problem is overcome by insisting that remote *Transfer* handle *argumentPtr* by passing *argumentPtr*↑—an argument record, not a pointer to one—as a value parameter, therefore copying it. The Transfer paper suggests using this technique on short argument records; remote *Transfer* must always use it.

This definition of *RTransfer* is readily implemented with messages. First, denote the node and context parts of a *port* by *port.node* and *port.context* and rename *argumentPtr*↑ to *argumentRec* for clarity. Then

RTransfer (*destinationInport, returnOutport, argumentRec*)

sends the message (*destinationInport, returnOutport, argumentRec*) to *destinationInport.node*, where *destinationInport.node* delivers control to *destinationInport.context* with *argumentRec*. The *returnOutport* always remains a fully qualified name because it can specify a return to any node at all, not necessarily to *RTransfer's* origin.

A Language Translator RPC Primitive

The *RTransfer* primitive is a concise and appealing abstract operation for implementing remote transfers. The Mesa programming system actually uses the local version—*Transfer*—to implement control transfers among programs, procedures, coroutines, and exceptions. . But there are other languages where *Transfer* is not available as an abstract machine operation, and indeed where the compiler generates a different sequence of instructions for each control transfer—typically, for only procedure call. Integrating an RPC primitive into these *Transferless* languages is not as easy as it is for languages that use *Transfer*. Consider two possibilities:

> *Compiled RTransfers.* Implement *RTransfer* directly with compiled remote transfer instruction sequences. This has the advantage of efficiency and transparency, but may not be feasible if the compiler cannot be changed.
>
> *Source-level stubs.* A separate source-level RPC *translator* can augment the compiler by generating *stub* routines that implement *RTransfer* in the language itself. This solution can be inconvenient for application programmers, but it requires no compiler changes.

In the stub approach, the translator generates two source-level stubs for each remote procedure—one for the client and one for the server. This is illustrated in the setting of the previous file server example in figure 2. In the figure, the client's remote call to *FileOps.ReadPage*

is automatically intercepted by the local *FileOps* stub interface, which has a declaration of *ReadPage* identical to the file server's real *ReadPage*.

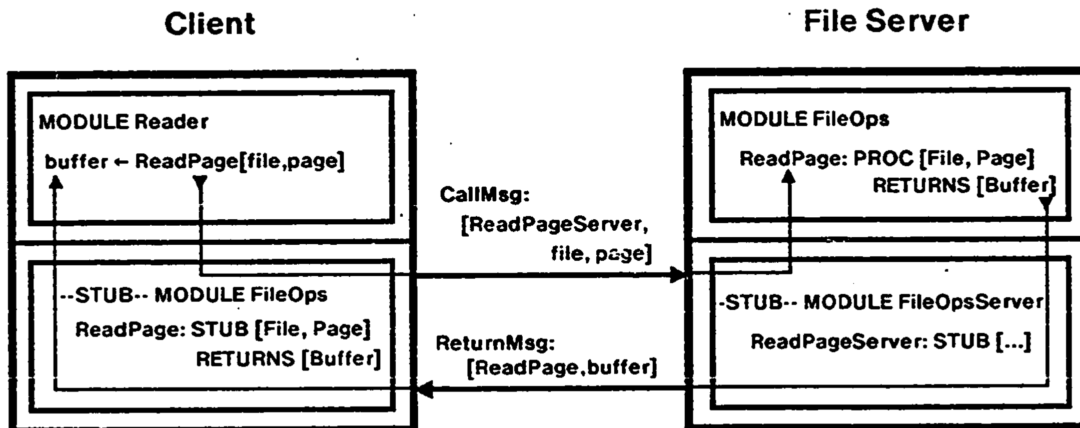


Figure 2: A stub remote procedure call from client to server.

The stub routines operate as follows: The client *ReadPage* stub packages the call into a *CallMsg* and sends it to the server's *ReadPageServer* stub. *ReadPageServer* unpackages the message and calls the server's actual implementation of *ReadPage*, which does not know whether the call is local or remote. When the real *ReadPage* procedure finally returns its *Buffer*, the server stub packages the buffer in a *ReturnMsg* and sends it back to the client. After the client stub receives *ReturnMsg*, it unpackages the message and returns the buffer to the original invocation of *ReadPage*. Because the call semantics are uniform, the client then proceeds as though the call had been local all along.

Notice a crucial property: The client and server programs—i.e., client module *Reader* and server module *FileOps*—do not change for remote use. The code in these modules is the same whether it is invoked locally or remotely because the physical location of the procedure implementations is hidden from programmers behind the *FileOps* interface. All the RPC details are in the stubs, which the stub translator creates automatically from a scan of the procedure declarations.

The stub approach to language-level RPC is typically less efficient than the *RTransfer* or compiler approaches because it is not as well integrated into the language runtime environment. But, in trading the efficiency of the latter methods for the translator's stub approach, procedure-calling uniformity is still preserved: users write the same code for local and remote use, and no program changes are necessary.

A Crash and Failure Model

The words *crash* and *failure* are used interchangeably in this dissertation. Their meaning is complicated in distributed systems by the presence of multiple machines and the unreliable communication between them—links of networks and gateways.

Node crashes can be classified in two ways.

Machine failure is the complete halt of a machine and all of its processes because of a hardware failure or because of a software error that affects all processes (e.g., operating system "crash"). Furthermore, in some environments the degraded response of a crippled machine can be tantamount to machine failure.

Process failure is the halt of one process because of a process-specific hardware problem (e.g., parity error) or because of a process-specific software error.

These definitions of machine and process crash—or failure—apply to both conventional and distributed systems. Serious communication failures, on the other hand, are most often a problem of distributed systems. For instance, if procedure-calling primitives break on a single machine it is hard to imagine that the processor has not broken, crashing all processes. But in a distributed system a broken link does not imply a broken processor, nor does the isolation of a process imply that it has failed.

Before taking a deeper look into this, three definitions of communication-related failure are needed.

Communication loss is the destruction of information (packets) because of unreliable transmission due to network or gateway errors. The duration of a communication loss can span many orders of magnitude, as the next two definitions discuss.

Communication outage is the loss of a small amount of information—say, a few packets—because of network or gateway errors. Communication outages are transient errors analogous to correctable memory parity errors and are from milliseconds to minutes in length. If the end-to-end communication strategy is providing reliable transmission, then short outages (milliseconds to seconds) are usually masked from clients and no information is lost. As outages increase in length or frequency, however, they become more serious.

Communication breakdown or *partitioning* is when no information at all can be routed to the destination because of network breaks or gateway failures. Communication breakdowns are hard failures analogous to hardware and software crashes and are from seconds to weeks in length. A reliable transport mechanism generally will not mask breakdowns from clients and will inform them when the duration of a partitioning is minutes or longer. Deciding when an outage should be declared a breakdown is usually system or application dependent. The choice is roughly analogous to the machine failure decision when a crippled processor has substandard performance.

Handling communication breakdown is conceptually straightforward: the isolated nodes simply wait until communication is reestablished and then retransmit their waiting information. There are two difficulties with this ideal approach. The first is timing considerations. Real programs are often not willing to wait even minutes, let alone hours or days, to complete their tasks. Rather than wait for a particular remote service these programs will often try to take their business elsewhere. This is, after all, touted as an advantage of decentralized computing. The second difficulty is the nature of the failure. When communication fails it is often impossible to tell whether the failure is occurring because of a partitioning or because of a remote machine or process crash. If the remote machine has crashed then waiting will not usually aid recovery. Because of this, most distributed programs elect to treat communication breakdown exactly like machine failure.

3 Overview of the Essential Remote Procedure Issues

Behind the conceptual simplicity of the examples in the previous section are a number of thorny issues. The alert reader may have had an inkling of these issues when considering the ramifications of the extended *RTransfer* definition. For instance, increasing the naming scope of *inPorts* and *outPorts* raises two questions: What happens when *RTransfer* is unable to deliver control to an isolated or unresponsive remote context? How are *RTransfer*'s previously local context names bound in the new global name space? In addition, the addressing problem with *argumentPtr* is recursive—what happens when *argumentRec* contains pointers?

Behind these questions there are five *essential issues*. They are defined very briefly here and are discussed in more detail below.

> *Call semantics* define the abstract invocation behavior of a remote procedure mechanism.
>
> *Binding and configuration* establish the naming and configuration (interconnection) of programs communicating with RPC.
>
> *Typechecking* enforces the type compatibility of interprogram bindings.
>
> *Parameter functionality* determines the restrictions, if any, on the parameters passed by an RPC mechanism.
>
> *Concurrency control and exception handling* define the interactions between the RPC mechanism and any independent parallel-processing and exception-handling mechanisms.

Fundamental and Nonfundamental Issues

The five issues listed above must be addressed by remote procedure mechanisms whose goal is language-level local and remote transparency in a homogeneously programmed distributed system. If the requirement of language-level transparency is relaxed, the issues can be split into two groups:

> *Fundamental invocation issues.* Call semantics and concurrency control and exception handling are fundamental to any remote invocation mechanism, whether it is based on procedures, messages, or some other communication discipline. These two issues are crucial because they determine the low-level semantic behavior of an invocation primitive.
>
> *Language-level transparency issues.* Binding and configuration, typechecking, and parameter functionality are vital for the *language-level* transparency of local and remote procedures. In this dissertation, these transparency issues merit the same attention as the fundamental invocation issues. This is because the goal of overall local and remote transparency requires defining more than the low-level semantic behavior addressed by the fundamental issues: To obtain transparent high-level semantic behavior, the language-level issues must be considered along with the more fundamental ones. Thus there are five essential issues in all.

In the thesis, the language-level issues of remote procedures are resolved in precisely the same fashion as the language-level issues of local procedures. Thus in figure 1, the example remote call is intentionally written to appear like a local call even though the two implementations are worlds apart. Again, making these local and remote calls *transparent* to the programmer is a fundamental requirement of this thesis' RPC implementations.

The five essential issues are now discussed in more detail.

Call Semantics

Call semantics are the most critical behavioral issue of remote procedure call. There are two cases to consider: RPC in the absence of crashes (the normal situation) and RPC in the presence of crashes (a rare situation caused by a local or remote process failure or a communication breakdown). Meeting the goal of local and remote transparency requires that local calls and remote calls have the same semantics: otherwise, programmers have to adapt their code to both. This goal is clarified by considering normal call semantics (no crashes) and abnormal call semantics (crashes) separately.

Exactly-once Semantics

In the absence of crashes, meeting the goal is straightforward. Local procedure call is characterized by *exactly-once* semantics: the caller transfers arguments and control to the callee, waits for the procedure computation to occur *exactly once*, and resumes execution when the callee returns results. Remote procedure call easily achieves these exactly-once semantics by demanding that *RTransfer* send and receive its control-passing messages reliably. Reliable transmission ensures that each *RTransfer* message is exchanged exactly once, as desired.

Last-one Semantics

In the presence of crashes, obtaining transparency is complicated. First of all, local calls no longer have exactly-once semantics. If procedure *P* performs a local call to procedure *Q*, their host machine can crash any time during the transfer to *Q*, during the execution of *Q*, or during the transfer back to *P*. A crash at any of these times violates exactly-once semantics because the call does not complete. If the machine is booted and its crashed programs are restarted, either from checkpoints or from scratch, then the call from *P* to *Q* will be repeated. In the face of crashes, this repetition will continue until the call finally completes and the program either terminates or reaches another checkpoint. In the presence of crashes, then, the call results and side effects that the program actually uses are those of the very *last call* that executes. The results of intermediate executions—partial or total—are abandoned, although the side effects of intermediate calls can potentially influence the last one. We will say that local calls have *last-one* semantics in the presence of machine crashes.

Crashes in distributed systems create new semantic difficulties because a remote procedure call is between two independent parties, and a crash of one party usually leaves the other one running. In addition, this qualitative semantic difference is often quantitatively magnified by failure-prone networks that make loosely coupled remote calls much more crash-prone than tightly coupled local calls. For example, a remote call can fail when the machine executing the callee crashes (leaving the caller running), when the machine executing the caller crashes (leaving the callee running), or when the transport mechanism partitions sometime after the *call* message is sent but before the

return is received. The caller and callee of a local call almost never have this problem when their *machine* crashes because both caller and callee are usually swept away. It can happen, however, during a local interprocess call when one of the two *processes* fails, and this is especially true in multiprocessors. This type of crash, when one of the communicating parties remains running, is considered below.

It is important to observe that the problems of failure, violated exactly-once semantics, and crash recovery are *not* unique to distributed systems. They are just exacerbated in autonomous, loosely coupled distributed environments.

Orphans and Extermination

In a single-machine environment, if a given procedure call is repeated after a crash recovery we know that *all outstanding activity of the call has ceased.* This is true because all processes are recreated (either from a checkpoint or by a boot) and therefore all outstanding activity in any old processes is terminated. In fact, operating systems usually extend this all-activity-ceased property throughout their domain by *resetting* I/O channels, disk controllers, and other semi-autonomous devices which can influence the processor's memory and state. Thus, in the presence of crashes, single processor exactly-once local procedure semantics degenerate into last-one semantics.

Now consider a machine crash in a distributed environment. To provide transparent last-one semantics in the presence of crashes, all outstanding remote activity must cease before restarting. Conceptually, a distributed *reset* must be issued to all the threads of control that the crashed machine has left outstanding on other machines. In a distributed system, the remote calls that continue to be executed by a crash's dangling threads of control are called *orphans*. For example, in figure 3, assume that machine *A* calls machine B_1, and that B_1 in turn calls *C* for *A*. Now consider what happens when B_1 crashes and leaves an orphaned call executing on *C*.

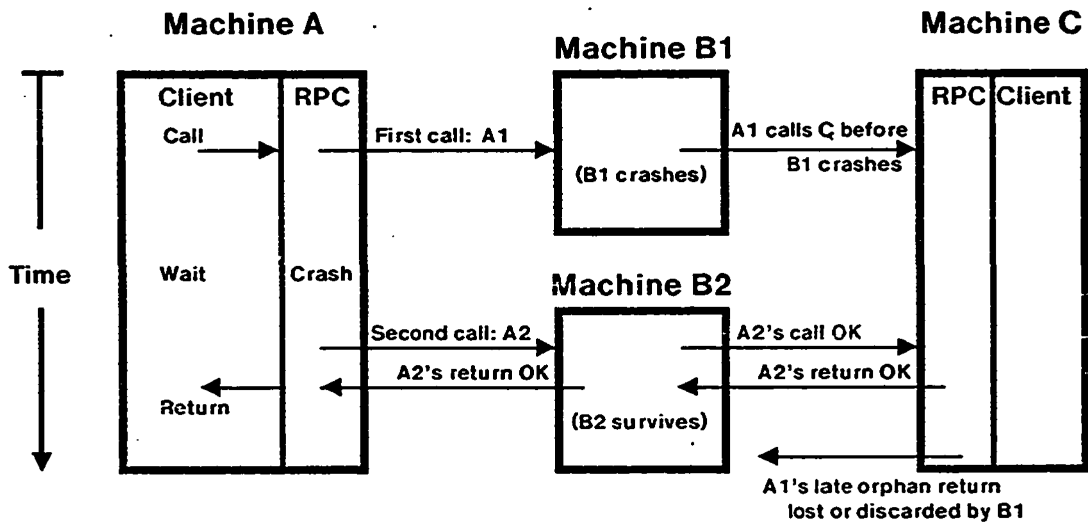


Figure 3: An orphaned remote procedure call that violates last-one semantics.

If A decides to retry its call through another machine, B_2, and B_2 calls C, C can return results to B_2 and thus to A *before* B_1's outstanding calls on C die out. The orphaned call from B_1 to C causes a violation of last-one semantics because it is possible for it to execute after A (incorrectly) decides that its original call to B_1 is dead. Furthermore, it is not necessary that B_2 be a different machine: it could just as well be B_1 after B_1 completes crash recovery.

This simple example shows that to achieve last-one semantics in a distributed system crash recovery must ensure that *all orphaned calls to other nodes are exterminated.* This guarantee must be met while the crashed machine is in recovery—*before* it can repeat any of its calls that would violate last-once semantics. The process of killing all of a node's orphans is called *extermination.*

One obvious and unacceptable method of exterminating orphans is to issue a master *reset* that brings down the entire distributed system when any machine crashes. This is, of course, unacceptable behavior, but it is just what many conventional systems do when a vital process fails—restart everything.

A second solution is to crash just those machines containing orphans. If this improvement still appears extreme, consider a further refinement that exterminates just those processes that are executing orphaned procedure calls. This seems ideal and very similar to the single-machine case. The difficulty with either of these two schemes, however, is tracking down all of the orphaned processes—in effect, determining a crashed machine's domain of influence by following the call stack from machine to machine. This becomes hard when these orphaned calls are *themselves* executing on crashed processors, making stack-following impossible.

Three methods for exterminating orphans are presented in chapter 5 of the thesis. The first scheme is a straightforward extermination, as described above. The second scheme, expiration, uses system-wide synchronized clocks to establish expiration deadlines on remote calls. The third scheme, reincarnation, is similar to expiration but does not require system-wide clocks.

Binding and Configuration

A language-level RPC scheme must give programmers convenient mechanisms for declaring distributed programs, or for specifying remote procedure bindings and module configurations. In particular, the assignment of program modules to nodes in a distributed environment should *not* require programming. There must be a higher-order scheme, such as Mesa's configuration language (for single machines), which handles the details of module assignment and intermodule procedure binding, linking, and loading. Reconfiguration of the modules and machines of a distributed program should be easily specified in this configuration language and require no lower-level changes to the programs themselves.

Typechecking

For remote procedures to be as typesafe as local procedures, the underlying programming environment must guarantee that whatever type calculus is enforced on local machines is extended completely into the network environment as well. In figure 1, this means that the types of x and y must conform to those of r and s, and similarly for t and z.

For weakly typed languages this guarantee will often simplify into (for example) a requirement that t and z are both integers. This simplification may even extend to type translation between heterogeneous environments of weakly typed languages like Bcpl and Lisp, where Bcpl integers may inherently conform to Lisp integers. For strongly typed languages, on the other hand, the guarantee may be much more strict. Mesa, for instance, requires that if t is a RED INTEGER then z is a RED INTEGER also.

Parameter Functionality

Permitting the parameters (arguments and results) of remote procedures to have a full range of types (i.e., be as general as possible) is as important as typechecking. For languages with only simple scalar and static array types this is easy. For variable length structures like strings, dynamic arrays, and variant records the implementation is still straightforward, typically requiring extra dynamic allocation—in the callee for arguments, and in the caller for results. The real problems occur when parameters have implicit or explicit pointers: because the caller and callee have distinct address spaces, pointers valid in one machine are not usually meaningful in the other. These problems, including those of procedure parameters, are addressed in chapter 4 of the thesis.

Parameter Serialization

The process of packing a parameter record into a *call* or *return* message is called *serialization*. Similarly, the inverse process of unpacking the message is called *deserialization*. The word *serialization* is derived from the observation that packets (messages) are usually transmitted serially over networks.

As noted above, serialization is complicated by parameters with pointers because a pointer's referent, not the pointer itself, is usually what must be transmitted. A parameter record containing pointers is therefore a tree with the pointers as branches and the referents as leaves. The job of serialization is to traverse this tree and flatten it, packing all of the leaves into the parameter portion of *call* messages (for arguments) or *return* messages (for results). Deserialization performs the inverse operation, reconstructing the tree from the flattened representation.

Concurrency Control and Exception Handling

The goal of transparency demands that remote calls execute synchronously, just as local calls do. Because a remote call (usually) involves a process switch, this implies that the caller automatically blocks and waits until the remote callee returns—just as local calls do without the process switch.

For parallel execution of any activities—local or remote—the programmer must therefore use whatever concurrency tools his language provides and not expect the RPC implementation to offer hidden concurrency.

The desire for transparency also requires that remote procedure calls report errors in the same fashion as local calls. This is straightforward as long as we remember that remote calls raise some exceptions that local calls never do, for instance, *NetworkPartitioned*, *CommunicationTimeout*, and *RemoteCrash*. The handling of these exceptions will of course depend on the particular call, but their *reporting* must use the same mechanisms available to local calls—not any new ones.

4 Summary of Ideal Remote Procedure Properties

The previous section discusses a number of remote procedure issues. These essential issues deal directly with the problem of providing transparent syntax and semantics in a homogeneous language. There are also a number of *pleasant issues* that address other problems such as efficiency, autonomy, and heterogeneity. Each essential and pleasant issue is now resolved into a brief statement of ideal behavior. These statements are one set of *essential* and *pleasant properties* for transparent RPC mechanisms.

Essential Properties

These five properties are essential to a remote procedure mechanism that is fully integrated into a homogeneous programming language and that provides transparent local and remote procedure semantics.

> *Uniform call semantics.* In the absence of crashes, remote procedures must have the exactly-once semantics of local procedures. In the presence of crashes, remote procedures must have the last-one semantics of crashing local procedures. Atomic procedure call is too expensive to be the standard RPC mechanism; the cheapest uniform semantics is obtained by automatically exterminating orphans after crashes. Remote calls must be invoked via concurrent invocation, not serial invocation.
>
> *Powerful binding and configuration.* Programming language facilities for binding—i.e., specifying, linking, and loading—configurations of separately compiled modules must be extended to handle modules that reside and execute on remote machines. The binder must permit flexible machine naming, and binding facilities must be available both declaratively and dynamically.
>
> *Strong typechecking.* The type calculus of the standard programming environment must be fully applied over the distributed system. Any operation that causes a type violation in the local environment must cause the same violation in a remote environment.
>
> *Excellent parameter functionality.* Parameters to remote procedures must be passed by value-result. Nearly all language- and user-defined datatypes allowed as parameters of local procedures must be valid as parameters of remote procedures. This includes many traditional uses of pointers, but excludes list structures: while Herlihy and others show how to pass shared structures in parameters, our mechanism does not automatically handle graphs. Global variables are disallowed as well.

Standard concurrency control and exception handling. The standard parallel-processing and exception-handling facilities of the language must interact identically with local and remote procedures. This may present performance problems for languages with poor (or no) concurrency control. It may also make error-handling more troublesome for languages without good exception mechanisms, especially those without interprocess exception mechanisms.

Pleasant Properties

These six properties are not fundamental for an RPC *language* mechanism but they do make any proposed scheme much more palatable for real distributed programmers.

Good performance of remote calls. To be a realistic tool, remote procedures must be comparable in cost to the application-tuned protocols they are intended to replace. It is well known that, in the limit, efficiency-minded clients will always trade abstraction and elegance for direct manipulation of bits.

Sound remote interface design. Remote interface designers must always evaluate their work in light of the increased cost of remote procedures. While transparency is wonderful for implementors and users, the interface designer must be careful to distribute functions cost-effectively.

Atomic transactions. Robust applications demanding high reliability must use independent transaction mechanisms. This is vital because the recommended RPC semantics are not atomic in the presence of crashes.

Respect for autonomy. The binder must respect both the autonomy and the capabilities of its host machine. This requires policy decisions based on the degree of cooperation in the environment.

Type translation. The compiler and binder should cooperate to perform automatic type translations in heterogeneous language and machine environments. Runtime negotiation can optimize performance.

Remote debugging. A language-level debugger that deals with multimachine configurations is vital for real programmers. Problems of heterogeneity and autonomy add complexity.

5 Design Approaches for a Transparent Mechanism: Emissary

Emissary is an RPC mechanism that addresses the essential issues. Emissary's design is composed of three distinct parts: orphan algorithms, remote call mechanisms, and distributed binding. The design of these components is chapter 5—fully one-third of the thesis. Here is a ridiculously brief description of how the parts combine to satisfy the five essential properties. Details must remain in the dissertation itself.

Orphan algorithms. To obtain semantic transparency during crashes, the uniform call semantics property demands that orphans be exterminated. Three orphan extermination algorithms are presented: extermination, expiration, and reincarnation. The costs, requirements, complications, tradeoffs, and reliability of each method are discussed.

Remote call mechanisms. Once a distributed program has been loaded and started on all of its nodes, it executes normal, *steady-state* remote calls. To achieve local and remote

transparency for steady-state calls, the calls must satisfy the uniform call semantics, typechecking, parameter functionality, and concurrency and exception control properties. The compiler (or stub translator) and RPC runtime environment bear most of these responsibilities. A complete Mesa program is given for Emissary's call mechanism.

Distributed binding. Before a distributed program can perform steady-state calls, it must undergo a binding *transient* where modules are configured, bound, loaded, and started. Transparency requires that remote binding schemes satisfy the strong typechecking and powerful binding properties. Careful extensions to local binders can transform them into distributed binders that meet this requirement, and such extensions for the C/Mesa binder and configuration language are described and illustrated with examples.

The orphan algorithms and distributed binding scheme, while described in detail, are unimplemented. The Emissary RPC mechanism is also unimplemented, but is fully specified and is based on the implementation and evaluation of several operation RPC mechanisms.

6 Performance Evaluation of a Family of Mechanisms

To explore the performance characteristics of RPC mechanisms I tested five related implementations (I built three myself): Envoy-Diplomat, Stubs, Liaison, EtherPkt, and EtherPktMC. Each succeeding mechanism is much faster than its predecessor. The mechanisms are written in Mesa, communicate over the prototype Ethernet (2.94 megabits/second), and execute on Dolphin personal computers (a null Mesa procedure call on Dolphins takes 40 microseconds). The detailed characteristics of the mechanisms are presented in chapter 6 of the thesis.

The discoveries and techniques used to tune and optimize the mechanisms are a vital part of the dissertation. No single, guiding optimization principle emerges. Rather, a set of principles emerges, each of which significantly increases performance when applied appropriately. As a consequence of this, the results of the performance evaluation are presented as a group of general *performance lessons*. These lessons are incorporated into the Emissary design.

Here is a list of the nine performance lessons; the thesis discusses them in detail. They are not ordered by importance.

1. Bytestreams are bad.
2. Watch for hidden protocol costs.
3. Special-purpose protocols are good.
4. Use microcode for exceptional performance.
5. Caches are very important.
6. Serialize by compiling inline, not interpreting out-of-line.
7. Serialize large blocks, not small ones.
8. Select your data protocol carefully.
9. Avoid copying whenever possible.

A graphic illustration of the performance lessons' impact on Dolphin RPC appears in figure 4. The data are from roundtrip timings of the procedure *Null*: PROCEDURE = {NULL}. Thus only remote call overhead is measured.

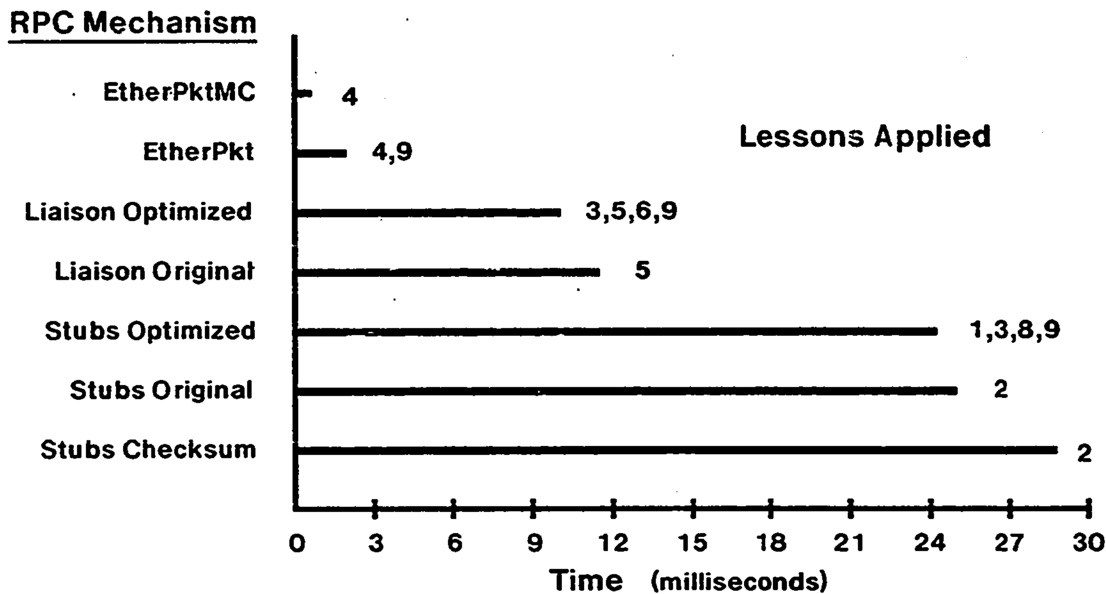


Figure 4: Performance comparison of the evaluated mechanisms.

The details of each performance step are explained in the thesis. Looking at the whole picture, it is worth emphasizing that *each step is important.* For example, the 900 microsecond decrease between Stubs Original and Stubs Optimized *appears* inconsequential when compared with its neighbor 12.9 and 3.5 millisecond decreases. But when compared against the final 800 microsecond EtherPktMC complete call time, the 900 microsecond decrease is significant indeed. To achieve a speed increase of an order of magnitude or more—in this case, a factor of 35—attention must be paid to even the smallest details.

Considered individually, the performance lessons give good but not outstanding results. Acting in concert, however, they result in a speedup of 35 times *with no significant change in functionality.* A *quantitative* performance improvement of this magnitude makes a *qualitative* difference in the value of remote procedure call as a language-level communication primitive. For instance, based on the Stubs performance numbers, assume that an existing RPC mechanism has a 25-millisecond remote call overhead. If a distributed system designer is willing to pay 10% for communication costs, then the remote operations of these slow calls must take 250 milliseconds. This is a long time, longer than the time needed to use even a very slow disk. If a remote call takes 1 millisecond, on the other hand, then 10-millisecond operations are feasible. In this case, remote operations that involve only computation and no disk activity are attractive. *Emissary's exceptionally large quantitative performance improvement makes a pronounced qualitative difference in the way that remote procedures can be used.*

7 Conclusion

The thesis of this dissertation is that remote procedure call is a satisfactory and efficient programming language primitive for constructing distributed systems. Three goals were established in the introduction to demonstrate this thesis. The dissertation meets these goals as follows.

> *Desirability.* Chapter 2 divides remote procedure applications into three classes—resource sharing, load splitting, and conversation. The utility of language-level RPC in each class is shown with specific examples. All of the examples can be programmed with message primitives as well as with remote procedures: there is no claim that RPC is a panacea for distributed programming. Instead, RPC emerges as one natural way to write distributed programs in procedural languages. Chapter 3 continues the general discussion of desirability by examining the strengths—and weaknesses—of some existing RPC schemes.
>
> *Transparency—theory.* Chapter 4 explores the semantic and syntactic ramifications of transparency in great detail. For homogeneous language systems, there are five *essential properties* that must be satisfied by any RPC mechanism that is fully integrated into a programming language. These five properties are uniform call semantics, powerful binding and configuration, strong typechecking, excellent parameter functionality, and standard concurrency control and exception handling. In addition to the essential properties, there are six *pleasant properties* that ease the work of constructing real distributed systems. The pleasant properties are good performance, sound remote interface design, atomic transactions, respect for autonomy, type translation, and remote debugging.
>
> *Transparency—practice.* Chapter 5 uses the semantic groundwork of chapter 4 to develop *Emissary*, a Mesa-based remote procedure mechanism that satisfies all five essential properties. The Emissary design has three major components: orphan algorithms to handle crashes, call mechanisms to perform steady-state calls, and a distributed binder to link the modules of distributed programs. Although unimplemented, the heart of Emissary—the call machinery presented in the Emissary algorithm—is based on the actual implementation and testing of a series of operational RPC mechanisms.
>
> *Efficiency.* Chapter 6 contains a performance evaluation of five working RPC mechanisms, three of which I implemented: Envoy-Diplomat, Stubs, Liaison, EtherPkt, and EtherPktMC. The results of the evaluation are a set of general *performance lessons* that decrease the roundtrip time for a remote call by an amazing factor of 35. These lessons are incorporated into the Emissary design of chapter 5; Emissary therefore satisfies the good performance property in addition to the essential properties.

The upshot of meeting the desirability, transparency, and efficiency goals is this: Remote procedures can and should be routinely used in applications where they have been previously regarded as an extravagant luxury. Transparent RPC is appropriate for constructing distributed systems because it is a good model for many distributed computations, is comfortable and familiar to programmers, and has excellent performance when properly implemented.