Remote Evaluation

by

James William Stamos

January, 1986

This research was supported in part by the Defense Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under Contract No. N00014-83-K-0125, and in part by IBM under the IBM Graduate Fellowship Program.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

Remote Evaluation

by

James William Stamos

Submitted to the
Department of Electrical Engineering and Computer Science
on January 10, 1986 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Abstract

A new technique for computer to computer communication is presented that can increase the generality and performance of distributed systems. This technique, called Remote Evaluation, lets one computer send another computer a request in the form of a program. A computer that receives such a request executes the program in the request and returns the results to the sending computer.

Remote evaluation provides a new degree of flexibility in the design of distributed systems. In present distributed systems that use Remote Procedure Calls, server computers are designed to offer a fixed set of services. In a system that uses remote evaluation, server computers are more properly viewed as programmable soft abstractions. One consequence of this flexibility is that remote evaluation can reduce the amount of communication that is required to accomplish a given task.

Our thesis is that it is possible to design a remote evaluation system that permits the processing of a program to be distributed among remote computers without changing the program's semantics. In support of this thesis our proposal for remote evaluation uses the same argument passing semantics for local and remote procedure invocations (call by sharing); it provides atomic transactions to mask computer and communication failures; and it provides a static checking framework that identifies procedures that can not be relocated from computer to computer.

We discuss both the semantics of remote evaluation and our experience with a prototype implementation. The idea of a remote data type is introduced to let one computer name objects at a remote computer. A detailed discussion of the compile-time and run-time support necessary for remote evaluation is provided, along with a detailed sample application.

Thesis Supervisor: David K. Gifford
Title: Assistant Professor of Electrical Engineering and Computer Science

Key Words and Phrases: atomic transactions, call by sharing, call by value-overwrite, computer networks, distributed computing, evaluation, global names, message passing, procedure call, programming languages, remote evaluation, remote procedure call, remote types

Acknowledgments

I would like to thank my advisor, David Gifford, for his support, guidance, and friendship throughout this endeavor. My readers, Barbara Liskov and David Reed, made many invaluable suggestions that improved the structure and clarity of the thesis.

I must also thank my fellow graduate students for their technical expertise, editorial assistance, and companionship. Special thanks go to John Lucassen, Gary Leavens, Brian Coan, and Jim Restivo. Members of the research, technical, and support staffs made my stay an enjoyable one. Steve Berlin always had an answer to my questions. Joe Ricchio, Shawn Routhier, and Tyrone Sealy deserve credit for keeping the Dover up and running, while Rebecca Bisbee provided a lot of help and information.

To the regulars at the Muddy Charles Pub, Newhall's, and the weight room: thanks for being there. Finally, I thank my family for never doubting that I would finish.

Table of Contents

Table of Figures

Chapter One

Preliminaries

Distributed computing systems have become commonplace: communication networks routinely link personal computers, professional workstations, and powerful mainframes. A distributed computing system lets a system designer distribute functionality to improve performance, to increase availability, or to provide for incremental growth. Other systems are inherently distributed in that their components are geographically dispersed. Remote procedure calls may be used to construct a program that runs on several computers, but as we shall see this idea has several drawbacks. This thesis proposes an alternative to remote procedure calls that remedies some of their drawbacks.

The alternative construct for building distributed systems is remote evaluation (REV), which is the ability to evaluate an expression at a remote computer. A computer supporting remote evaluation evaluates each expression it receives and returns the results to the sender. This technique, which can simplify the design and implementation of distributed systems, can also improve performance. We have found a number of scenarios in which the ability to send an algorithm to the data is almost indispensable. Because the generality provided by REV does not have inordinate costs, we feel that some form of REV should be routinely provided in distributed computing environments. REV generalizes the idea of a remote procedure call and has a novel argument passing semantics. The thesis covers the meaning, compile-time checkability, run-time requirements, and utility of REV requests.

This chapter begins by introducing our model of a distributed system and then discusses how application programmers can build programs that use several computers. After introducing remote evaluation, we compare it to existing methods for building distributed systems. We concentrate on the semantics and cost of these methods and describe how they affect the generality and performance of distributed systems. We also discuss systems that transfer executable code between protection domains on the same or different computers. Finally, we outline the contents of the thesis.

1.1 Distributed System Design

We view a distributed system as a collection of computers linked by a communication system. For the purpose of this thesis, we will define a *node* to be a virtual processor with memory. A node resides on a single computer and consists of an address space of processes and *objects*. An object is an instance of an abstract data type. Although a single computer can simultaneously support several

nodes, nodes do not share memory and communicate only by sending messages. Thus we are concerned with *loosely-coupled* distributed systems. We assume programmers use an imperative programming language with abstract data types and strong type checking. In this thesis, *code* will mean a sequence of instructions. For the most part, we will not be concerned with the representation of code.

The conventional, abstraction-based methodology for building large, centralized software systems [26] has been used in distributed systems, but as explained below performance considerations can restrict its applicability. We will argue that REV makes it easier to apply this methodology to distributed systems. In this methodology, a program is developed by decomposing the problem at hand, envisioning subsidiary abstractions that solve the subproblems, and then using the abstractions to solve the original problem. The same approach is applied to each subproblem, and the process continues until all the abstractions have been implemented or exist in the programming language. The methodology relies on the following software engineering principles, which control the complexity of a system:

1. *Information Hiding:* Distinguish the *specification* (i.e., what something does) from its *implementation* (i.e., how it is accomplished). Release the specification, but keep the implementation private.

2. *Generality:* When implementing an operation, make it as independent of the intended application as possible.

These principles promote the reuse of code and therefore can enhance programmer productivity. They are captured by the notion of an abstract data type.

Designers of distributed systems regularly use information hiding, because it reduces complexity without significantly affecting performance. Because internode communication is costly compared to the overhead of a local procedure call, designers of distributed systems try to minimize the number of times one node communicates with another node. Thus performance considerations usually limit the generality in distributed systems. REV will decouple generality from performance considerations and let designers of distributed systems use conventional software engineering methodology.

In a distributed system built with this methodology, each node exports (i.e., makes available to other nodes) a fixed set of general operations. When writing code that calls these operations, an application programmer uses their specification but not their implementation. Exporting general operations lets nodes accommodate unanticipated applications and styles of use, which is important when there is a large user community with diverse needs and expectations. Some remote nodes, such as array processors, displays, printers, and sensors, may be viewed as programmable devices. Other remote nodes, such as databases and file systems, are information repositories. Both kinds of nodes are likely to be used differently by different applications, which means that application programmers want nodes to export general operations.

Our goal is to simplify the construction of distributed applications that need both generality and good performance. Given the above model of a distributed system, we must answer several questions:

1. In a single request, how many operations can one node invoke at another node?
2. What does it mean to send arguments and results between nodes?
3. What happens when nodes crash and communication links fail?

These questions have been answered differently by programming language designers, systems programmers, and database designers. Section 1.3 reviews their choices and the reasons why the choices were made. Before we discuss these alternatives, we introduce remote evaluation.

1.2 Remote Evaluation

Our goal is to give the application programmer fine-grained control over the location of processing in a distributed application. We meet this goal with REV, which is the ability to evaluate an expression at a remote node.

In an REV request, the application programmer specifies a program fragment and a remote node that will execute the program fragment. The identity of the remote node may not be known until run time. As explained in this thesis, the compiler enforces strong type checking and ensures that every operation executed by the REV request at the remote node either exists at the node or accompanies the request.

We illustrate how REV works with an example. Consider an REV request that relocates the execution of a procedure. Assume that this procedure has no free variables and that every procedure called by this procedure is exported by the remote node. Figure 1-1 outlines the processing behind this simple kind of REV request but ignores time-outs, retransmissions, and the suppression of duplicate requests. The *client*, which is the node that invokes the REV request, evaluates the expression that designates the *service*, which is the node that will execute the REV request. After determining the service, the client evaluates the procedure's arguments and creates a message containing the procedure and its arguments. Then the client sends this request message to the service. When the service receives the message, it extracts the procedure and arguments from the message. The service then uses an interpreter to evaluate the procedure with the arguments. The results of the evaluation, if any, are placed in a message and sent to the client. When the client receives the reply message, it extracts the results, which are the value of the REV request. In short, the REV request causes the service to execute a procedure that the client would execute if there were no REV request.

REV requests can *nest*; i.e., one REV request can contain another REV request. The ability to nest REV requests supports modularity, because a programmer can relocate the execution of a procedure without worrying whether the procedure itself contains REV requests.

Nested REV requests are useful when an application program deals with several remote nodes.

```
          CLIENT                                SERVICE

insert procedure into message                    .
insert arguments into message                    .
send the request message                         .
wait for a reply message                         .
              .                      receive a request message
              .                      extract procedure from message
              .                      extract arguments from message
              .                      evaluate procedure with arguments
              .                      insert results into message
              .                      send the reply message
receive the reply message                        .
extract results from message                     .
return the results                               .
```

Figure 1-1: A simple view of remote evaluation (REV).

Consider a distributed mail system that contains registry nodes and maildrop nodes. A maildrop contains mailboxes for a subset of the mail system users, while a registry maintains the mapping from users to maildrops. Individual mailboxes are not replicated on different maildrops. Assume a programmer wants to perform some function on the mailbox of many other users, say to send each a customized version of a form letter. The programmer could use REV to have a registry iterate over the recipient list. The REV request would determine the maildrop for each recipient's mailbox and use a nested REV request to customize the form letter at the appropriate maildrop. A more complicated program might have better performance, especially if the form letter is short and the recipient list is large. The complicated program would use an REV request at the registry to partition recipients according to their maildrop. For each maildrop, the program would use a nested REV request to customize the form letter for the subset of recipients whose mailbox exists at the maildrop.

We suggest the following programming methodology, which uses REV in different ways at different times. When first writing a program, the programmer concentrates on making the program correct without being too concerned about performance. The programmer uses REV to relocate the execution of those program fragments that the client can not or should not execute. For example, if the client does not implement the procedures called by a program fragment, it can not execute the fragment. If the net result of executing a program fragment depends on the node that executes the fragment, the specifications for the program may require that the program fragment be executed by some node other than the client. The REV requests that relocate the execution of these program fragments will in general change the *semantics* of the program. The semantics of a program is its visible behavior, including any results computed by the program and any visible side effects it causes. We do not include resource consumption, such as processor time, memory requirements, and network traffic, in the definition of program semantics.

Once the program is debugged, the programmer may use REV in an attempt to improve performance or reduce communication. If REV requests introduced at this time do not affect program semantics, they will not introduce any bugs. In the thesis we explain how the compiler uses a conservative algorithm to decide whether relocating processing with each REV request changes program semantics. The programmer is free to heed or ignore the compiler's findings.

We can summarize REV by answering the three questions posed in the preceding section:

1. *In a single request, how many operations can one node invoke at another node?* We achieve generality without degrading performance by letting a single request invoke several operations.

2. *What does it mean to send arguments and results between nodes?* We achieve uniformity by defining the argument passing semantics for REV requests to be the argument passing semantics for local procedure calls. Because we will use the programming language CLU [24], the semantics will be *call by sharing*, which we describe in the next chapter.

3. *What happens when nodes crash and communication links fail?* We mask such failures from the application programmer by using atomic transactions.

The next section, which discusses related work, will show how our first two answers significantly depart from current remote procedure call mechanisms. Our treatment of failures, however, is similar to that found in Argus [25], which we also discuss.

1.3 Related Work

Ideas from both single-site and distributed systems are relevant to answering the three questions posed above. We explain below how remote procedure calls provide several advantages but suffer from a tradeoff between generality and good performance. We then discuss a technique from an early operating system that lets an application transmit a procedure to the operating system kernel, which then carefully executes the procedure. Because this technique of passing a procedure between protection domains is applicable to distributed systems, we survey work in this area. In particular, researchers involved with multiprocessors, functional programming languages, and distributed systems have started to consider sending executable procedures between nodes. Finally, we briefly examine remote and distributed databases, since efficient query processing in such databases requires the ability to invoke multiple operations in a single request to a remote node.

1.3.1 Remote Procedure Call

The idea of a remote procedure call (RPC) has simplified the design and implementation of distributed systems. A *remote procedure call* occurs when one node (the *client*) uses a different node (the *service*) to execute a procedure stored at that node. This procedural interface facilitates high-level

communication between heterogeneous nodes: RPC's let different nodes running different operating systems and supporting different programming languages communicate easily. A good RPC mechanism hides communication details, provides powerful but clean semantics, enforces strong type checking, and runs with acceptable performance. Neither the programmer implementing a remote procedure nor the application programmer that invokes the remote procedure must worry about these details.

Remote procedure calls are fairly well-understood and are being used to construct real systems. The idea of using a remote procedure call for communication across a network has existed for many years. Early references to RPC were related to resource sharing in the Arpanet [46, 47]. More recently, several RPC mechanisms have been built [4, 48] or are under development [25, 36]. Nelson's thesis [34] discusses RPC at some length and contains performance measurements as well as an annotated bibliography. In the absence of node and communication failures, there is agreement on RPC semantics: *exactly-once*. Under exactly-once semantics, the service executes the call once, and the client receives the results from the service. Although exactly-once semantics are desirable, node and communication failures force us to choose an alternative semantics. In the presence of such failures, there is no firm consensus on RPC semantics. Nelson advocates *last-one* semantics; Liskov [25] advocates *at-most-once* semantics.

- Under last-one semantics, the client receives the results from the very last call that executes. Side effects from each earlier call may exist, even if the earlier call did not complete. Results from earlier calls, however, are discarded.

- Under at-most-once semantics, either the call executes exactly once and the client receives the results, or the service (effectively) never receives the request and the client is so informed. Partial and multiple executions of an RPC can not occur.

The proper semantics depends on the needs of the distributed applications and, to a lesser extent, on performance considerations. Because we want to give REV a simple semantics in the presence of failures, we will prohibit partial and multiple executions of an REV request. Our REV semantics, however, is not at-most-once semantics. The following chapter describes their differences.

There is a consensus on argument passing semantics for RPC's [25, 34]: arguments and results are passed by value. Call by value has an efficient implementation, and it keeps separate address spaces disjoint. Besides simplifying garbage collection in a distributed system, disjoint address spaces are important because of performance, autonomy, and availability considerations. A novel idea in this thesis will be how to keep separate address spaces disjoint without using call by value for REV requests.

With synchronous RPC's, a client process invoking an RPC waits until the RPC returns before continuing. A client process may invoke only one remote procedure in each request sent to a service, which means that a programmer using RPC's can not realize the advantages of REV. Our REV mechanism generalizes the idea of an RPC by allowing the transmission of code. This lets one node invoke several operations in a single request sent to another node.

RPC's simplify the task of implementing distributed systems but have several drawbacks. We address the following three drawbacks in this thesis. First, although high-performance RPC mechanisms exist [4], their performance is limited by the overhead of internode communication. Birrell and Nelson [4] measured the time to execute remote and local versions of trivial procedures that simply return their arguments. For procedures with fewer than five 16-bit arguments/results, a remote procedure was roughly 100 times as slow as a local procedure. We assume throughout the thesis that the overhead for invoking a remote procedure is much greater than the overhead for invoking a local procedure. As we shall see, this assumption is a key reason for using remote evaluation instead of RPC's.

A second RPC drawback is the tradeoff between generality and performance. When RPC's are used, the latency of the communications mechanism demands a careful system design that minimizes the expected number (and size) of messages sent between nodes. Because service designers often choose performance over generality, the procedures exported by services are usually designed for a specific application. Thus the exported procedures may not be useful to a programmer implementing a different application.

The final RPC drawback we consider is a restriction on remote procedures. In the standard RPC model, code is stationary, and the arguments must always be sent to the code. The inability to transmit code means that a remote procedure can not have a procedure as an argument. Algorithms that summarize or filter information can not be sent to a remote data repository using RPC's. Algorithms that search or plan based on the contents of the repository also can not be sent to the repository. Clients must use the techniques built-in by the repository programmers and perform the remaining processing locally.

1.3.2 Nonlocal Evaluation

The ability to pass a procedure as an argument to another procedure is a simple but powerful method that lets a programmer customize existing software. When a procedure is transmitted between protection domains, the recipient must execute the procedure with care. This section discusses such procedure transmission in the context of a single-site operating system, a distributed computing system, and a multiprocessor system that supports a functional programming language.

Gaines [13] describes an operating system that lets an application process instruct the kernel to perform a complicated action that is not built into the kernel. The kernel supports a small set of primitive operations dealing with files, integers, scheduling, interprocess communication, and so on. The application programmer combines these primitive operations into a *supervisory computer program* that the kernel executes without interruption. The kernel interprets such a program and checks the arguments carefully. By prohibiting backward jumps and giving the program only one opportunity to handle an error, the kernel avoids nonterminating programs.

A supervisory computer program is more powerful than the corresponding sequence of calls on the operating system because it executes without interruption. Thus the application programmer does not need the ability to inhibit interrupts. By granting this ability in a controlled fashion, Gaines simplifies application programming and prevents an application from monopolizing a shared computer.

This approach of using a supervisory computer program has two other advantages, both of which are related to the idea of generality. First, this approach needs only a small, stable, easily-debugged kernel. Supervisory computer programs let application programmers construct facilities normally found in a complete time-sharing system. Second, a supervisory computer program can be customized to the requirements of the application.

Gaines' idea of transmitting an executable procedure between protection domains is applicable to distributed systems and multiprocessor systems. In such systems, one node sends executable code to another node. This ability, which lets a programmer compose several remote operations into a single request, is more powerful than the notion of an RPC. We review work related to this idea below.

Gifford [14] introduced *remote form evaluation* as the method by which one processor evaluates a function at a second processor. A detailed algorithm is presented that uses connections, achieves exactly-once semantics in the absence of failures, and detects processor restarts. Gifford, however, did not address type-checking, argument passing, protection, and implementation considerations.

Burton [6] uses annotations to give the programmer control over parallelism in a distributed computing environment. The annotations declare which work may be transferred to another processor and in what form the work may be transferred. Burton remarks that transferring work to another processor is advantageous only if the computation requires more work than the transfer. Combinators are used to ensure that each transferred subexpression is self-contained. Only the required part of the environment is actually transferred.

There are three differences between Burton's efforts and ours. First, Burton considers a simple, functional language, the lambda calculus. We consider an actual programming language, which supports mutable objects and persistent state at a processor. Our language contains several interesting features, such as exceptions, iterators, and abstract types. Second, Burton is interested in capitalizing on parallelism, whereas we are primarily interested in reducing the communication overhead in an application that uses the data or peripherals at several nodes. Finally, Burton's annotations do not give the programmer control over which processor executes a relocated subexpression. There is no notion of a remote interface, and there is no way to bind to a particular processor. All processors are apparently equivalent, and some form of load balancing algorithm is assumed. In our model, the operations a processor exports, as well as its persistent state, physical location, and peripherals, distinguish one processor from another.

1.3.3 Query Processing in Remote and Distributed Databases

Efficient query processing in remote and distributed databases requires the ability to invoke multiple operations in a single request to a remote node. For concreteness, we discuss two real systems: the Datacomputer [9, 12, 29], a remote database accessible through the Arpanet; and R* [23], a distributed relational database system developed at the IBM Almaden Research Laboratory (formerly, the IBM San Jose Research Laboratory).

The Datacomputer, which was developed by the Computer Corporation of America, was a network utility that provided shared use of a trillion-bit store. The intended user of the Datacomputer was a remote program. The Datacomputer supported a high-level language that contains lists, structures, strings, integers, and bytes. Communication with the Datacomputer was through self-contained requests. Upon receipt of a request, the Datacomputer compiled the request, executed the compiled request, and then returned the results. The language supported by the Datacomputer was designed to let a programmer efficiently retrieve a subset of the data stored on the Datacomputer. This language was high-level for performance and security reasons. Bandwidth limitations required a language with a good deal of expressive power. This improved performance by reducing the size of requests. The ability to compose multiple operations into a single request, which meant that intermediate data need not be sent between nodes, also improved performance. A high-level language provided security because the compiler ensured that no hostile user programs were executed.

The Datacomputer supported a limited form of REV, because a request could contain multiple operations. It also addressed the security problem by having the compiler check each request. However, it did not deal with abstract data types, own variables, and other constructs found in a real programming language. Since there was only a single service, there was no need to support nested requests, services that exported different operations, or a remote binding mechanism.

R* [23], a distributed database manager, supports a limited form of remote evaluation by evaluating *ad hoc* queries submitted by users. It uses virtual circuits to support request-response interactions between autonomous nodes. The planning, compiling, binding, and execution of queries that span more than a single node are done in a distributed fashion [7]. R* also supports preplanned transactions, which are again processed at compile time in a distributed fashion. Each remote fragment of a preplanned transaction is permanently stored at the relevant node as an *access module*. An access module is a low-level program whose representation is similar to the P-codes used in some Pascal implementations. Each node contains an interpreter that evaluates the access modules when a distributed query is invoked. This early binding of distributed queries improves performance. By associating dependencies with compiled transactions, R* detects relevant configuration changes and dynamically recompiles invalidated transactions.

As we shall see, there are two similarities between R* and REV. First, both support distributed

computation by transmitting program (query) fragments. Upon receipt of a program fragment, a node dynamically binds the code to its own code and data. Second, both assume a collection of autonomous, cooperating nodes governed by a shared transaction mechanism.

The differences between R* and REV fall into two categories. The first major difference is the language level. R* deals with a constrained, high-level query language. Except for the ability to define new relations, the set of types is small and fixed. An application programmer or an end user declares what needs to be done and lets the query compiler decide how to accomplish the task. The language level gives the query compiler a moderate amount of freedom when it translates a query into executable code. Remote evaluation, on the contrary, deals with a general-purpose programming language that includes variables, environments, mutable abstract objects, and numerous flow control constructs. Since programs are implementations rather than specifications, an REV mechanism has little freedom to change a program without altering its semantics.

The second major difference between R* and REV concerns binding. R* assumes the existence of a catalog that describes data to be accessed, including its current location and access paths. In contrast, an REV programmer makes no assumptions about data location at compile-time and uses primitive binding mechanisms at run time.

The differences between R* and REV let R* realize several benefits. The query compiler uses the catalog to estimate the processing, communication, and I/O costs for the plans it generates. Selecting the plan with the lowest expected cost achieves automatic program partitioning. Furthermore, a node does not need to act defensively when executing a program fragment, because it created the fragment from an *acceptable*, high-level request. An REV programmer, on the other hand, must partition a program for distributed execution manually. The programmer knows much less about the relative and absolute locations of various pieces of data. A node executing an REV request must expect the worst and execute requests in a restricted protection domain.[1] R* permanently stores code fragments at the appropriate nodes for preplanned transactions. Unless services cache REV requests, an REV mechanism always sends the code with each request.

In conclusion, the language level and environmental assumptions let R* perform a specific task extremely well. Compile-time checking and optimization improve performance and place fewer requirements on the run-time execution environment at each node. The implementors and maintainers of R* provide a pleasant distributed-computing environment to application programmers and end users. REV makes fewer assumptions and is applicable in more situations. REV programmers, however, must partition their own programs.

---

[1] Later in the thesis we discuss how a strong type system, digital signatures, and a trusted compile-time request checker can eliminate this requirement.

1.3.4 Abstract Value Transmission

Because a service and its clients have disjoint address spaces, the arguments and results of an RPC (or an REV request) must be sent between nodes. Values from both built-in and user-defined data types may be sent between nodes. Transmission involves the determination of the structure of an object in the original environment, the transfer of the information, and the creation of a new structure in the receiving environment. When both environments are the same and there is only a single concrete representation for each data type, transmission is a generalization of garbage collection, compaction, and reorganization [35].

An early reference to the transmission of data is due to Morris [33]. An abstract data type is *transferable* if its operations are powerful enough to translate between the type and a new encoding based on different types. Although Morris was concerned chiefly with the completeness and expressive power of a type's operations, this capability is useful for storing, retrieving, printing, and displaying instances.

Wallis [42] describes an external representation for user-defined types that permits instances to be stored on an external medium and to appear as program literals. The string-based representation does not accommodate pointers in the data structure.

An earlier but more comprehensive machine-independent and language-independent transmission scheme is due to Atkinson [2]. A two-phase traversal handles arbitrary data structures and supports inter-machine communication and the external storage of data structures. Character strings are again the external representation.

Intermetrics' Linear Graph package [30] converts arbitrary networks of interconnected data structures to sequential text files and reconstructs networks from the resulting files. A standard text editor can create or modify the external representation, which is verbose and human readable. This verbosity, however, has contributed significantly to the inefficiency of translation between networks of data structures and text files. The Linear Graph package, which is used during the design and implementation of compilers, arose from a similar system [22] developed as part of CMU's PQCC project.

The aforementioned techniques do not distinguish between abstract objects and concrete representations. Herlihy [18, 19] addresses this issue and describes a *template* scheme in which the implementor of an abstract data type makes the type *transmissible* by writing only two procedures: **encode** and **decode**. These procedures translate the concrete representation of an instance to and from a common, external representation. Herlihy's algorithms preserve sharing, accommodate cyclic structures, and allow different concrete representations in different environments. Nevertheless, Herlihy and other researchers have not paid much attention to the performance and possible optimizations of his template scheme.

Mamrak et al. [27] solve the problem of converting between different types on different machines using different languages by adding a new layer to existing operating systems. However, details regarding the conversion method and the possibility of handling user-defined types are not given. The network is largely transparent to users and application programmers, but performance remains a problem [28].

Efficient transmission of abstract values requires a suitable space of message representations. Human-readable text strings are neither compact nor efficient. A better representation is one similar to that used in the implementation of the programming language. Herlihy's thesis [18] contains an appropriate low-level representation. Nelson [34] details one possibility that is suitable for remote procedure calls.

Researchers have paid little attention to the problems of transmitting code, environments, and closures, but an extension to Simula [5] is relevant to this discussion. Minsky [31] modified Simula to let programs run in a persistent environment that included type definitions. Instances of user-defined types could be stored in protected files. Each file held instances of a single type along with relevant type and representation information. These files also contained code to encipher and decipher instances, check the validity of a file access, and translate instances to and from the file representation. In addition, procedures implementing the type's operations could be present in the file. Storing code with external data allowed any authorized user to access and manipulate information even if the type had not been implemented in the programming environment.

The address spaces of different nodes in a distributed system are typically disjoint for reasons of performance, availability, and autonomy. Hence a natural way to communicate information between nodes is by value. The preceding techniques for transmitting abstract values all implement call by value. Stroustrup [38, 39], in contrast, suggests the alternative approach of using a shared address space. If arguments to local procedures are passed by reference, using call by reference instead of call by value for RPC's unifies the semantics of argument passing and supports reconfiguration. Depending on the type specification for a remote procedure, the mechanism for passing an argument is either call by reference or call by value. Call by reference is accomplished by using a capability that is resolved in the global address space. We will also use global capabilities to let one node refer to an object kept at another node. This thesis extends Stroustrup's approach by integrating this ability into the type system via *remote data types*.

Our REV mechanism achieves Stroustrup's uniformity by using the argument passing semantics for local procedures (call by sharing) as the argument passing semantics for REV requests. Although call by sharing is similar to call by reference, we do not implement call by sharing with global capabilities or a similar mechanism. Our implementation for call by sharing in a distributed system, which transmits abstract values between nodes, is novel and efficient. Our implementation does not use remote data types, but it can coexist with them.

1.4 REV Advantages

We designed REV to improve the performance of certain distributed applications that would otherwise be built with RPC's. The key idea is that REV is a more general mechanism than RPC's. As we explain in the thesis, this can simplify the design of distributed systems that require both generality and good performance.

When there is a sequence of operations executed by the same remote node, a programmer using REV can execute all the operations with a single request, while a programmer using RPC's must execute each operation in a separate request. The REV approach amortizes the communications overhead over all the operations executed by the remote node. When the results of one operation are used only as inputs to another operation in the message, the REV approach reduces the communication between the client and the service. We illustrate these advantages with an example that uses REV.

WFS [40] is a remote file service that provides page-level access to files. WFS exports procedures to deallocate individual pages from a file and to delete a file with no pages, but it does not export a procedure that deallocates a nonempty file. Using RPC's, deleting a file with N pages requires N + 1 requests. In contrast, a single REV request can delete the same file.

A programmer using REV can partition a program into components for local and remote execution in a variety of ways, but a programmer using only RPC's does not have this ability. In the RPC model, a program has a unique decomposition into fragments for local and remote execution. The client executes local procedures; remote procedures are executed by the appropriate service. Service programmers, who attempt to accommodate all *expected* uses of a service, implicitly force a unique partition on each application program without seeing the program. REV supports a better division of labor in the construction of distributed systems: service programmers decide the semantics of the operations they implement, and application programmers partition their programs according to performance considerations, subject only to the procedures exported by the services. Figure 1-2, which shows a spectrum of possibilities for partitioning a hypothetical program, illustrates the point that REV allows many partitionings, one of which is *the* RPC partitioning.

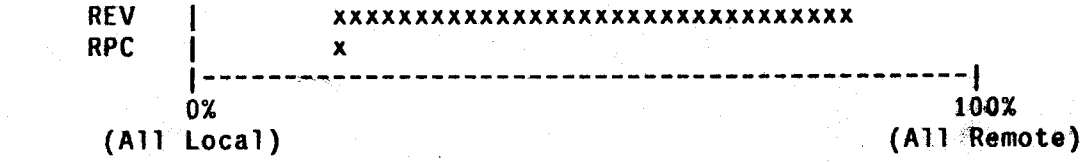


Figure 1-2: Fraction of a hypothetical program executed at remote nodes.

An REV service will export the operations built into the language and many other operations that

would never be remote procedures. For example, operations that perform arithmetic, manipulate strings, or access a record or an array, are performed so quickly by a processor that converting any of these operations into a painfully slow remote procedure is unthinkable. In contrast to the RPC model, REV lets services export such inexpensive operations without a drastic degradation in performance. The more operations a service exports, the more flexibility a programmer using REV has when partitioning a program. A programmer using REV can execute many service operations in a single REV request, but a programmer using RPC's can execute only one service operation in a remote invocation to the service.

We believe REV is a general mechanism with many advantages. First, an REV mechanism is more powerful than an RPC mechanism. An RPC is a simple REV request in which the code to be relocated is the invocation of a single remote procedure. The performance of an RPC implemented by REV should be comparable with traditional RPC performance. Thus we shall assume that our system provides REV but not RPC's.

Second, REV lets remote procedures have procedures and closures as arguments. A client using REV can easily customize a service routine that deals with a large amount of data. For example, a parameterized routine that searches, sorts, filters, summarizes, or plans can be tailored to a specific application.

Third, a service programmer could let a client using REV extend the set of remote procedures exported by a service.[2] When a client binds to a service, it could send the "new" remote procedures to the service and request that the service install them. The syntax and semantics for invoking new and old remote procedures would be identical. The service extension would be private; other clients could not access the extension.

Fourth, REV simplifies the partitioning of a program into components for local and remote execution. The programming methodology we suggested has two stages. During the first stage, the programmer uses REV to write a correct program. During the second stage, the application programmer, or perhaps someday an automatic optimizer with a cost model of distributed computation, can use REV to improve performance without changing program semantics. For example, if the programmer notices four REV requests in a row that are sent to the same service, the programmer can nest them inside a larger request that is sent to the service. Because the programmer does not want to introduce bugs while improving performance, REV requests inserted during the second stage should not change program semantics. A *location-independent* REV request relocates code whose meaning does not depend on the state of the node that executes it. We will define REV semantics so that relocating processing with a location-independent request does not change program semantics. Thus the programmer wants to insert only location-independent requests while improving

[2] There is a parallel between user-defined service interfaces and user-derived interfaces for application programs. Some researchers involved with user interface design advocate adapting the system to the user instead of vice versa [15].

performance. The thesis explains how the compiler uses a conservative algorithm to decide whether an REV request is location-independent.

Finally, REV is applicable to loosely-coupled multiprocessor systems. As long as shared, mutable data is not communicated between processors, a busy processor could use REV to offload processing. When there are no processor or communication failures, our characterization of location-independent REV requests would guarantee that program semantics were unchanged.

1.5 Thesis Overview

REV gives an application programmer fine-grained control over the location of processing in a distributed application. We provide simple rules that let the programmer or compiler decide when the relocation of processing might change program semantics. To test the ideas described in the thesis, we built a working prototype. Because the generality provided by REV does not have inordinate costs, we feel that some form of REV should be routinely provided in distributed computing environments.

The thesis does not address three important considerations. First, we do not consider the *automatic* partitioning of a program into fragments for local and remote execution [11]. The general problem has been shown to be NP-complete, but for sufficiently restricted programs a polynomial-time algorithm exists [10]. Second, we do not discuss how to build an REV mechanism with high performance. The interested reader should consult [4, 34] for performance lessons that are applicable to both RPC and REV implementations. Finally, although our REV mechanism is language independent, we do not consider the problems caused by multiple programming languages.

Many constraints influenced our REV design. Since an REV mechanism is more general than an RPC mechanism, we immediately adopted Nelson's [34] five *essential properties* and six *pleasant properties* for a remote procedure mechanism (see Figure 1-3). Sound remote interface design takes on an added meaning when REV is available, but the remaining pleasant properties are immediately applicable. There were four other constraints on our REV design:

- *Powerful Semantics:* We impose minimal constraints on REV requests to make them easy to use. All the constraints may be checked before run time.
- *Implementation Efficiency:* The REV requests inserted after a program has been debugged are meant to be optimizations.
- *Ease of Use:* REV should be simple to use and understand. Being an optimization, location-independent REV requests should yield predictable results but require little programmer effort.
- *Language Independence:* With minor changes, our design should be applicable to existing and future programming environments.

Essential Properties	Pleasant Properties
uniform call semantics	good performance of remote calls
powerful binding and configuration	sound remote interface design
strong type checking	atomic transactions
excellent parameter functionality	respect for autonomy
standard concurrency control and exception handling	type translation
	remote debugging

Figure 1-3: Nelson's requirements for an RPC mechanism.

The thesis contributes in several areas: semantics of remote invocations and their effect on program semantics; procedure and closure transmission; remote data types; and an efficient implementation for the advanced semantics we advocate. Because the bulk of an REV mechanism is an RPC mechanism, our implementation discussion must be viewed as a supplement to descriptions of high performance RPC mechanisms, such as [4, 34].

Chapter 2 defines the semantics for REV requests that relocate the invocation of a procedure. Our goal is to ensure that relocating processing with a location-independent REV request does not change program semantics, even in the presence of concurrency, node failures, and communication link failures. We use atomic transactions and an unusual argument passing semantics (call by sharing) to achieve our goal. Besides defining REV semantics, Chapter 2 incorporates REV into the programming language CLU [24]. We provide linguistic support, extend the CLU type system to accommodate REV, and characterize location-independent REV requests.

Chapter 3, which describes how to implement REV, compares an REV mechanism with an RPC mechanism by highlighting their compile-time and run-time differences. This chapter describes our novel implementation for call by sharing in a distributed system and explains how we transmitted code between nodes.

The next two chapters extend our simple model of REV. Chapter 4 extends REV so that an application programmer can send a closure to a remote node instead of a procedure. The

implementation of such requests is also described. Chapter 5 describes remote data types, which let a program running on one node refer to an object kept on another node. This chapter presents a syntax and semantics for remote data types, sketches an implementation, and evaluates their utility.

Chapter 6 presents an extended example using REV and remote data types. Chapter 7 describes our prototype REV implementation, summarizes our findings, and presents areas for future work.

Chapter Two

Semantics and Linguistic Support

The purpose of this chapter is to incorporate REV into CLU [24], a programming language with strong type checking. Our goal is to let the application programmer relocate processing with REV. In many cases this relocation of processing will not change program semantics. When the relocation does change program semantics, the compiler will inform the application programmer. We explain how: (1) transactions; (2) a nontraditional argument-passing semantics for REV (a restricted form of call by sharing); and (3) a labeling of routines exported by services let the compiler decide whether an REV request changes program semantics. To support modularity, we let REV requests nest.

In this chapter we define what an REV request means (its semantics) and how one is written (its syntax). We defer implementation issues until a later chapter.

2.1 Programming Language Support for REV

Although the idea of remote evaluation is language independent, for pedagogical reasons our discussions and examples are based on the programming language CLU [24]. The thesis assumes familiarity with CLU. Because a complete introduction to CLU is not relevant to our mission, we refer the interested reader to the CLU manual [24]. Section 2.1.1 mentions those aspects of CLU that will be relevant to REV and our examples. Our REV semantics will require a transaction mechanism, as we want to avoid partially executed requests. Therefore, Section 2.1.2 incorporates into CLU a simple model of nested transactions. Finally, Section 2.1.3 incorporates services into CLU.

2.1.1 CLU

CLU is a real programming language that supports program development according to the methodology of problem decomposition and the use of abstract data types. The type system in CLU lets a programmer define abstract data types and enforces strong type checking. A *cluster* is a module that implements an abstract data type, which is a set of objects and a set of routines to create and manipulate those objects. A programmer implementing an abstract data type chooses a *concrete representation* for the type that may use both built-in and user-defined types. A CLU *compilation environment*, which contains compiled specifications for abstract data types, permits separate compilation while retaining strong type checking across module boundaries.

Each cluster operation, such as **stack$push**, is an operation for some abstract data type. A dollar

sign ($) separates the type name from the operation name. *Equates* may be used to abbreviate constants with a lengthy textual representation. For instance, programmers would balk at writing "stack[set[int]]" every time this type appeared in a program. Including in a program the equate

```
SISets = stack[set[int]]   % stack of integer sets
```

lets the programmer write "SISets" instead.

A CLU program consists of a collection of clusters and routines, where each routine is either a procedure or an *iterator*. An iterator can be thought of as a procedure with two arguments: a collection of objects and a closure. A *closure* consists of code and an environment in which to evaluate the code. An iterator computes a sequence of objects from the collection and applies the closure to each object in the sequence.

A CLU program manipulates objects in heap storage. An object can refer to other objects, including itself. Objects have names, and one object refers to another object by containing its name. Variables in CLU refer to objects. Variables and objects, however, do not refer to variables. An object is *mutable* if its state can change; otherwise, it is *immutable*. An object exists as long as it is accessible; inaccessible objects are automatically reclaimed.

CLU uses *call by sharing* to pass arguments to procedures. In call by sharing, the caller and called routine *share* the argument objects, i.e., both refer to the object. Mutations of arguments performed by the called routine are visible to the caller. The called routine, however, is unable to modify any of the caller's variables. Call by sharing, which is different from call by value and call by reference, is similar to argument passing in LISP.

Parameterization lets procedures, iterators, and clusters define a class of related abstractions. CLU distinguishes arguments from parameters. An *argument* is an object passed to an iterator or procedure at run time. A *parameter* is a value that is computable at compile time. A programmer instantiates a parameterized module by supplying parameters of the appropriate type. For example, **array[t:type]** is a parameterized cluster. Until parameter t is supplied, the operations in the cluster can not be invoked. Using **int** for type t yields **array[int]**, an ordinary cluster that defines a set of objects (integer arrays) and a set of primitive routines to create and manipulate integer arrays. In this example, **int** is the parameter; its value is known at compile time.

CLU uses the *termination* model of exceptions. Raising an exception terminates the current activation, which can not be resumed. Each routine lists the exceptions it raises in its **signals** clause. Exceptions do not propagate automatically across procedure boundaries. Unhandled exceptions become **failure** exceptions. The **failure** exception, which has a single argument of type **string**, is a special exception. Since every procedure can raise a **failure** exception, **failure** never appears in the **signals** clause of a procedure declaration. When a procedure does not handle **failure**, **failure** propagates unchanged to the caller.

2.1.2 Atomic Transactions

Since our REV semantics will require atomic transactions, we define transactions and atomic data and then incorporate them into CLU. A *transaction* [17] is an activity that is both *recoverable* and *serializable*. Recoverable means that the net effect of an activity is all-or-nothing: either all the changes the activity makes to data happen, or none of the changes happen. Transactions are serializable in that the net effect of executing several concurrent transactions is equivalent to executing them in some sequential order. A transaction either completes successfully (*commits*) or has no effect (*aborts*).

Transactions have been useful in conjunction with databases. By masking concurrency, crashes, and communication failures, transactions simplify the construction of programs that access on-line data that must remain consistent despite concurrent access and failures. A programmer need not consider partially executed requests or interference from other programs. A program that works correctly in isolation will work correctly in the presence of concurrency and failures.

We assume every abstract data type is atomic. An *atomic* data type [43, 44] provides synchronization and recovery mechanisms for all objects of that type. In order to read or write an atomic object, a process must run as part of a transaction. Conceptually, atomic data types mask concurrency in the distributed system by serializing access to atomic objects. Atomic data types also coordinate updates to atomic objects, as all changes made to atomic objects during a transaction take effect when the transaction commits. Aborting a transaction undoes the changes made to atomic objects during that transaction. Immutable types are automatically atomic, since they provide the appropriate synchronization and recovery.

Nested transactions, which let a programmer introduce concurrency within a transaction, also let a programmer limit the scope of failures. A *subtransaction* is a transaction that runs as part of some other transaction. Aborting a subtransaction undoes all the changes it made to atomic data. Committing a subtransaction is actually conditional upon the committing of all (sub)transactions that contain the subtransaction. Therefore, if a parent (sub)transaction aborts, the effects of all its descendants are automatically undone. We call two subtransactions *siblings* if they are part of the same top-level transaction but neither is part of the other. Two siblings can commit or abort independently of each other.

We assume an atomic transaction mechanism spans the entire distributed system. Real transaction systems such as Argus [25] provide many useful features, such as concurrency mechanisms and nested top-level transactions. We illustrate the flavor of such systems without becoming deeply involved in the details by incorporating a simple model of nested transactions into CLU. We use two new reserved words: TRANSACTION and ABORT. A programmer constructs a transaction by annotating a BEGIN-END block:

```
begin [transaction] body end
```

The square brackets surrounding TRANSACTION mean that it is optional. The TRANSACTION qualifier causes *body* to execute as a subtransaction or a top-level transaction, depending on whether or not the process is already running as a transaction. Control flow statements that exit *body*, such as **return** or **signal**, implicitly commit the transaction unless they are qualified with ABORT. If a top-level transaction can not commit, the begin-end block raises the exception **failure**("commit failed"). The **failure** exception aborts every transaction that it exits. The same applies to unhandled exceptions. Note that this extension to CLU lets the programmer use nested transactions.

Programmers must use transactions with care, because the transaction mechanism does not apply to all aspects of the distributed computer system. For instance, variables fall outside the transaction mechanism. When a transaction aborts, the system does not automatically undo every assignment made by the transaction. In addition, the system does not synchronize access to variables shared by several transactions. Hence a transaction should not use variables to communicate with another transaction.

Until a transaction commits, it should not perform *external* actions. An external action is an action whose visible effects fall outside the transaction mechanism. For example, firing a missile and dispensing cash from an automated teller machine are external actions. Since an external action can not be undone by aborting the transaction in which it occurred, we assume that external actions requested during a transaction are performed (shortly) after the transaction commits. Without this assumption, transactions would not be recoverable.

Since we assume that every data type is atomic, if programmers follow the above rules concerning communication via variables and the timing of external actions, all transactions are serializable and recoverable. This will simplify the semantics of REV, which makes it easier for the compiler to determine which REV requests change program semantics.

2.1.3 Services

Before incorporating REV into CLU, we add the notion of a *service* to CLU. A service is a node that *exports* some of the routines it implements. Other nodes can use REV requests to invoke the routines exported by the service. We explain below how a service programmer declares which routines a service exports. A later section in this chapter explains how the compiler uses this information to decide if an REV request can be executed by the service.

Because a service can export many routines, we use a two-level description to structure service definitions. A service definition consists of a set of *interfaces*, and an interface defines a set of routines. We present the syntax and give examples of both interfaces and services by beginning with the syntax for interfaces:

```
interface ::= idn = [location_independent] interface [parms] [where] is
                    routineSpec, . . . end idn
```

```
routineSpec ::= procSpec | iterSpec

procSpec ::= idn = proc [parms] args [returns] [signals] [where]

iterSpec ::= idn = iter [parms] args [yields] [signals] [where]
```

The nonterminals *parms*, *returns*, *yields*, *signals*, *where*, and *args* are defined as in the CLU manual. To avoid name conflicts, we assume interface names are globally unique.

An *interface* is a collection of type specifications for routines. Whether an interface defines a type or simply gathers together a collection of related routines without defining a type is irrelevant to this discussion. A routine is either a procedure or an iterator. No two interface routines may have the same name. A CLU cluster, which implements an interface, *exports* each routine listed in the interface. Interfaces let the programmer separate specifications of routines from their implementations.

LOCATION_INDEPENDENT is a new reserved word. We say a routine P is *location-independent* if it does not depend on which node executes P. For example, logical operations on booleans, such as **and** and **or**, are location-independent. Functional routines, array operations, and record operations are also location-independent. Other routines, such as **GetMyNetworkAddress**, **GetNearestPrinter**, and **LocalFile$Open** are *location-dependent*: the semantics of each such routine depends on the state of the node that executes it.

If LOCATION_INDEPENDENT is present in an interface, all routines defined by the interface are location-independent. Otherwise, the routines in the interface are assumed to be location-dependent. Although LOCATION_INDEPENDENT could annotate individual routines, we assume it annotates only interfaces and clusters. Section 2.4 explains how the compiler uses LOCATION_INDEPENDENT annotations.

We illustrate interfaces and location independence with two examples. Figure 2-1 defines the abstract data type **point**, which has location-independent routines. The **point** routines do not depend on the internal state or the physical location of the node that executes them. Figure 2-2 defines an interface for a mail system without defining a new abstract data type. Because the **postOffice** routines can be affected by the internal state of the node implementing the routines, the reserved word LOCATION_INDEPENDENT is not present. For instance, it makes a difference whether an administrator removes Jones from the Dallas registry or the Chicago registry.

Having discussed interfaces, we turn to service definitions, which are sets of interfaces. A programmer defines a service by a list of identifiers:

```
service ::= idn = service is idn, . . . end
```

Each identifier names an interface or another service, and SERVICE is a new reserved word. The meaning of a service definition is the set of routines contributed by the identifiers. An interface identifier contributes the routines it defines, while a service identifier contributes the routines it exports.

```
point = location_independent interface is

  create = proc (x, y: int) returns (point)

  x = proc (p: point) returns (int)

  y = proc (p: point) returns (int)

  r = proc (p: point) returns (real)

  theta = proc (p: point) returns (real)

  distance = proc (p, q: point) returns (real)

end point
```

Figure 2-1: An interface defining the abstract data type **point**.

```
postOffice = interface is

  addUser = proc (requester, user: userID) signals (userAlreadyExists)
    % requester must be a system maintainer

  removeUser = proc (requester: userID, user: string) signals (noSuchUser)
    % requester must be the user or a system maintainer

  readMail = proc (user: userID) returns (array[string]) signals (noSuchUser,unreadable)

  anyMail = proc (user: userID) returns (bool)

  sendMail = proc (user: userID, msg: string) signals (undeliverable)

end postOffice
```

Figure 2-2: An interface for a mail system.

The ability to define a service by extending another service provides programming convenience. A service definition is meaningless, however, if it is directly or indirectly recursive. For example, a programmer can not define service S1 in terms of service S2 if service S2 is defined in terms of service S1. The compiler "flattens" a service definition that depends on another service. Two service definitions are equivalent if their compilation results in the same set of routines.

The following three service definitions are well-defined because they are not recursive:

```
built-ins = service is int, bool, char, real, string, array, record end

mathematics = service is int, real, complex, matrix, polynomial, trig, algebra end

graphics = service is bitmap, point, line, polygon, font, built-ins end
```

Built-ins is a collection of useful types that most or all services should support. Besides being a bona fide service, built-ins can be part of another service definition, such as graphics. A node

supporting **built-ins** exports the seven interfaces in the definition of **built-ins**. A node supporting **graphics** exports the twelve interfaces in the definition of graphics. Five interfaces are mentioned explicitly, while the other seven are inherited from **built-ins**. A node supporting only **mathematics** supports only straight-line code, because it does not export **bool**. To avoid this problem, we shall assume that every service runs CLU and therefore automatically exports **built-ins**.

A service that exports interface T must implement interface T, all the types mentioned in interface T, all the types mentioned in the interfaces for these types, and so on. An easy way of extending a service without increasing the number of interfaces that its instances must implement is to take the type closure of a service. Chapter 7 presents the details.

When a service definition contains a parameterized interface, such as **array[t:type]**, for simplicity we restrict type parameters to those types exported by the service. For instance, a **graphics** service supports operations on objects with type **array[array[line]]** since it exports both **array** and **line**. A **graphics** service, however, does not support objects with type **array[matrix]**, since it does not export **matrix**. This restriction on type parameters does not reflect any fundamental limitation on the transmission of code in a network, since it is possible to transmit a cluster implementing **matrix** to a **graphics** service. We made this decision to avoid problems that arise when clusters with own data are sent between nodes. Specifically, a service that implements but does not export a type with own data might otherwise find itself with several sets of own data for the same type.

We distinguish between the specification (i.e., definition) of a service and its instances. A *service instance* is a node that exports the interfaces in the service specification. Such a node can have internal state by using own variables. A node can be an instance of many services simultaneously. For example, a node exporting **graphics** to one client can export **built-ins** to another (or the same) client. A service is a view of a node: it guarantees that the node exports a certain set of interfaces, but does not prevent the node from exporting additional interfaces.

The separation between specification and implementation imposes a partial order on the compilation of interfaces, service definitions, and programs. The compiler rejects any service definition that names an interface or service whose definition is not in the compilation environment. This rejection prohibits recursive service definitions. Similarly, the compiler rejects any program that names an interface or service whose definition is not in the compilation environment. This rejection, which lets the compiler perform strong type checking across module boundaries, will let the compiler determine whether a service can execute an REV request. Section 2.3 presents the details.

Before an application program can send an REV request to an instance of service S, it must bind to the instance. Two important issues in distributed binding are *naming* and *location* [4]. A client specifies what constitutes an acceptable service, and a network facility such as Grapevine [3] uses the description to locate an appropriate instance of the service at run time. We focus on naming issues, under the belief that the corresponding location mechanisms may be built.

Our remote binding model consists of two procedures which we discuss in turn. A programmer needing any instance of a service uses the following procedure to bind to a node exporting the service:

```
Service[s:serviceName]$Any = proc () returns (s) signals (NoneAvailable)
```

This procedure, which is part of the run-time system, consults a network facility to locate some instance of the specified service. **Service[s]$Any** returns a node that exports at least service **s**. Because the programmer has no control over which instance is selected, the service **s** should be a *location-independent* service. A service is location-independent if all its instances are indistinguishable except for performance. For example, consider a room containing ten identical nodes, and assume no node contains a local file system. If each node compiles Fortran programs, they are indistinguishable to a user. The appropriate node is the least busy. If **Fortran** is a location-independent service that compiles Fortran programs, the following statement finds an instance of this service:

```
compiler: Fortran := Service[Fortran]$Any()
```

No particular Fortran compilation service is requested, since they are all equivalent.

A programmer needing some instance of a service uses the following procedure to bind to an appropriate node exporting the service:

```
Service[s:serviceName]$Lookup = proc (string) returns (s) signals (NotAvailable)
```

Service[s]$Lookup returns a node that exports at least service **s** and corresponds in some way to the string argument. As explained below, the interpretation of the string argument depends on the service **s**.

Service[s]$Lookup gives the programmer some control over which instance is found, which is useful when the programmer wants to use a *location-dependent* service. A service is location-dependent if its instances are distinguishable because of their data or physical locations. For example, a user may prefer the instance of the print service down the hall instead of the instance across town. Similarly, many services may contain street maps and export relevant operations, but a user may want one containing a map of Boston.

Suppose **Map** is the name of a location-dependent service that contains street maps and provides operations on the maps. The following statement finds a service containing a map of Boston:

```
BeanTown: Map := Service[Map]$Lookup("Boston")
```

Since all **Map** services do not contain a Boston map, the programmer must specify which instances are acceptable. As we mentioned above, the interpretation of the string argument ("Boston") depends on the service (**Map**). A single string may correspond to zero, one, or several instances of the service.

Each of the two binding routines presented above normally returns a *capability* for a service instance.

A capability for service instance S lets a node execute REV requests at S. Service capabilities are first-class objects. They may be assigned to variables, stored in data structures, passed as arguments, or returned as results. Service capabilities may also be sent between nodes; Chapter 5 presents the details.

Although a client that has finished interacting with a service is expected to notify the service, for simplicity we do not include a **Service[s]$finished** routine in the discussion or any examples in the thesis. Such a routine would invalidate the service capability, break the binding between the client and the service, and let the service reclaim any resources it devoted to the binding. An REV request whose destination is an invalid service capability does not run. Instead, it raises the exception **failure**("invalid service capability").

A service can unilaterally invalidate any service capability it issued. For example, if a service crashes and then recovers, it can declare that all its outstanding service capabilities are invalid. Similarly, a service unable to communicate with a client can declare that the capability issued to the client is invalid. When either the client or the service invalidates the service capability for a binding, the service reclaims the resources it devoted to the binding.

2.2 REV Requests

Having extended CLU with services and a simple model of nested transactions, we now incorporate REV into CLU by defining what an REV request means (its semantics) and how one is written (its syntax). This chapter assumes an REV request relocates the execution of a procedure. Consider a procedure P that is not exported by a service S, and assume an REV request relocates the execution of P to S. For simplicity, assume the body of P invokes only procedures exported by service S. The run-time system for a client executing the REV request places the body of P and the arguments into a request message; sends the request message to the service; and waits for a reply message. When the service receives the request message, it extracts the procedure body and arguments from the message; evaluates the procedure with the arguments; and then places the procedure's results into a reply message. The service sends the reply message to the client, which extracts the results and continues execution. We formalize the semantics of such an REV request after presenting the syntax.

An application programmer writing an REV request specifies the procedure, the arguments, and the service that executes the REV request. We use the extended BNF defined in the CLU manual [24] to present REV syntax:

```
rev_expression ::= at expression eval invocation
```

AT and EVAL are two new reserved words, and *expression* must have a service type. A relocated invocation has the same syntax as an ordinary procedure invocation:

```
invocation ::= primary([expression, . . .])
```

The nonterminal *primary*, which is one kind of CLU expression, produces literals, identifiers, and invocations. We extend *primary* to produce REV requests, which are relocated invocations:

```
primary ::= . . . (as in CLU manual) . . . | rev_expression
```

A client executing an REV request evaluates the expression following the reserved word AT to determine a service. After evaluating the relocated invocation's *primary* and arguments, the client sends the request to the service, which performs the invocation and returns the results to the client. We present restrictions on REV requests and define their semantics below.

An REV request that is executed by service S is *valid* if the procedure and arguments may be transmitted to S and the results may be transmitted to the client. If T is any type other than code, an object of type T may be transmitted between two nodes if T is a transmissible type and both nodes implement type T. The transmissibility of code is more involved. As explained in Section 2.3, the request message contains enough code for the service to execute the invocation. This code, which the compiler must be able to determine at link time, is self-contained in that it never refers to nonlocal variables. Moreover, every routine invoked by the code at the service is either in the request message or exported by the service.

An REV request lets an application programmer relocate the execution of a procedure. If the procedure is location-dependent, relocating execution with REV may change program semantics. The meaning or *semantics* of a program is its visible behavior, including any results computed by the program and any visible side effects it causes. We do not include resource consumption, such as processor time, memory requirements, and network traffic, in the definition of program semantics.

Using REV to relocate the invocation of a location-dependent procedure may change program semantics. On the other hand, if the procedure is location-independent, we would like the REV request not to change program semantics. Section 2.4 argues that the following argument passing semantics and crash semantics for REV requests achieve this goal.

Both REV requests and ordinary procedure calls have the same argument passing semantics: call by sharing. Conceptually, the client places the names of the arguments in the request message sent to the service. Because such an implementation would be hopelessly inefficient, we discuss a novel implementation for call by sharing in a distributed system in Chapter 3.

Having specified the argument passing semantics for REV requests, we must define their crash semantics: how do node and communication failures affect the meaning of an REV request? We simplify the construction of distributed applications by masking node and communication failures with atomic transactions. Each REV request must run as part of a transaction that aborts if the REV request does not complete.[3] Invoking an REV request outside the scope of all transactions raises the

[3] Transaction may mean top-level transaction or subtransaction in this discussion.

exception failure("No current transaction"). In this case, the REV request is not executed. The normal completion of an REV request does not affect the status of the current transaction. In contrast, unilateral termination of an REV request by the client's run-time system aborts the current transaction. For instance, if the client's REV mechanism can not periodically elicit a low-level response from the service, it might conclude that further attempts at communication are worthless and abort the transaction. When the transaction is aborted, the nearest enclosing transaction block raises the exception failure(reason: string). Being part of a transaction that modifies only atomic objects, each REV request has the following semantics: either the service executes the REV request and returns the results to the client, or the REV request has no effect on both the service and the client. In the absence of failures, REV requests have *exactly-once* semantics since each request is executed once.

Figure 2-3 contains an example of REV. Suppose a remote array processor supports addition, subtraction, and multiplication of matrices. A programmer who wants to exponentiate a square matrix can use REV to implement an exponentiate procedure that executes at the array processor. For simplicity, we assume that the matrix is square and the power is a nonnegative integer. The algorithm calculates the result by successive squaring.

```
APS = ArrayProcessorService      % an equate

exponentiate = proc (m: matrix, power: int) returns (matrix)
  signals (unavailable(string))

  begin transaction
    ap: APS := Service[APS]$Lookup()
      except when NotAvailable: abort signal unavailable("No ap's")end
    answer: matrix := at ap eval exp(m, power)
    return (answer)
  end except when failure (reason: string):
          % REV or commit problem
          signal unavailable(reason) end
end exponentiate


exp = proc (m: matrix, p: int) returns (matrix)
  % calculate m^p by successive squaring

  square: matrix := m
  ans: matrix := matrix$identity(matrix$length(m))
  while p>0 do
    if int$mod(p, 2)=1  % is p odd?
      then ans := matrix$multiply(ans, square) end
    square := matrix$multiply(square, square)
    p := p/2
  end
  return (ans)
end exp
```

Figure 2-3: Using REV to enhance a remote array processor.

Since procedure **exp** in Figure 2-3 does not depend on the state of the node that executes it, **exp** is a location-independent procedure. We argue in Section 2.4 that using REV to relocate the execution of a location-independent procedure does not change program semantics. In other words, removing the phrase "**at ap eval**" in **exponentiate** does not change program semantics. Whether the client executes **exp** (as an ordinary procedure invocation) or the service executes **exp** (as an REV request) does not change the results calculated by **exp**. Because **exp** causes no side effects, we do not need transactions to ensure this particular REV request does not change program semantics. Other examples in the thesis, however, show that transactions are often necessary.

A programmer can nest REV requests, as shown in Figure 2-4. Procedure R uses REV to relocate the execution of procedure P, which in turn uses REV to relocate the execution of procedure Q. As explained in Section 2.3, the compiler determines the code the client sends to *s1* and the code *s1* sends to *s2*. The code sent to *s2*, which probably includes the body of Q, is first sent to service *s1* which treats the code as a black box. The code sent to *s2* depends on Q and *s2* but not on *s1*.

```
R = proc (**args**) returns (**results**)
    .
    .
  at s1 eval P(**args**)
    .
end R

P = proc (**args**) returns (**results**)
    .
    .
  at s2 eval Q(**args**)
    .
end P
```

Figure 2-4: A nested REV request.

REV requests can nest to an arbitrary depth. For instance, if procedure Q invoked procedure P, they would be mutually recursive. In general, their dynamic nesting depth could not be predicted at compile time.

The transmissibility of service capabilities facilitates nested REV requests. If *s2* is (contained in) an argument to procedure P, the service denoted by *s2* accompanies P to service *s1*. If service capabilities were not transmissible, the programmer would have to establish a binding in procedure P to relocate the execution of procedure Q.

2.3 The Code Portion for an REV Request

The *code portion* for an REV request consists of all routine implementations and routine names the client sends to the service, excluding the code portions for all nested REV requests. The code portion represents the smallest amount of code that the client must send to the service for the service to execute the REV request. When determining the code portion for an REV request, we treat nested requests as self-contained, black boxes, because a nested REV request may be executed by another service and therefore may be unintelligible to the intermediate service.

At this point, we explain in detail the three restrictions we impose on REV requests. The first restriction, which we impose for semantic reasons, applies to the routines invoked by an REV request at the service that executes the request. We say the code portion for an REV request *imports* a routine if:

1. the REV request uses the routine *at the service executing the request*; and
2. the request message sent to the service does not contain an implementation for the routine.

Given this definition, our first restriction is that every routine imported by an REV request is exported by the service executing the request. Otherwise, a service could receive a request that asks the service to execute a routine the service does not implement.

The second restriction, which is also imposed for semantic reasons, prohibits the code portion from having free variables and own variables. Unlike LISP, we use type information to distinguish free variables from free procedure names. Since CLU does not let a programmer define a routine in the body of another routine, the only free variables a routine can have are own variables. Hence this restriction is equivalent to one that prohibits own variables in the code portion of an REV request. This restriction on own variables does not reflect any fundamental limitation on the transmission of code in a network, since it is possible to transmit the objects bound to the own variables appearing in the code portion of a request. If we allow own variables in the code portion, implementing REV so that location-independent requests preserve program semantics is difficult because of our assumption that variables fall outside the scope of the transaction mechanism. This restriction and the preceding restriction ensure the code portion is self-contained as long as it is executed by an instance of the appropriate service.

Our third and final restriction on REV requests, which is imposed for pragmatic reasons, requires that the compiler be able to determine the code portion at link time. This restriction supports early error detection, since the compiler can check the above two restrictions. This restriction also lets the compiler encode each code portion. *Encoding* an object results in a sequence of bits that represents the abstract value of the object in a node-independent way. Such a bit sequence may be sent between nodes in messages. Having the compiler encode the code portion of each REV request can improve run-time performance.

The code portion for an REV request depends on the service executing the request. Consider the REV request in Figure 2-5, which relocates the execution of procedure P to a instance of service S. Assume no procedures are used as arguments to A, B, C, or P. If service S exports P, the code portion consists of the name P. This case is comparable to a remote procedure call, which shows that remote evaluation includes remote procedure call as a special case. If service S exports A, B, and C but not P, the code portion consists of the body of P. If service S exports only A and B, the code portion consists of the bodies of P and C. If S does not supply enough routines, the code portion may not exist. For example, assume S exports only B and C, and assume that A is a system routine that is not implemented in CLU. If only CLU routines may be transmitted between nodes, A is not transmissible and no self-contained code portion exists.

```
P = proc (**args**) returns (**results**)
  A(**)
  B(**)
  return C(**)
end P

C = proc (**args**) returns (**results**)
  if B(**)
    then return P(**)
    else return A(**)
  end
end C

  .
  .

somewhere: S := Service[S]$Any()
at somewhere eval P(**args**)
```

Figure 2-5: An REV request that relocates the execution of procedure P.

We call the procedure whose execution is relocated by an REV request the *relocated* procedure for that request. In the preceding example, P is the relocated procedure. The relocated procedure is independent of the service that executes the request. In particular, it does not matter whether the service exports the relocated procedure or whether the body of the relocated procedure accompanies the request message sent to the service. We call any other routine whose body accompanies the REV request to the service a *client-supplied* routine. As we saw in the previous example, the client-supplied routines depend on the service executing the request. For example, if the service exports only A and B, then C is a client-supplied routine.

We describe how the compiler determines the code portion for an REV request by first assuming that routines are not first-class objects. Under this assumption, a programmer can write routines and invoke routines, but can do nothing else with routines. While linking the program, the compiler determines the code portion for each REV request by generating part of the *program call graph*,

which represents the "who-calls-who" relation for routines. Because routines are not first-class objects, the program call graph can be readily constructed at link time. For each REV request, the compiler begins with the relocated procedure and decides whether the client must send the procedure's name or body to the service. In the latter case, the compiler must also decide whether the client must send the name or body of each routine invoked by the relocated procedure, and so on. In the worst case, the compiler must determine all routines reachable from the relocated procedure. If the compiler comes across a nested REV request, it recursively calls itself on the nested REV request and then continues processing the current REV request. If no code portion exists or if the code portion contains an own variable, the compiler informs the programmer via a fatal diagnostic and continues checking for other link-time errors, but the compiler does not produce an executable program. On the other hand, if the compiler finds the smallest self-contained code portion, it encodes the code portion and continues linking the program.

Using routines as first-class objects complicates the task of determining the code portion for an REV request, since arguments and results may be routines. The compiler rejects an REV request with routines as arguments if any of the routine arguments are unknown at link time. Similarly, the compiler rejects an REV request whose relocated procedure is unknown at link time. The compiler also rejects any argument to the REV request that contains code, such as an array of procedures, by prohibiting any external representation that contains code. Without an acceptable external representation, an abstract data type is not transmissible, which means that the REV request is not valid. These restrictions ensure the code portion of an REV request is apparent at link time.

In a distributed system with REV, routine names are bound to their implementations at different times. We will use Figure 2-6 to show the different times at which binding occurs. A, B, C, and D are clusters. Service *S1* exports C and D, and service S2 exports C. Consider a program running at the client that contains REV requests. Assume modules A and B are compiled once, linked together, and then loaded at the client.

Consider an invocation of a routine that is not in the code portion of any REV request. Such an ordinary invocation is bound early and only once. If the routine is invoked in the same cluster as it is defined, binding occurs at compile time. For instance, a routine in cluster A could invoke another routine in A. Otherwise, if the routine is invoked in one cluster and implemented in another cluster, the invocation is bound to an implementation when the modules comprising the client program are linked. For example, a routine in A could invoke a routine in B.

A routine that is invoked in an REV request but is not imported by the REV request must accompany the REV request. Such invocations are bound early and only once. Suppose an REV request sent to service *S1* invokes the routine **B$op**. Since *S1* does not export B, **B$op** must accompany the request. The binding occurs when the modules comprising the client program are linked.

A routine that is imported by an REV request is bound at run time. Consider an REV request that

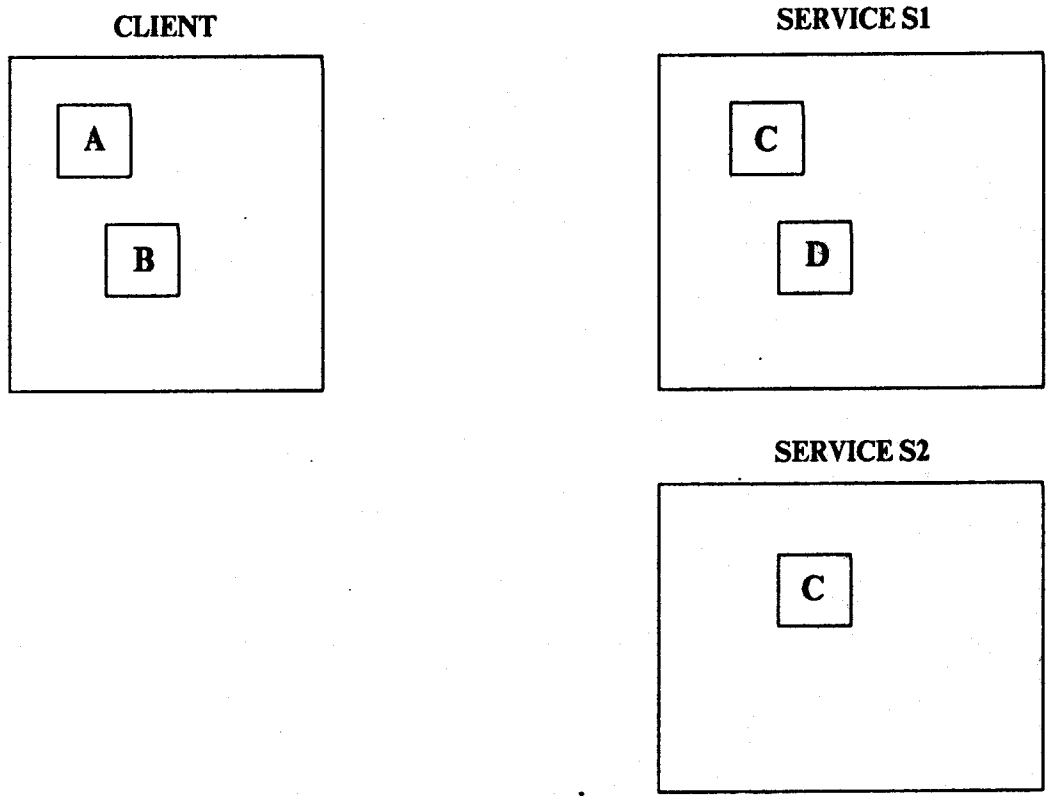


Figure 2-6: A simple distributed system.

imports the routine **C$op** and is sent to a service that exports C such as *S2*. The invocation of **C$op** is bound to an implementation for **C$op** every time the REV request is executed by *S2*. The binding can occur many times at the same service or many times at many services. For instance, the REV request could also be executed several times by *S1*.

2.4 Location Independence

A *location-independent* REV request is a valid REV request that imports only location-independent routines. The code portion for such an REV request is a composition of location-independent routines, which means that it is also location-independent. Our goal is to let the programmer relocate execution with location-independent REV requests without affecting program semantics. In other words, every execution sequence of a program containing a location-independent REV request must be equivalent to some execution sequence of the corresponding program without the REV request and vice versa. This section argues that relocating execution with a location-independent REV request has no effect on program semantics. It also discusses how location independence affects application programmers and service programmers.

We being by arguing that relocating execution with a location-independent REV request has no effect on program semantics, provided that the expression that specifies the service is side-effect free. Let R be an REV request of the form:

```
at somewhere eval P(**args**)
```

where *somewhere* is an expression with no side effects and P is a location-independent procedure. All of P's observable side effects must be modifications to its arguments, because otherwise P would be location-dependent. Assume R is always evaluated during a transaction. The following shows that replacing expression R with expression P(**args**) has no effect on program semantics:

- *The argument passing semantics for* R *and* P *are identical.* Both use call by sharing. Since P is location-independent, the arguments are modified in the same way whether the client or the service executes P.

- *The results computed by* R *and* P *are identical.* Since P is location-independent, the objects returned by P(**args**) do not depend on the node that executes it. Therefore, it does not matter whether the client or the service executes P.

- *The crash semantics for* R *and* P *are identical.* In both cases either P(**args**) is completely evaluated or it appears that P(**args**) was never evaluated. The transaction mechanism masks all node and communication failures that might affect the evaluation of P(**args**) or R.

Hence we may conclude that location-independent REV requests preserve program semantics. Each execution sequence of a program containing location-independent REV requests is equivalent to some execution sequence of the corresponding program without these REV requests and vice versa.

Our suggested programming methodology is based on the ideas of location dependence and location independence. The application programmer first writes and debugs an application using REV only as a way of fixing the execution site of location-dependent procedures that are not executed by the client. We recommend that the service expression in such an REV request be location-independent, so that the entire REV request will a location-independent program fragment. For example, let M be an REV request of the form:

```
at someService eval Q(**args**)
```

where *someService* is a location-independent expression and Q is a location-dependent procedure. No matter which node begins to execute M, the specified service ultimately executes Q. Thus the location dependence of the relocated procedure Q does not affect the location independence of the entire REV request M. This kind of request is a prime candidate for becoming a nested request, as explained below.

Location-independent requests will normally be used during the second stage of our suggested programming methodology. Once the application has been debugged, the application programmer inserts location-independent REV requests to improve performance. For example, consider a procedure with five REV requests to the same service. The programmer can try to use REV to relocate the execution of the entire procedure to that service, which will make the five REV requests

in the procedure nested requests. The programmer need not worry about introducing new bugs at this stage, as location-independent requests preserve program semantics, and the compiler can check whether each new REV request is location-independent. This programming methodology facilitates the automatic insertion of location-independent requests to improve performance, a topic that is beyond the scope of this thesis.

Our notion of location independence involves the service programmer, who must declare whether each interface exported by a service contains location-independent routines or location-dependent routines. The compiler uses this information when checking the location independence of an REV request. The following approach can statically check the location independence of service routines. System programmers label the built-in routines as to their location independence, while service programmers label the routines they implement. The compiler ensures that every routine invoked or named by a location-independent routine is location-independent. This approach, however, may be too conservative, since a routine can invoke location-dependent routines yet remain location-independent.

Application programmers need not decide whether their routines are location-independent, since the compiler infers their location independence when checking the location independence of an REV request. This division of labor between application programmers, service programmers, and the compiler reflects an important theme in the thesis. Whenever possible, we place the burden on the compiler and service programmer rather than on the application programmer in an attempt to simplify the construction of distributed applications. A handful of expert language designers and compiler writers can support a small number good service programmers, who in turn can make life easier for the hordes of application programmers. This division of labor increases the leverage of language designers, compiler writers, and service programmers.

2.5 Discussion

To facilitate (automatic) program optimization, we defined REV semantics so that location-independent requests would not change program semantics. Since an REV request relocates the execution of a procedure, we had to ensure that procedures and REV requests had identical semantics. We also wanted to minimize the changes to CLU, since we were concerned with programming style and the efficiency of CLU procedures. Because our goals were different than those of RPC researchers, our REV semantics is unlike any RPC semantics.

Our argument passing semantics for REV requests, call by sharing, is unusual in a distributed system. Most RPC systems use call by value, but CLU uses call by sharing. Unifying the semantics of local procedures and REV requests forced us to choose call by sharing for REV requests. Although call by sharing and call by value are equivalent for immutable types, we did not want to use call by value for REV requests and limit their arguments and results to immutable types because we thought that it would constrain the application programmer (and optimizer) too much.

Chapter 3 describes our novel implementation for call by sharing in a distributed system. In terms of performance, our implementation is shown to be comparable to an implementation for call by value. Chapter 3 also discusses how we keep separate address spaces disjoint, even though REV requests use call by sharing.

We masked node and communication failures with atomic transactions so that location-independent REV requests would not change program semantics. Our crash semantics, however, is slightly different than the semantics found in transaction-based RPC systems like Argus [25], which automatically enclose each RPC in a transaction. If we used the same approach and automatically enclosed each REV request in a transaction, at least part of the transaction structure of a program would depend on how the program is partitioned into components for local and remote execution. This in turn means that program semantics depend on how the program is partitioned. Relocating execution with a (location-independent) REV request would alter the transaction structure and in general change program semantics.

Besides helping us ensure that location-independent REV request preserve program semantics, transactions simplify the construction of distributed applications by masking node and communication failures. When failures are visible to the application programmer, building a distributed system, especially one that must maintain the consistency of distributed data, is a difficult task. Independent failure modes complicate the behavior of the system, because a client with an outstanding REV request may not be able to communicate with the service purportedly executing the request. In such cases, the client is unable to determine whether the service received and (partially) executed the request.

2.6 An Example

We highlight the important points in the chapter with an example that uses a location-independent REV request to improve performance without affecting program semantics. Consider the problem of sending a form letter to several people. Current mail systems let a user send the same message to several recipients. A user who wants to customize each copy of the letter, however, must send separate messages. Assume the customization can be automated. For example, the user may want to insert "Dear John," in the message to John, "Dear Sue," in the message to Sue, and so on. Furthermore, assume all recipients of the message are in the same mail registry.

Let **mail** be a service that exports the **registry**, **maildrop**, **set**, and **string** interfaces. **Customize**, the procedure in Figure 2-7, uses REV to customize the message at the mail service and returns those recipients without a mailbox. Without REV, the user would be forced to customize the message at the client and send each copy to the mail service. If there are many recipients or if the message is long, REV should improve performance substantially.

```
ss = set[string]        % an equate

customize = proc (user, passWord, msg, registry: string, friends: ss)
  returns (ss) signals (NotAvailable)

  begin transaction
    postOffice: mail := Service[mail]$Lookup(registry) abort resignal NotAvailable
    badNames: ss := at postOffice eval
      customizeMsg(postOffice, friends, user, passWord, msg)
    return (badNames)
  end except when failure (reason: string):
    % REV or commit problem
    signal NotAvailable end
end customize


customizeMsg = proc (po: mail, friends: ss, user, pwd, msg: string) returns (ss)

  badNames: ss := ss$new()
  newMsg, firstName: string

  for friend: string in ss$elements(friends) do
    firstName := at po eval registry$firstName(friend)
      except when noSuchUser:
        ss$insert(badNames, friend)
        continue   % start the next iteration
      end
    newMsg := "Dear "||firstName||msg  % string concatenation
    at po eval maildrop$send(user, pwd, friend, newMsg)
  end

  return (badNames)
end customizeMsg
```

Figure 2-7: Using REV to customize a form letter.

The REV request that relocates **customizeMsg** is location-independent, since its code portion imports only **set** and **string** operations, which are location-independent. Although **customizeMsg** invokes location-dependent **registry** and **maildrop** routines, it does not import them. This is because the code portion for an REV request by definition does not include the code portion for any nested requests. Following our suggested methodology, the application programmer writes and debugs **customizeMsg** using REV to fix the execution sites of location-dependent **registry** and **maildrop** procedures. Later, the programmer can easily convert any invocation of **customizeMsg** to an REV request without changing program semantics. Both procedures and REV requests use call by sharing, and the application programmer does not have to worry about keeping separate address spaces disjoint.

Making location-independent REV requests be an optimization simplifies application programming. For example, the application programmer can relocate the execution of **customizeMsg** without changing program semantics. To improve performance, the programmer could relocate **customizeMsg** from the client to the mail service that contains the recipient's mailboxes, as shown

in Figure 2-7. Even if the programmer accidentally relocates **customizeMsg** to the wrong mail service, performance may degrade but the net effect of the REV request will be unchanged because the nested REV requests are executed by the correct service. Which node executes a location-independent REV request can affect program performance but not program semantics.

This example also shows how transactions make location-independent REV requests be an optimization. Whether **customizeMsg** is part of an REV request or simply an ordinary procedure has no effect on program semantics. When REV is used to relocate the execution of **customizeMsg**, the client sends only a single REV request to the service. The service sends the REV requests inside **customizeMsg** to itself. Removing the phrase "**at** postOffice **eval**" in **customize** replaces the REV request with an ordinary invocation of **customizeMsg**. The number of REV requests the client now sends to the service depends on the number of recipients and the number of recipients with mailboxes. The failure of any REV request causes the transaction to abort and **customize** to raise the exception **NotAvailable**. Either all the friends with mailboxes receive customized messages or none of them do. These two outcomes are also the only possible outcomes when REV is used to relocate the execution of **customizeMsg**. Hence the possible outcomes are independent of whether the client or the service executes **customizeMsg**.

One final point to note about this example is how REV can extend the set of routines "exported" by a service. Whether the code portion contains the name or the body of **customizeMsg** is transparent to the application programmer in terms of syntax, semantics, and performance. The only difference is whether the procedure exists at the service or accompanies the REV request.

2.7 Summary

REV is the ability to relocate the execution of a procedure. REV requests, which give the application programmer fine-grained control over the location of processing in a distributed application, use call by sharing. Instead of automatically enclosing each REV request in a transaction, we require that every REV request run as part of some transaction. If the client's REV mechanism unilaterally terminates an REV request, it aborts the associated transaction and then raises the exception **failure**. Transactions, which mask concurrency, node crashes, and communication link failures, manipulate only atomic objects.

We restrict REV requests so the compiler can verify the validity of each REV request and encode its code portion. An REV request is valid if its arguments and results are transmissible between the client and the service and the code portion is self-contained and apparent at link time. In particular, the code portion must not contain any own variables; every procedure argument must be known at link time; and every routine imported by the code portion must be exported by service executing the request. Valid requests may be encoded by the compiler.

A location-independent REV request is a valid REV request that imports only location-independent routines. A routine is location-independent if its semantics does not depend on the node that executes the routine. Location-independent REV requests do not change program semantics.

To determine the validity and location independence of an REV request, the compiler consults the appropriate service definition, which is a set of interfaces. An interface specifies a collection of routines. A service programmer defining an interface must specify whether the interface defines location-independent routines.

An instance of a service, which is a node exporting one or more interfaces, advertises its existence by registering with some network facility. A program that uses REV binds to one or more services. This binding typically occurs at run time. An invocation of a routine imported by an REV request is usually bound late and often. Other invocations of routines are bound early and only once.

Chapter Three

Implementing REV

The preceding chapter presented an integrated set of new ideas including REV, services, and call by sharing in a distributed system. This chapter, which explains how to implement these ideas, is based on a prototype REV mechanism we constructed. Because of the similarity between RPC and REV mechanisms, we focus on the major innovations in our implementation.

3.1 Overview

We begin by surveying the compile-time and run-time tasks of an REV mechanism. An REV mechanism has two tasks at compile time. First, it verifies the validity of REV requests and determines their location independence. Recall that a valid request is one the compiler can encode and the service can execute, while a location-independent request is a valid request that does not change program semantics. Compared to run-time checking, this static checking can improve performance and detect errors earlier.

The second compile-time task of an REV mechanism is to generate *stubs*. A stub is a procedure that encodes or decodes arguments and results at run time. Stubs interface the application program with the communication system and free the application programmer from worrying about communication details. Generating stubs at compile time rather than at run time improves run-time performance.

At run time, an REV mechanism has six tasks:

1. *Service Binding:* A program with REV requests must locate the services that will execute the requests and bind to these services.

2. *Reliable Communication:* Two nodes must reliably exchange messages of arbitrary length over an unreliable network that may be based on packets.

3. *Failure Recovery:* Each REV request must run as part of some transaction that aborts if the request does not complete. Thus node and communication failures are hidden by atomic transactions.

4. *Call by Sharing:* Our argument passing semantics for REV requests is call by sharing. An efficient implementation for call by sharing in a distributed system is the major innovation in this chapter.

5. *Code Transmission:* A client must transmit routine implementations and routine names to

a service. We will evaluate several alternatives for the external representation for code and discuss scenarios in which each alternative is appropriate.

6. *Request Interpretation:* If the external representation for code is something other than compiled code, each service needs an interpreter to evaluate REV requests.

The difference between an REV mechanism and an RPC mechanism is small. In a remote procedure call, data and results are transmitted between the client and service. The RPC request message names one procedure that resides at the service (Figure 3-1). A client using REV, in contrast, transmits the "remote" procedure along with its arguments (Figure 3-2). The procedure sent from the client can name several procedures residing at the service.

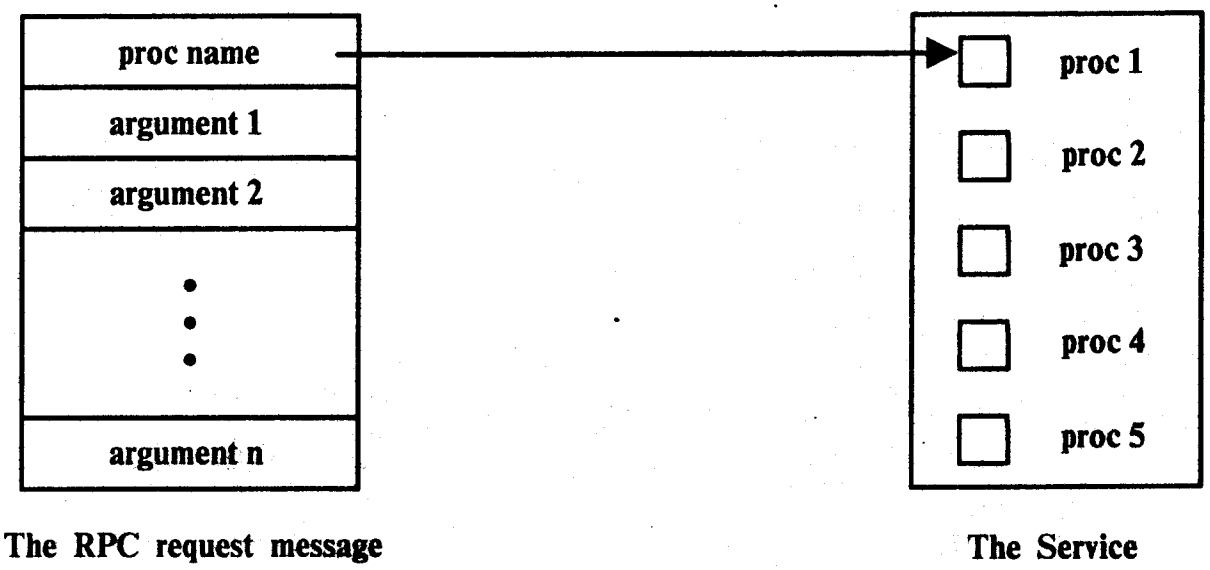


Figure 3-1: An RPC received by a service.

The remainder of the chapter is structured as follows. Section 3.2 explains our technique for implementing call by sharing for REV requests. Section 3.3 discusses the compile-time activities of an REV mechanism in some detail, while Section 3.4 does the same for the run-time activities. Because of the similarity between an REV mechanism and an RPC mechanism, these two sections focus on the REV tasks that are not found in an RPC mechanism: verifying the validity of REV requests and determining their location independence; implementing call by sharing in a distributed system; and transmitting code between nodes. The REV tasks that are also found in an RPC mechanism are discussed briefly.

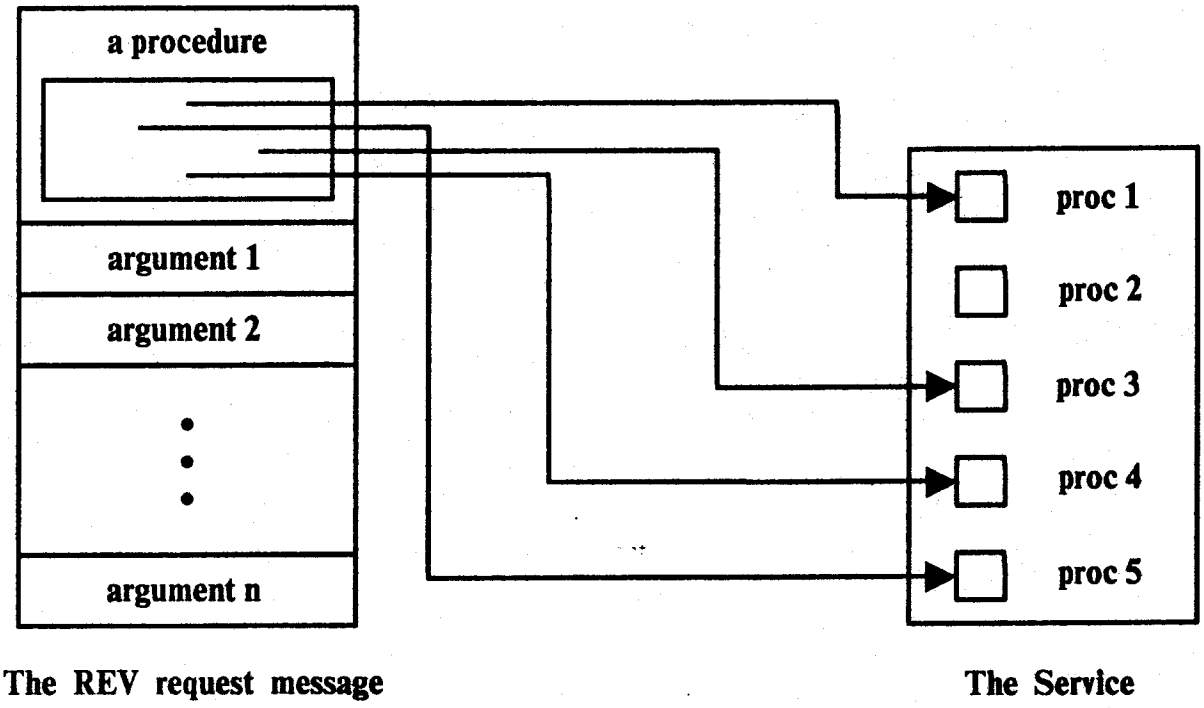


Figure 3-2: An REV request received by a service.

3.2 Call by Sharing in a Distributed System

A straightforward implementation of call by sharing for REV requests uses object names that are resolved in a global address space shared by all nodes. This approach has two drawbacks. First, *call-backs* will cause poor performance. A call-back is a nested REV request sent from the recipient of the outer REV request to the sender of the outer request. The following example shows how global names in an REV request can cause call-backs. Let A be an argument to an REV request sent to the service. If A is represented by a global name that refers to an object kept at the client, accessing A during the REV request requires a call-back. If a single REV request requires dozens of call-backs, most of the performance advantages of REV will disappear.

Second, a straightforward implementation of call by sharing will not keep node address spaces disjoint. A service that holds onto the arguments of an REV request will refer to client objects. If the results of the REV request are service objects, the client will refer to service objects. Call by sharing semantics dictates that an accessible reference be valid for all time. This requirement complicates

garbage collection in a distributed system without disjoint address spaces. Other arguments for keeping node address spaces disjoint are based on availability, autonomy, and performance considerations.

We avoid these two drawbacks by implementing call by sharing for REV requests with *call by value-overwrite*, a new argument passing technique that has an efficient implementation in a distributed system. Before presenting the details of call by value-overwrite, we illustrate the basic idea with an example and then list the problems we must solve to have a complete and correct implementation of call by sharing.

Consider an REV request with one argument, an array of integers, and no results. Suppose the request appends a 3 to the high end of the array. Part (a) of Figure 3-3 represents the state of the client and the service before the REV request. The circle represents the array argument to the REV request. Under call by value-overwrite, the client sends a copy of the array to the service. Both the client and the service have a copy of the array, as shown in part (b). The service executes the request and modifies its copy of the array without affecting the client's copy, as shown in part (c). During the REV request, the client must not access its copy of the array, since it may be out of date. At the end of the REV request, the service uses the reply message to send its copy of the array to the client. The client then overwrites its copy with the value sent from the service, as shown in part (d). Once the request completes, if the service does not retain a pointer to its copy of the array, it appears as if call by sharing was used.

The following list presents all of the problems we must solve so that call by value-overwrite implements call by sharing:

1. *Faithful Data Transmission:* The abstract value of an object, rather than its name, is transmitted between nodes. This transmission must not have any visible side effects once the REV request completes. Furthermore, it must preserve sharing within an argument and between arguments.

2. *Argument Modification:* Call by sharing lets a procedure communicate to its caller by modifying its arguments. Call by value-overwrite must also provide this capability.

3. *Argument-Result Sharing:* Call by sharing lets a procedure return some of its arguments as results. Call by value-overwrite must also provide this capability.

4. *Time of Updates:* Updates happen to objects in real time with call by sharing, but updates to arguments are delayed until the end of the request with call by value-overwrite. We must hide this timing difference in the presence of concurrency, node failures, and communication failures.

5. *Disjoint Address Spaces:* Call by value-overwrite keeps separate address spaces disjoint, since one node can not refer to an object at another node. Therefore, we must prohibit programs that would not keep separate address spaces disjoint if call by sharing were fully implemented.

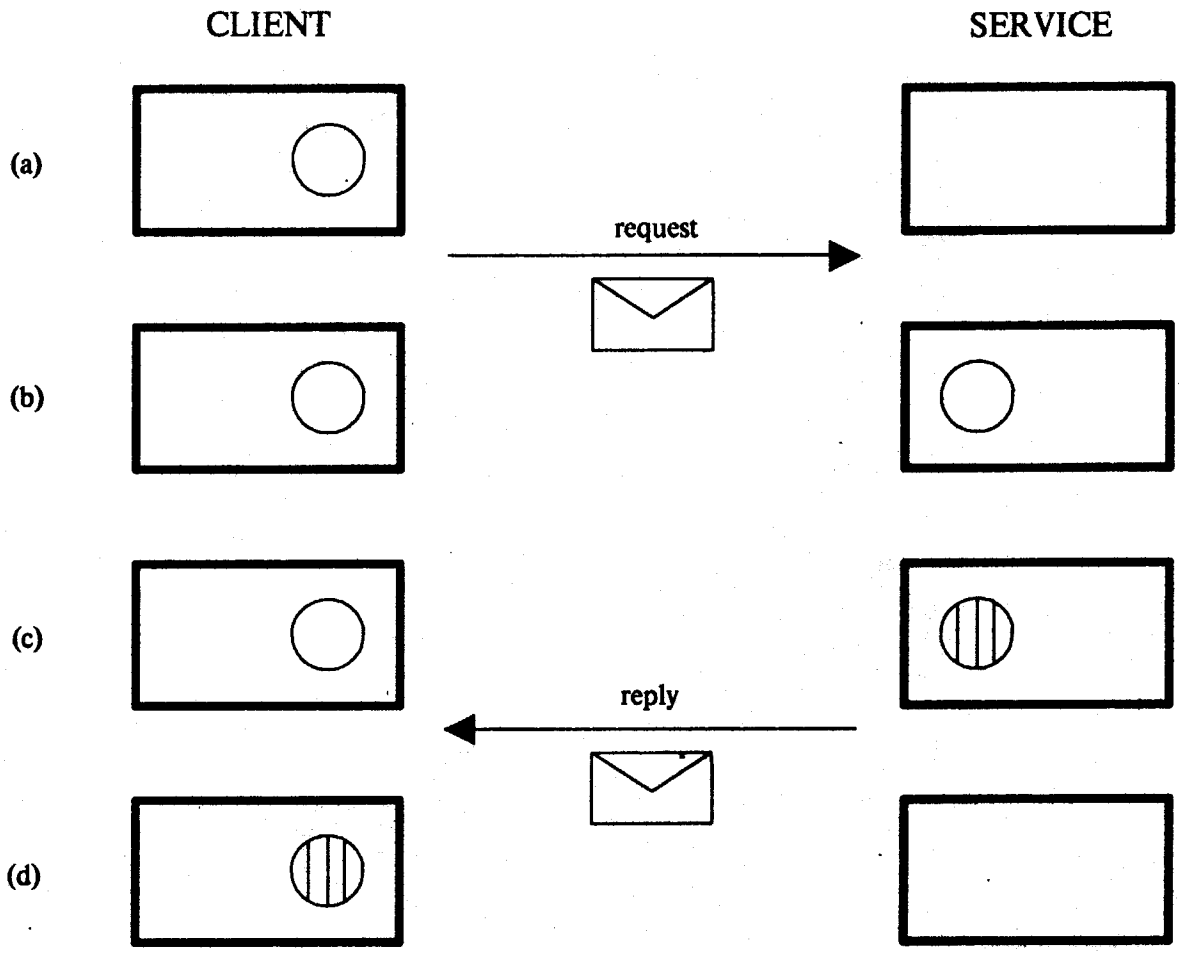


Figure 3-3: Call by value-overwrite.

We will discuss each of the above problems in turn and describe how to solve it. For some problems we augment our algorithm for call by value-overwrite, and for other problems we impose compiler-enforced restrictions on service programmers. Section 3.4.1 presents the details of our call by value-overwrite implementation.

An important consequence of solving these problems is the flexibility we give to the REV mechanism. For each REV request, the REV mechanism can choose whether to use call by value-overwrite or a more traditional implementation of call by sharing. If the client and service are two different nodes, the REV mechanism will use call by value-overwrite. However, if the client and service are the same node, the REV mechanism can *short-circuit* the request; i.e., the client can execute the request as a local procedure and use a traditional, efficient implementation of call by sharing.

Although it seems unlikely that many requests can be short-circuited, our programming methodology

will make such requests common. In the first stage of our programming methodology, the programmer uses REV to fix the execution site of location-dependent procedures. After the program is debugged, the optimizer uses REV to improve performance during the second stage. The optimizer may be the programmer, but someday we hope it will be part of the compiler. The REV requests inserted during second stage will make some of the REV requests inserted during the first stage nested requests. As explained below, these nested requests are candidates for short-circuiting.

Assume a procedure P contains several REV requests and sends them all to the same service. If the service is one of P's arguments, the optimizer can use REV to relocate every invocation of P to the appropriate service. If an REV request that relocates P is location-independent, the new requests are acceptable to the optimizer because they do not change program semantics. In this case the requests inside P become nested requests, and each service that executes P will send these nested requests to itself. Either the compiler or the service's run-time system can short-circuit these requests. The compiler short-circuits a request by removing the "AT . . . EVAL" part of the request, assuming the service expression is side-effect free. The run-time system short-circuits a request by avoiding the communication aspects of the request.

3.2.1 Faithful Data Transmission

Call by value-overwrite requires that arguments and results be sent between the client and the service. Our approach to sending abstract values between nodes is based on Herlihy's template scheme for call by value in a distributed system [18, 19]. Because Section 3.4.1 extends an implementation for call by value into one for call by value-overwrite, at this point we simply review Herlihy's scheme.

A type is *transmissible* if every abstract value of the type may be sent in a message between nodes. The built-in scalar types, such as **integer**, **real**, and **boolean**, are transmissible. The transmissibility of a parameterized type, such as **array[T]**, depends on the transmissibility of the component type T. User-defined types, such lists, sets, and queues, can also be transmissible.

For any transmissible type T, **T$put** converts the abstract value of an instance of T into a transmissible format and appends this information onto a message. In general, **T$put** linearizes an arbitrary graph structure. **T$get** removes information from a message and produces an instance of T. In general, **T$get** converts linear information into an arbitrary graph structure. As the client and service may implement an abstract data type differently, a canonical format is needed to communicate values of a transmissible type. This standard, which is called the *external* representation for the type, must be transmissible.

A programmer implementing a transmissible type plays a small role in making the type transmissible, whereas the system generates much of the code automatically. Let **XT** be the external representation for type T. A programmer implementing T makes it transmissible by implementing two routines that convert between T and **XT**:

```
T$encode = proc (T) returns (XT)
T$decode = proc (XT) returns (T)
```

T$encode maps the concrete representation into the external representation, while **T$decode** does the opposite. Imagine T is an integer set, and XT is an array of integers. If a programmer implements the set as a binary tree, **T$encode** creates an array with the same elements as its one argument, a binary tree. Given an array of integers, **T$decode** creates a binary tree with the same elements. Herlihy's scheme automatically extends **T$encode** and **T$decode** into **T$put** and **T$get**, respectively.

In Herlihy's scheme decoding a cyclic object from its external representation may require lazy evaluation. Lazy evaluation is not essential, since it simply enlarges the set of transmissible abstract values. Moreover, suitably restricting **decode** procedures for types with cyclic values avoids this problem [19]. For these reasons we ignore the difficulties caused by lazy evaluation. This concludes our review of Herlihy's scheme.

When one node transmits the abstract value of an object O to another node, the abstract value of every object accessible from O may also have to be transmitted. We formalize this key fact with the following definition. Let P be a procedure whose execution is relocated by an REV request. The *argument objects* for an invocation of P are all objects forming the arguments to P just before P is invoked. The argument objects include the objects passed to P (i.e., the top-level arguments) as well as all objects accessible from these objects.

We are now ready to tackle the first problem: faithful data transmission. Because we have not introduced the other problems, the worst case we must handle is an REV request executing in the absence of concurrency and failures. The procedure relocated by the request can not modify any of its arguments, and the arguments and results do not overlap. Furthermore, the service can not hold onto any of the arguments or results. Because call by value provides the right semantics under these conditions, we can use Herlihy's scheme. Programmers implementing **encode** and **decode** for transmissible types must ensure these routines have the following properties:

1. they are side-effect free; and
2. they preserve sharing, both within an argument and between arguments.

If these properties hold, the **put** and **get** routines generated by the system will not produce any side effects that are visible once the REV request completes. Furthermore, these system-generated routines will preserve sharing, both within an argument and between arguments. Cyclic structures will also be handled correctly. In other words, these properties provide faithful data transmission.

3.2.2 Argument Modification

The second problem we solve is how to support an REV request that modifies its arguments. Because Section 3.4.1 presents the implementation details, this section describes our solution at a high level

and then illustrates it with an example. In this section only, we assume the request does not return an argument object as a result.

Under call by sharing, a procedure can arbitrarily modify any of its argument objects as long as it adheres to the type system. We support such arbitrary modification by having the service send the following information to the client. At the end of the REV request, the service's run-time system sends the final abstract value of every argument object, and the client uses this information to bring the argument objects at the client up to date. In particular, the client modifies (i.e., overwrites) each argument object with the final abstract value it had at the service. We call this modification a *delayed update*. Because an immutable object can not change its immediate state, only mutable argument objects need a delayed update. Before we discuss how the client knows which abstract value corresponds to which argument object, we pause for an example.

Suppose we use REV to relocate the execution of procedure P, which is shown at the top of Figure 3-4. Assume the argument to P is array A, which is shown in the middle of the figure. Procedure P puts a 4 in IntBox B, creates a new IntBox containing a 5, and then overwrites A[1] with the new IntBox. This detaches IntBox B from array A, as shown at the bottom of the figure. Our objective is to transform the current state of the client, which is shown in the middle of the figure, into the state shown at the bottom of the figure. We will accomplish this by having the service send the client the final abstract value of every mutable argument object.

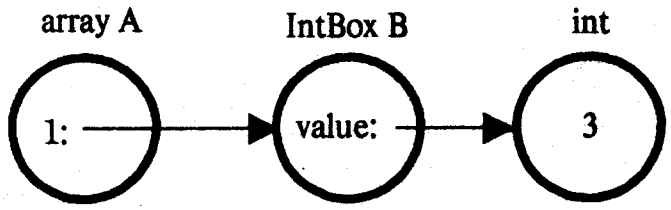
When a service receives an REV request, its run-time system remembers all the argument objects. In this example, the argument objects are the array A, the IntBox B, and the integer 3. The first two are mutable objects, while the third one is immutable. At the end of the REV request, the service sends the explicit results and every mutable argument object back to the client. This example has no explicit results, but it has two mutable argument objects, A and B. Thus the service sends A to the client, which means that B' and 5 are sent. The service also sends B to the client, which means that 4 is sent. Since 3 is immutable and no mutable argument object refers to it directly, the service does not send 3 to the client. The service uses a reply message to send the client the explicit results and the mutable argument objects.

After extracting the explicit results from the reply message, the client's run-time system extracts abstract values from the message and performs delayed updates until the message is empty. In our example, the client extracts the final value of array A and overwrites A with this value. As a consequence, the client creates a new integer, 5, and a new IntBox that refers to the integer. Array A now refers to the new IntBox. Next, the client extracts the final value of IntBox B and performs the delayed update on B. This causes the client to create a new integer, 4. In summary, when the REV request returns, the client overwrites every mutable argument object with the final value it had at the service. This may entail the creation of new objects at the client.

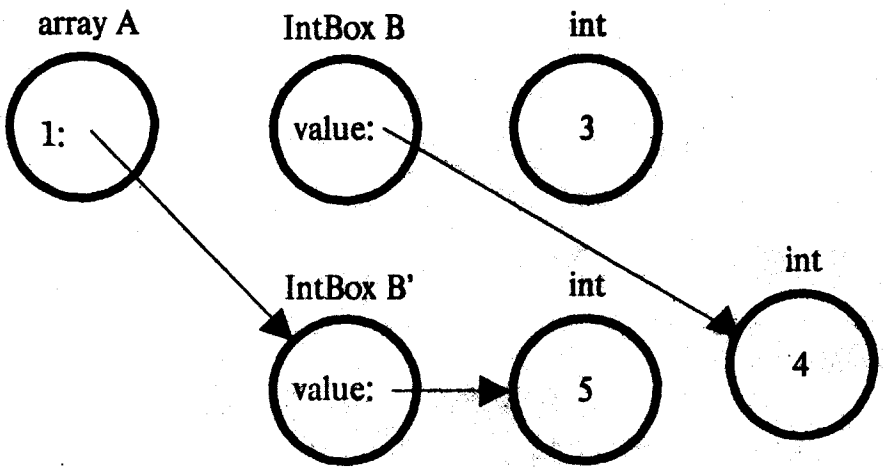
We tag objects so that the client knows which object to overwrite with which value. The client's

```
% two equates

IntBox = record[value: int]
ab = array[IntBox]

P = proc (a: ab)
  a[1].value := 4
  a[1] := IntBox${value: 5}
end P
```

(a) The procedure relocated by an REV request.



(b) The arguments before P is called.



(c) The arguments after P is called.

Figure 3-4: An example illustrating argument modification.

run-time system attaches a unique tag to each argument object it sends to the service, but the service removes the tags before executing the request. During the REV request, both the client and the service remember which tag refers to which object. At the end of the request, the service attaches

the proper tag to every argument object it sends to the client. If the service sends an object that is not an argument object, such as B' in the preceding example, the service attaches a special tag that indicates the object does not exist at the client. When the client extracts such an object from the reply message, the client creates a new object and initializes it to the abstract value sent by the service.

Because we have faithful data transmission between the client and the service, we preserve sharing between objects. This means that neither the client nor the service in the preceding example has more than one copy of an object. The following examples should clarify the point. Consider an REV request in which two argument objects (C and D) refer to another argument object (E) before the request is sent to the service. After receiving the request message, the service creates only one copy of E, and the service copies of C and D both refer to this copy. If the copies of C and D at the service refer to E at the end of the request, the client objects C and D will refer to the same object E after all delayed updates have been performed. If at the end of the request the service copies of C and D refer to object F which is not an argument object, only one copy of F is created at the client, and the client objects C and D will refer to this copy after all the delayed updates have been performed. The client creates only one copy of F, even if the relocated procedure returns F as a result.

3.2.3 Argument-Result Sharing

Our current algorithm for call by value-overwrite already solves our third problem, since it supports an argument object returned as a result. Before arguing this point, we define the result objects for an REV request.

An REV request has two kinds of results: the results computed by the procedure it relocates, and the supplementary information the client uses to perform delayed updates. Because accessibility is important when objects are transmitted between nodes, we define the results of an REV request in terms of accessibility. Let P be a procedure whose execution is relocated by an REV request. The *result objects* for the REV request, which are defined just before P returns, have two sources:

1. *Explicit results:* the results P returns to the caller, including all objects accessible from these results.
2. *Extra results:* all mutable argument objects, including all objects currently accessible from these objects.

An REV request can have either, neither, or both kinds of results. For instance, the REV request in the previous section had extra results but not explicit results. The explicit results and extra results can overlap, since an explicit result object and an extra result object can refer to the same object. Until now we have assumed that no argument object was also an explicit result object.

Our current call by value-overwrite algorithm consists of delayed updates, tags for transmitted objects, and faithful data transmission. The delayed updates reflect modifications to the argument objects done at the service, and the tags are used by the client to match extra results with argument

objects. Faithful data transmission preserves sharing between argument objects. It also preserves sharing between all result objects.

Suppose we introduce the third problem by letting an argument object be an explicit result object. This does not affect faithful data transmission, since it already preserves sharing between all result objects. The client's run-time system must expect tags on explicit result objects in addition to tags on extra result objects. Once this minor change is made, an argument object returned as an explicit result object is identified with the original client object; i.e., we preserve sharing between arguments and results.

3.2.4 Time of Updates

With call by sharing, changes to an argument object happen in real time, but with call by value-overwrite, changes to an argument object (i.e., delayed updates) happen only when the procedure returns. An observer at the client able to view argument objects during the execution of a procedure could distinguish these two argument passing techniques. For instance, a node or communication failure could expose an intermediate state in a computation that would show the difference between call by value-overwrite and call by sharing. Similarly, another client process examining argument objects during an REV request might notice the difference between call by value-overwrite and call by sharing.

Our transaction mechanism prevents these activities by masking concurrency and by masking node and communication failures. Each REV request must run as part of some (sub)transaction, and we assume there is no concurrency within a (sub)transaction. Since we assume every object is atomic, it does not matter whether changes to an argument object happen in real time or at the end of the REV request. The changes are visible to other processes only if the (sub)transaction commits, which can happen only after the REV request successfully completes.

3.2.5 Disjoint Address Spaces

Call by value-overwrite keeps separate address spaces disjoint, but call by sharing does not. Because we want separate address spaces to be disjoint for the reasons mentioned earlier, we leave our call by value-overwrite algorithm alone and restrict service programmers. This section provides linguistic support that lets the compiler ensure service programmers obey the restrictions we impose. The restrictions, which are invisible to the application programmer, solve our fifth and final problem: how to keep separate address spaces disjoint while still providing call by sharing semantics. Because the restrictions will force service programmers to copy certain objects, we begin by discussing how much copying call by value-overwrite does.

Call by value-overwrite copies objects while a straightforward implementation of call by sharing does not. For example, call by value-overwrite maintains two copies of an argument object during an REV

request: the original stays at the client, while a copy is sent to the service. Call by value-overwrite also creates two copies of a result object that is not an argument object. The original stays at the service, while a copy is sent to the client. A straightforward implementation of call by sharing, in contrast, does not copy argument objects or result objects.

Since we want to implement call by sharing with call by value-overwrite, we must hide the fact that call by value-overwrite makes extra copies of objects. We do this by ensuring that the copies at the service are inaccessible to the service after the REV request completes. Our scheme is based on colored objects. We will use colors to motivate our restrictions on service programmers and to argue that our restrictions work.

In our scheme, client objects are red, and service objects are blue. We explain below how the compiler ensures that the result objects of an REV request are red. This means the client has only red objects at the end of the request. The compiler also ensures that all accessible objects at the service are blue at the end of the REV request. Since objects have only one color, the compiler-enforced coloring prevents the service from referring to a mutable argument or result object once the REV request completes. Thus the client and service address spaces are disjoint at the end of the REV request, even if a straightforward implementation of call by sharing is used. This section supplies linguistic support that lets the service programmer use colors. Application programmers, in contrast, never deal with colors.

A *color* is a static attribute of a mutable type. Each mutable object has only one color for its entire existence. Because procedures and iterators can refer to own variables, we consider them mutable and give them colors. Since a program can not tell the difference between an immutable object and a copy of the object, the service can refer to immutable argument and result objects without showing the difference between call by sharing and call by value-overwrite. For this reason, immutable types have no color and are irrelevant to this discussion.

Our syntax for colors was designed to be unobtrusive. As mentioned above, client objects are red and service objects are blue. Client variables are red; service variables are either red or blue. For any mutable type T, T! represents the same type with the color blue. T represents the type with some color. This color may also be blue, but for strong type checking we assume it is red. For example, the following code fragment creates a red integer array and a blue integer array:

```
redArray: array[int] := array[int]$new()
blueArray: array[int]! := array[int]!$new()
```

Strong type checking prevents the service programmer from assigning *redArray* to *blueArray* and vice versa.

Service routines must copy certain arguments and results to keep the client and service address spaces disjoint between REV requests. We assume any mutable type T with a copy procedure (T$copy) automatically provides the following two procedures:

```
T$red_to_blue = proc (T) returns (T!)
  % returns a blue copy of a red object

T$blue_to_red = proc (T!) returns (T)
  % returns a red copy of a blue object
```

The implementation for each of these procedures is **T$copy**.

Figure 3-5 contains part of an implementation for a remote bulletin board. Note that the arguments and results of **post** and **retrieve** are either immutable or assumed to be red. The color annotation "!" distributes over parameterized types. For example, if **foo** and **x** are types,

```
array[foo[x]]!
```

is equivalent to

```
array[foo[x!]!]!.
```

Post copies a notice before installing it in the bulletin board. Similarly, **retrieve** copies all relevant notices before returning them to the client. Note that erasing the color annotations yields CLU code that does not violate the type system.

```
notice = record[sender: string,          % an equate
                time: time,
                expiration: time,
                categories: set[string],
                message: string]

notices: set[notice]! := set[notice]!$create()  % a blue own variable

post = proc (info: notice)
  serviceCopy: notice! := notice$red_to_blue(info)
  set[notice]!$insert(notices, serviceCopy)
end post

retrieve = proc (keyword: string) returns (set[notice])
  answer: set[notice] := set[notice]$create()
  for n: notice! in set[notice]!$elements(notices) do
    if set[string]!$isIn(n.categories, keyword)
    then
      clientCopy: notice := notice$blue_to_red(n)
      set[notice]$insert(answer, clientCopy)
    end
  end
  return(answer)
end retrieve
```

Figure 3-5: Using colors in service routines to keep separate address spaces disjoint.

We say the specification for a routine is *uniform* if it has no color annotations. For example, the routines **post** and **retrieve** in Figure 3-5 have uniform specifications. We say an implementation for a routine is *uniform* if it has no color annotations. The implementations for **post** and **retrieve** are not uniform because they contain blue annotations.

A routine with a uniform specification may be applied to either blue arguments or red arguments: the presence or absence of "!" in an invocation lets the compiler perform strong type checking. Assume **T$P** is a service routine with a uniform specification. **T!$P**, which may be applied only to blue arguments, has blue results. **T$P**, which may be applied only to red arguments, has red results. For instance, assume **post** is defined in the **bboard** interface. Then **bboard$post** can be applied to a client notice, while **bboard!$post** can be applied to a service notice. In both cases a copy of the notice is posted on the bulletin board.

Using call by value-overwrite to implement call by sharing forces us to ensure the service does not refer to a mutable argument or result object once the REV request completes. We argue below that the following rules ensure the client and service address spaces are disjoint between REV requests:[4]

1. Each type constructor (e.g., **record** and **array**) has a uniform specification.

2. Each abstraction primitive (e.g., **up** and **down** in CLU) has a uniform specification.

3. Each routine exported by a service has a uniform specification.

4. All own variables at the service are blue.

5. Service processes communicate with each other by using blue objects.

The compiler checks the preceding rules and performs strong type checking. Each mutable type is now an ordered pair consisting of a conventional type and a color.

To simplify our argument that these rules ensure the client and service address spaces are disjoint at the end of an REV request, we use the following invariant:

- *All objects are monochromatic.* An object is *monochromatic* if every object it directly or indirectly refers to has its color. Hence a red object may refer to only red objects. A blue object may refer to only blue objects. This invariant implies that the colors of the concrete representation and abstract value of an object are the same. A red object can not masquerade as a blue object and vice versa.

This monochromatic invariant follows from strong type checking, uniform type constructors (rule # 1), and uniform abstraction primitives (rule # 2).

We claim strong type checking, the monochromatic invariant, and the last three compiler-enforced rules together imply the client receives no blue objects from the service and the service has no client objects at the end of an REV request. Assume the client contains only red objects and the service contains only blue objects before an REV request occurs. An REV request has two categories of result objects, and we show that each category contains only red objects:

1. *Explicit results:* Each REV request is uniform. It deals exclusively with red types, since

[4] When closures are first-class objects, another rule is needed to guarantee the disjointness of client and service address spaces. Any service closure whose lifetime can extend past the completion of an REV request must access only blue variables.

the compiler chooses the "red" version for each routine imported by the REV request. The red version exists because every routine exported by the service has a uniform specification (rule #3). The monochromatic invariant and strong type checking imply the explicit results of an REV request consist of only red objects.

2. *Extra results:* These results consist of mutable argument objects and all objects accessible from them. Mutable argument objects come from the client, which by assumption contains only red objects. The monochromatic invariant implies that all objects accessible from mutable argument objects are also red. Hence the extra results are red objects.

The client begins an REV request with only red objects and receives only red objects from the service. Therefore, the client ends an REV request with only red objects. Our requirements that all own variables and interprocess communication paths at a service be blue (rules #4 & #5), coupled with the monochromatic invariant and strong type checking, prevent the service from keeping any red argument or result objects past the completion of the REV request. The remaining objects at the service are blue. Therefore, at the end of an REV request, the service and client address spaces are disjoint.

Implementing call by sharing with call by value-overwrite requires the cooperation of service programmers, who must copy mutable objects logically sent between address spaces. This section outlined rules based on colors that indicate when service programmers must copy objects. The linguistic support we provided lets the compiler enforce these rules. The application programmer, in contrast, does not worry about colors.[5]

This division of labor is similar to the way we split the responsibility for ensuring a valid REV request is location-independent. In both cases, service programmers follow certain rules, and their efforts are checked by the compiler. Unlike service programmers, application programmers do not use colors or LOCATION_INDEPENDENT attributes.

3.2.6 Discussion

We can avoid the complexity of call by value-overwrite and still retain identical argument-passing semantics for local procedures and REV requests if we require that all arguments and results of an REV request be immutable. Then we can implement call by sharing for REV requests with call by value, the traditional semantics for RPC's. We rejected this alternative because we felt it would overly constrain a programmer or (automatic) optimizer using REV to relocate processing.

The relationship between call by value-overwrite and call by sharing is similar to the relationship between call by value-result [16] and call by reference. In the absence of aliasing in the programming

[5] If we let an application programmer send a nested REV request to the client, client routines would have to obey the coloring rules, since the client is acting as a service. However, the client cannot receive a nested request, because the application programmer can not name the client, let alone send an REV request to it. We shall return to this point in Chapter 5.

language and concurrency at the processor, call by value-result and call by reference have identical semantics for local procedure calls [8]. A stronger statement applies to call by value-overwrite, since it implements call by sharing for local procedure calls even in the presence of aliasing.

We used colors and programming rules to make service copies of client objects inaccessible at the end of an REV request. This technique, which let us implement uniform argument-passing semantics for local procedures and REV requests, also keeps separate address spaces disjoint. NIL [37], a language for distributed programming designed and implemented at the IBM T. J. Watson Research Center, also provides uniformity and disjointness, but it does so in a different way. NIL completely avoids aliasing by disallowing the notions of pointers and shared data. An object assigned from one variable to another variable can not be accessed from the first variable, which becomes uninitialized. Similarly, an object transferred from one process to another process can not be accessed by the first process. The compiler enforces this viewpoint by doing *typestate checking*. This keeps the address spaces of different nodes disjoint and provides uniform interprocess communication.

Since NIL processes can not share data, information may be communicated between processes by reference or by value-result. Thus the NIL implementation can choose one technique when the processes share the same address space and another technique when the processes exist on different nodes. A similar option is available to an REV mechanism. If the client and service are different nodes, the mechanism uses call by value-overwrite. If the client and service are the same node, the REV mechanism can short-circuit the request and use a conventional implementation for call by sharing.

3.2.7 Summary

We implement call by sharing for REV requests with call by value-overwrite. At the end of a request, the service sends the client all mutable argument objects. The client then overwrites existing objects with their new values. We assume that programmers correctly implement **encode** and **decode** for each transmissible type; i.e., we assume faithful data transmission. Given this assumption, call by value-overwrite implements call by sharing for REV requests if each REV request meets the following requirements:

1. every argument is atomic;
2. every argument type and result type is transmissible;
3. the client and service address spaces are disjoint between REV requests;
4. the REV request runs as (part of) an atomic transaction that aborts if the REV request does not complete.

The first requirement is automatically met, as every type is assumed to be atomic. The compiler checks the second and third requirements, and the run-time system checks the final requirement. We provided linguistic support that lets the compiler ensure service routines keep separate address spaces disjoint. This linguistic support affects service programmers but not application programmers.

3.3 Compile-time Tasks

Having explained how we use call by value-overwrite to implement call by sharing in a distributed system, we now explain how to implement REV. This section considers the compile-time tasks of an REV mechanism: static checking of REV requests and stub generation. The next section discusses the run-time tasks. In both sections, we highlight the differences between an REV mechanism and an RPC mechanism.

3.3.1 Static Checking of REV Requests

Compared to run-time checking, compile-time checking can detect errors earlier and improve run-time performance. Besides having the compiler perform strong type checking, we want the compiler to verify the validity of REV requests. Recall that a valid REV request is one the service can execute and the compiler can encode. If a service is unable to execute an REV request, we want to notify the application programmer of this problem at compile-time. Encoding an REV request once at compile time, rather than encoding it each time it is executed, can improve performance. We also want the compiler to determine the location independence of REV requests. This information lets an optimizer know whether an REV request changes program semantics. Service definitions give the compiler enough information to determine the validity and location independence of REV requests. We first consider service definitions and then consider the static checking of REV requests.

A compiler supporting RPC's processes interface definitions, while a compiler supporting REV processes both interface and service definitions. Compiling a service definition, which indirectly lists a set of interfaces, is not a difficult task. MESA [32], a systems programming language developed at Xerox PARC, enforces strong type checking and supports *configurations*. A configuration is a collection of interfaces, only some of which are exported. Configurations and services are analogous: a configuration is defined in terms of interfaces and other configurations, whereas a service is defined in terms of interfaces and other services. Techniques for compiling a configuration definition are applicable to compiling a service definition.

The compiler uses a service definition to determine the code portion, validity, and location independence of an REV request. The previous chapter outlined how the compiler generates part of the program call graph to determine the code portion of an REV request. The compiler checks the validity of the request by ensuring that every imported routine is exported by the service. Recall that a routine in the REV request that is executed at the service but not exported by the service can be sent with the request. Checking the validity of a request is straightforward when the request has no parameterized types. Since type parameters must be types exported by the service, the compiled service definition may further constrain the type parameters appearing in a parameterized request. In any event, the request is invalid if the code portion contains an own variable or if a procedure argument to the request is unknown at link time. The latter restriction minimizes the changes to the run-time support for the programming language, since we never need to check the validity of an REV

request or encode its code portion at run time. Checking whether a valid request is location-independent is straightforward: every imported routine must be declared location-independent.

If separate compilation is used, supporting client-supplied routines and code arguments means that the static checking of REV requests might have to be deferred until link time. The code portion of an REV request, and hence its validity and location independence, can depend on the implementation of the client-supplied routines and code arguments. Under separate compilation, the implementations may not be known until link time. While the type specifications for these routines are known at compile time, type specifications may not provide enough information to determine the code portion of the REV request.

Although client-supplied routines and code arguments complicate static checking and encoding of REV requests, they have two advantages. First, they increase the number of ways in which a program may be partitioned into components for local and remote execution. The net effect on performance depends on the tradeoff between fewer but larger REV requests. Second, a powerful feature of many programming languages is the use of routines as arguments to other routines. It may be desirable for some service routines and REV requests to take client-supplied routines as arguments.

3.3.2 Stub Generation

Besides performing static checking, a compiler supporting REV generates stubs. A *stub* is a procedure that interfaces an REV request with the communication primitives. Our approach, which is based on Nelson's RPC mechanism [34], hides the communication details of REV requests from both the application programmer and the service programmer. For each REV request, the compiler generates two simple procedures called stubs, as shown in Figure 3-6. One stub, which is located at the client, lies between the application program and the client communication package. The compiler replaces the REV request with a call to this stub. The other stub, which is sent to the service with the REV request, lies between the service communication package and the procedure whose execution is relocated by the REV request.

Consider an REV request that relocates the execution of a procedure (P) that is not exported by the service. Furthermore, assume there are no client-supplied routines in the request. Hence, the code portion consists of the body of P. The calling sequence for such an REV request is as follows. The application program calls the client stub, which is an ordinary procedure. Besides P's arguments, the client stub has an argument that denotes the node that will execute the REV request. The client stub creates a message containing the service stub, the procedure P, and the arguments to the request. The client stub calls the client communication package, which reliably sends the request message to the service as a sequence of network packets. The communication package is responsible for routing, retransmissions, and acknowledgements. The service communication package reconstructs the request message from the packets it receives and then extracts the service stub and P from the request message. The service communication package calls the service stub with a single argument:

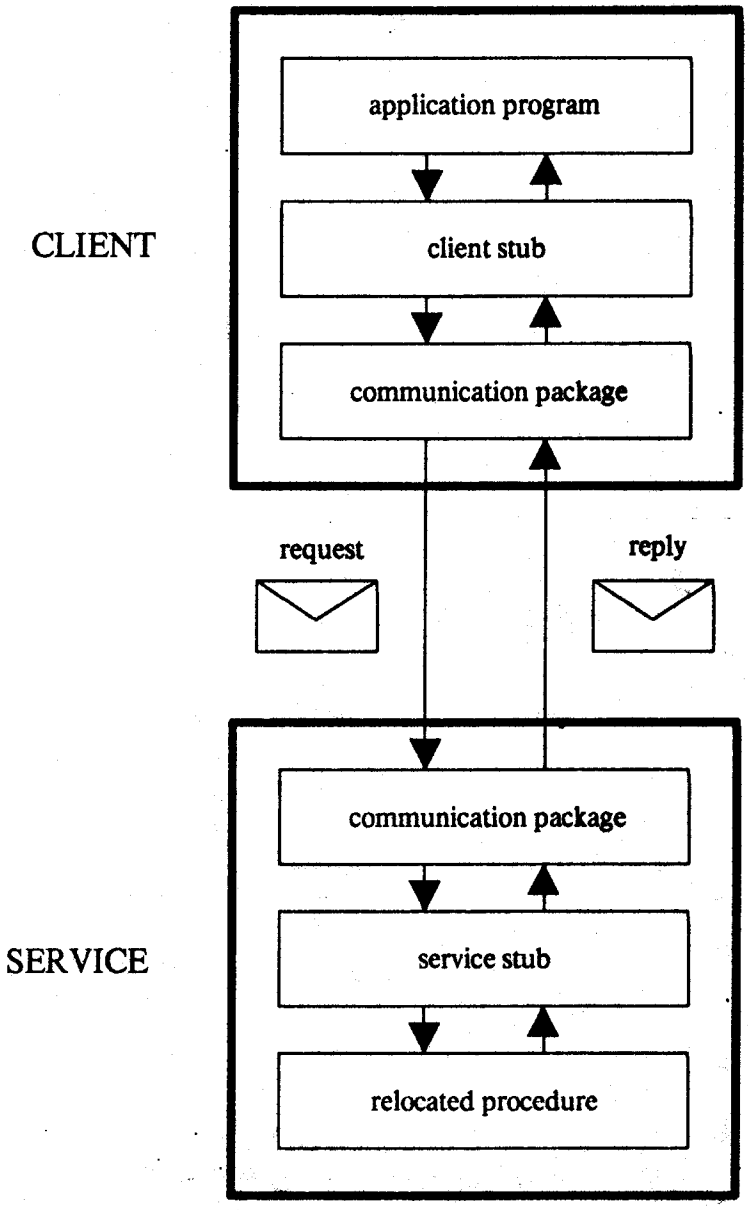


Figure 3-6: A stub-based implementation of REV.

the remainder of the request message. The service stub extracts the arguments from the request message and then calls P with the arguments. When P returns, the service stub creates a new message and inserts P's results in the message. The service stub returns the reply message to the service communication package, which reliably sends the message to the client communication package as a sequence of packets. The client communication package reconstructs the message from the packets it receives and returns the entire reply message to the client stub. The client stub extracts the results from the reply message and returns them to the application program. Although the client stub appears to be a local procedure to the application program, in reality it represents an REV request.

There are three differences between RPC stubs and REV stubs. First, an RPC stub transmits a procedure name while an REV stub can transmit one or more procedures. Second, an RPC service stub exists at the service while an REV service stub is sent in the request message. Finally, an RPC stub typically implements call by value while an REV stub implements call by value-overwrite.

Before showing an example of REV stubs, we present an abstract data type called **REVcontext** (Figure 3-7). An REV context, which may contain a request message or a reply message, hides the details of encoding and decoding data. The reader should use this figure as a reference during the rest of this section. Recall that for any transmissible type T, **T$put** converts the abstract value of an instance of T into a transmissible format and appends this information onto a message. **T$get** does the opposite. It removes information from a message and produces an instance of T.

We use a simple example to show how REV works as well as the relationship between the stubs and the original REV request. Although in this example we neglect exceptions raised by the relocated procedure, REV stubs can accommodate exceptions in the same way that RPC stubs accommodate exceptions. The procedure at the top of Figure 3-8 contains an REV request, and the relocated procedure is shown at the bottom of the figure. The client executes the code in Figure 3-9, which contains **SomeProc** and the client stub (**G1991**). The name of the client stub is irrelevant as long as it is unique in the current environment. Note that the REV request in **SomeProc** has been transformed into an ordinary procedure call that invokes the client stub. As mentioned earlier, compared to procedure P the client stub has one additional argument, the node that executes the REV request. The client stub has three tasks: prepare the request message; perform the REV request; and finally extract the results from the reply message. We discuss each of these tasks in turn.

The first part of the client stub prepares the request message. After creating and initializing an REV context, the first part of the client stub inserts the service stub (**G1992**) and the arguments (*a*, *b*, and *c*) into the request message. As shown below, the service stub invokes the relocated procedure P. Hence inserting the service stub into the request message also inserts P into the request message. Section 3.4.2 discusses how procedures are inserted into messages. While an REV client stub inserts the service stub and the relocated procedure into the request message, an RPC client stub inserts only the *name* of the remote procedure into the request message.

```
REVcontext = interface is

  % CLIENT ROUTINES

  new[s:service] = proc (destination: s) returns (REVcontext)
  % create an REVcontext with an empty request message and two empty mappings

  send = proc (r: REVcontext)
  % send the request message to the service, discard the old
  % mapping not needed for reply phase, and create new mapping

  getMutableArgs = proc (r: REVcontext)
  % extract the remaining mutable argument objects from the
  % reply message and perform the delayed updates


  % SERVICE ROUTINES

  process = proc (m: message) returns (REVcontext)
  % return a new REV context with the supplied request message and two empty mappings

  apply = proc (r: REVcontext)
  % extract the service stub from the request message and invoke it on the REV context

  prepareForReply = proc (r: REVcontext)
  % discard the request message and the old mapping not needed for the
  % reply phase, create a new mapping, and create an empty reply message

  putMutableArgs = proc (r: REVcontext)
  % encode all mutable arguments not already in the reply message

  reply = proc (r: REVcontext)
  % send the reply message to the client

  abort = proc (r: REVcontext)
  % terminate an REV request


  % ROUTINES FOR BOTH THE CLIENT AND SERVICE

    .
    .   % Routines that support get and put
    .

end REVcontext
```

Figure 3-7: The abstract data type **REVcontext**.

The second part of the client stub performs the REV request by calling the client communication package, which sends the request message to the appropriate service, periodically retransmits the request message, and waits for a reply message. An REV request normally completes when the client receives a reply message, in which case the client communication package places the reply message in the REV context and the client executes the third part of the client stub. On the other hand, if the client communication package does not receive a reply message and can not communicate with the

```
ai = array[int]   % an equate

SomeProc = proc (a: ai, b: int, c: ai) returns (int)
  begin transaction
    aService: built-ins := Service[built-ins]$Any()
    d: int := at aService eval P(a, b, c)
    return(d*d)
  end
end SomeProc

P = proc (a: ai, b: int, c: ai) returns (int)
  a[b] := c[b]
  a := ai$new()
  b := b+1
  return(b)
end P
```

Figure 3-8: A simple REV request.

service communication package, the client communication package may unilaterally terminate the REV request by aborting the current transaction and then raising the exception **failure**.

If the REV request completes normally, the client executes the third and final part of the client stub. This part of the client stub extracts the explicit results from the reply message, extracts the extra results, and then returns the explicit results. The client stub calls the appropriate **get** routine for each explicit result. **GetMutableArgs** calls the appropriate **get** routine for each extra result, as explained in Section 3.4.1. When any **get** routine extracts an argument object from the reply message, the **get** routine performs the delayed update on the object. Section 3.4.1 again provides the details. In this example, the explicit result is an integer assigned to *d*. The extra results are the two arrays originally bound to *a* and *c*. An RPC client stub implementing call by value, in contrast, deals with only explicit results and does not perform delayed updates.

If the relocated procedure is not exported by the service, we want to emphasize that we send at least two procedures with an REV request: the relocated procedure (**P** in this example) and the service stub (**G1992** in this example). We found that sending two procedures was a natural way to implement REV.

When a service communication package receives a request message, it creates a new REV context with **REVcontext$process** and calls **REVcontext$apply**. **REVcontext$apply**, which is shown in Figure 3-10, extracts the service stub and hence the relocated procedure from the request message. We discuss how procedures are extracted from messages in Section 3.4.2. **REVcontext$apply** then applies the service stub to the REV context. As explained below, the service stub extracts the arguments from the request message, invokes the relocated procedure P, and inserts the results into a new reply message. Once the service stub has completed, **REVcontext$apply** returns. The service communication package then sends the reply message to the client. If an encode/decode

```
ai = array[int]   % an equate

SomeProc = proc (a: ai, b: int, c: ai) returns (int)
  begin transaction
    aService: built-ins := Service[built-ins]$Any()
    d: int := G1991(aService, a, b, c)
    return(d*d)
  end
end SomeProc

G1991 = proc (aService: built-ins, a: ai, b: int, c: ai) returns (int)

  % the client stub

  begin

    % part 1: prepare the request message
    REVcode = proctype (REVcontext)     % an equate
    rev: REVcontext := REVcontext$new[built-ins](aService)
    code[REVcode]$put(rev, G1992)
    ai$put(rev, a)
    int$put(rev, b)
    ai$put(rev, c)

    % part 2: perform the REV request
    REVcontext$send(rev) abort resignal failure

    % part 3: extract the results
    d: int := int$get(rev)              % explicit result
    REVcontext$getMutableArgs(rev)      % extra results
      % delayed updates for a and c
    return(d)

  end except when others (s: string):
    abort signal failure(s) end

end G1991
```

Figure 3-9: The client code for Figure 3-8.

---

exception occurs, **REVcontext$abort** resets the REV context and places a distinguished error value in a new reply message, which causes **REVcontext$send** at the client to raise the exception **failure**("encode/decode problem"). The client stub aborts the current transaction and then resignals the exception. No result objects are sent to the client when an encode/decode exception occurs at the service. Note that **REVcontext$abort** must not raise an exception, as there is no handler in **REVcontext$apply** that could catch the exception.

Our implementation of **REVcontext$apply** requires that service stubs have the following type specification:

```
proc (rev: REVcontext) returns ().
```

Each service stub extracts arguments from an REV context and inserts results into the same REV context. This lets **REVcontext$apply** have a simple implementation that does not violate the type system.

```
apply = proc (rev: REVcontext) returns ()
  begin
    REVcode = proctype (REVcontext)      % an equate
    serviceStub: REVcode := code[REVcode]$get(rev)
    serviceStub(rev)
  end except when others: REVcontext$abort(rev) end
end apply
```

Figure 3-10: The implementation of **REVcontext$apply**.

Figure 3-11 contains the service stub, which has three functions. It extracts the arguments from the request message, invokes the relocated procedure, and then inserts the results into a reply message. An RPC service stub differs from an REV service stub in that an RPC stub returns explicit results but not extra results to the client. Section 3.4.1, which explains how to implement call by value-overwrite, explains how to implement **REVcontext$putMutableArgs** and **REVcontext$getMutableArgs**.

```
ai = array[int]

G1992 = proc (rev: REVcontext) returns ()

  % part 1: extract the arguments
  a: ai := ai$get(rev)
  b: int := int$get(rev)
  c: ai := ai$get(rev)

  % part 2: invoke the relocated procedure P, which
  % is encoded, transmitted, and decoded with G1992
  b := P(a, b, c)

  % part 3: insert the results
  REVcontext$PrepareForReply(rev)
  int$put(rev, b)                  % explicit result
  REVcontext$putMutableArgs(rev)   % extra results
    % encodes the current state of the
    % original arguments a and c

end G1992
```

Figure 3-11: The service stub for Figure 3-8.

For pedagogical purposes, no optimization was performed on the stubs in this section. For high performance, the abstract data type **REVcontext** should be replaced by its implementation, and simple put and get procedures should be expanded in-line.

3.4 Run-time Tasks

Having discussed the compile-time tasks of an REV mechanism, we turn our attention to the run-time tasks. A client REV mechanism must support call by value-overwrite, transmit procedures between nodes, bind to remote services, provide reliable communication, and recover from failures. A service REV mechanism, in addition, must evaluate the REV requests it receives. While discussing these tasks, we emphasize those aspects of our implementation that are novel.

3.4.1 Call by Value-Overwrite

Our implementation of call by value-overwrite is an extension of Herlihy's template scheme for call by value in a distributed system [18, 19], which we discussed in Section 3.2.1. We present an implementation of Herlihy's scheme, extend it to implement call by value-overwrite, and then offer some possible optimizations.

3.4.1.1 Implementing Call by Value

Under call by value, the client inserts each argument into a message using the appropriate **put** routine. Then the client sends the request message to the service. The service extracts each argument from the request message using the appropriate **get** routine. When the remote invocation completes, the service inserts each result into a new message using the appropriate **put** routine and sends the reply message to the client. The client extracts each result from the reply message using the appropriate **get** routine. Since the types of the arguments and results of the remote procedure are known to both the client and the service, both know which **put** and **get** routines to use. As we saw earlier, **put** and **get** routines play an important role in client and service stubs.

Get and **put** routines coordinate their activity to preserve sharing within an object, between arguments, and between results. Figure 3-12 shows some of the data structures in Herlihy's scheme for call by value. Each data structure in the figure, which we call a *mapping*, relates objects to message positions or vice versa. The mappings and the figure are explained in the following discussion, which assumes that **T** is a transmissible abstract data type and **XT** is its external representation type. For instance, **T** could be a set of integers implemented by a binary tree, and **XT** could be an array of integers.

An implementation for call by value often keeps a mapping from objects to positions in a message during an encode phase. Mappings **A** and **D** in Figure 3-12 are examples of this kind of mapping, which is used to detect and preserve sharing and cycles. Although the rest of this paragraph refers only to the client and its mapping (**A**), the discussion applies equally well to the service and its mapping (**D**). Before the client encodes any objects, mapping **A** is empty. **T$put** uses **A** to decide whether an object has been inserted into the message, as an object appears in **A** if and only if it has been (or is being) inserted into the message. When **T$put** encounters an object that has not been

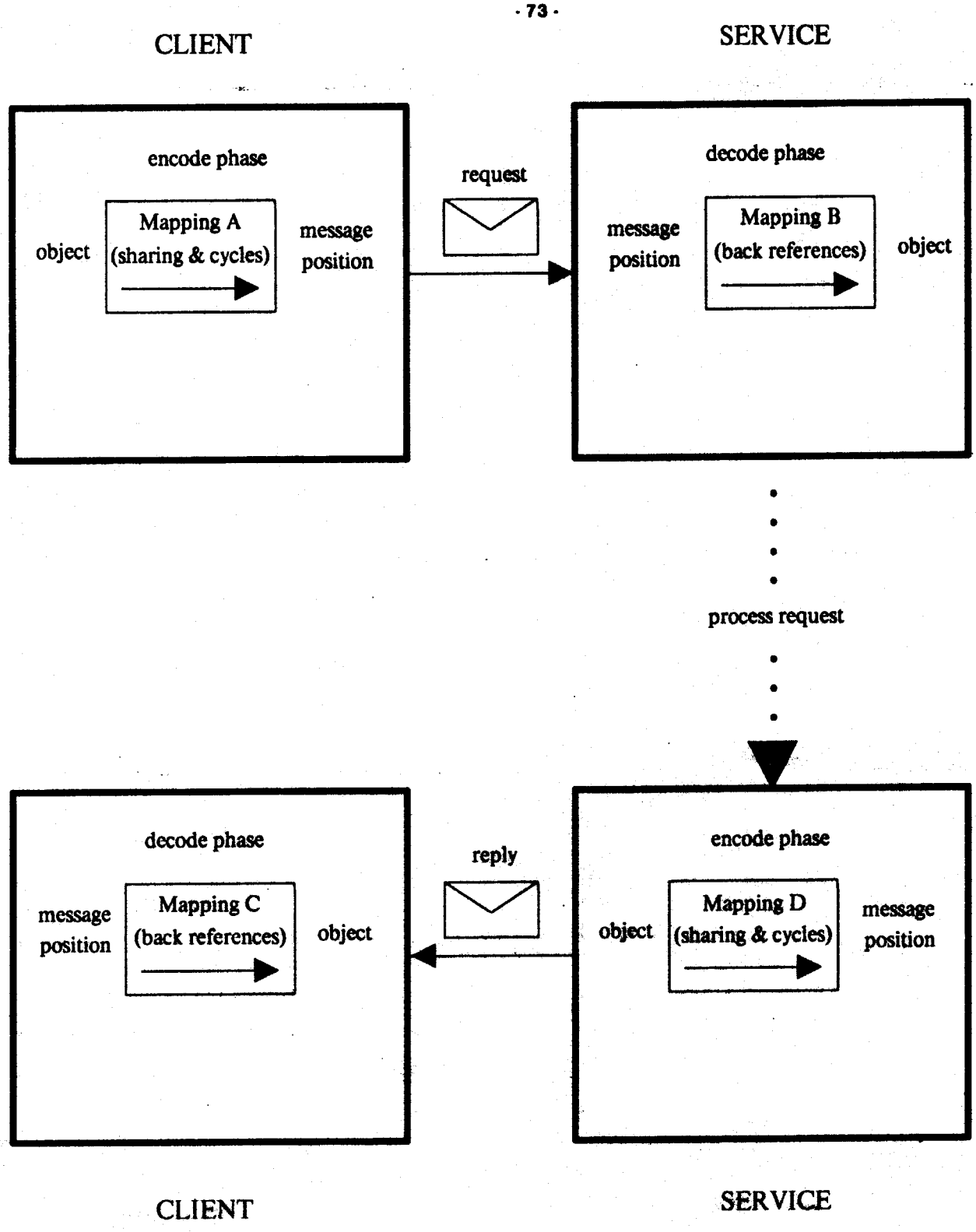


Figure 3-12: An implementation for call by value.

inserted into a message, it inserts the object's name and the current message position into A. After converting the object to its external representation (XT) with **T$encode**, **T$put** calls **XT$put**. **XT$put** inserts the object's abstract value into the message in its canonical format. When **T$put** encounters an object that has already been inserted into the message, it does not change mapping A. **T$put** inserts a *back reference* into the message instead of calling **T$encode** and **XT$put**. The back reference contains the message position of the object, which is determined from A. The client constructs A during its encode phase and then discards A.

The reverse mapping, which maps message positions to objects, is constructed during a decode phase and then discarded. Mappings B and C in Figure 3-12 are examples of this kind of mapping, which is used to resolve back references. Although the rest of this paragraph refers only to the service and its mapping (B), the discussion applies equally well to the client and its mapping (C). When **T$get** encounters a back reference, it consults mapping B and returns the appropriate object without doing any decoding. When **T$get** encounters an encoded object, it inserts the future name of the object and the current message position into B. **T$get** then calls **XT$get**, which returns the abstract value in its canonical format. **T$get** uses **T$decode** to convert the canonical format into the node's format for an object of type T.

Note the symmetry between the mappings for a request message (A and B) and the mappings for the reply message (D and C) in Figure 3-12. The encode phase, whether it is at the client or at the service, requires a mapping from objects to message positions. This mapping is used to detect and preserve sharing and cycles. The decode phase, whether it is at the client or at the service, requires the reverse mapping, which is used to resolve back references. Under call by value, the client and service together use four mappings for each REV request.

3.4.1.2 Implementing Call by Value-Overwrite

Herlihy's template scheme can be extended to support call by value-overwrite for user-defined, abstract data types. Since we want call by value-overwrite to implement call by sharing, for each transmissible type T we assume that **T$encode** and **T$decode** have no side effects. Furthermore, we assume they preserve sharing within an argument and between arguments. A programmer implementing **T$encode** and **T$decode** does not need to know whether call by value or call by value-overwrite will be supported. **T$put** and **T$get**, which are automatically generated, determine the argument passing semantics. We explain below how our versions of **T$put** and **T$get** differ from Herlihy's versions.

We tag transmitted objects so that the client knows which abstract value sent from the service corresponds to which argument object at the client. The *external name* for an argument object is its position in the request message. Result objects that are not argument objects have the same external name: -1, an invalid message position. External names appear explicitly in a reply message, as each object in a reply message is preceded by its external name. External names do not appear explicitly

in a request message, since the service determines the message position and hence the external name for each argument object.

Figure 3-13 contains some of the data structures in our implementation for call by value-overwrite. Since mappings A through D play exactly the same role as they did in Herlihy's implementation for call by value, we concentrate on the two new mappings, E and F. Both the client (E) and the service (F) maintain a mapping involving external names and objects. These two mappings, which are created and written during the request phase and read during the reply phase, exist for the duration of an REV request. Once the request has completed, the mappings are discarded. Mapping E maps external names to pairs of the form **<object, T$get>**, where an object with type T is paired with **T$get**. We explain below how the **get** routines are placed in E and why they are needed. Mapping F maps objects to pairs of the form **<external name, T$put>**. The routine for an object with type T is **T$put**. As external names are message positions, mappings A and E are inverses if the **get** routines are ignored. Likewise, mappings B and F are inverses. Under call by value-overwrite, the client and service together use six mappings for each REV request, which is two more than the number of mappings needed for call by value.

Under call by value-overwrite, the client creates two mappings (A and E) and inserts the arguments into a request message. A is discarded once the message is sent, while E is kept for the reply phase. E prevents the client's garbage collector from reclaiming the storage occupied by an argument object before the end of the REV request. The client sends the request message to the service, which extracts the arguments from the message. The service creates two mappings (B and F), but saves only one of them (F) for the reply phase. F prevents the service's garbage collector from reclaiming the service copy of an argument object before the end of the REV request.

When an REV request finishes executing, the service uses the appropriate **put** routine to insert each explicit result into a reply message as under call by value. The service also returns the extra results to the client, which lets the client's run-time system perform delayed updates. The client must perform delayed updates on all modified argument objects that will be accessible after the REV request completes and the client releases its mappings. We make the following conservative assumptions:

1. every argument object will be accessible to the client at the end of the request; and
2. every mutable argument object has been modified at the service.

Therefore, the service must send the client all mutable argument objects and all objects currently accessible from them. To put these objects in the reply message, the service calls **putMutableArgs**, which processes each argument object in mapping F. The action taken by **putMutableArgs** depends on the argument object O:

- O *is immutable.* Since an immutable object can not be updated, there is no need for a delayed update, and **PutMutableArgs** does nothing. The objects immediately accessible from O must be argument objects, because O was created before the REV request and has not changed. Each of these objects will be (or already has been) considered by **putMutableArgs**.

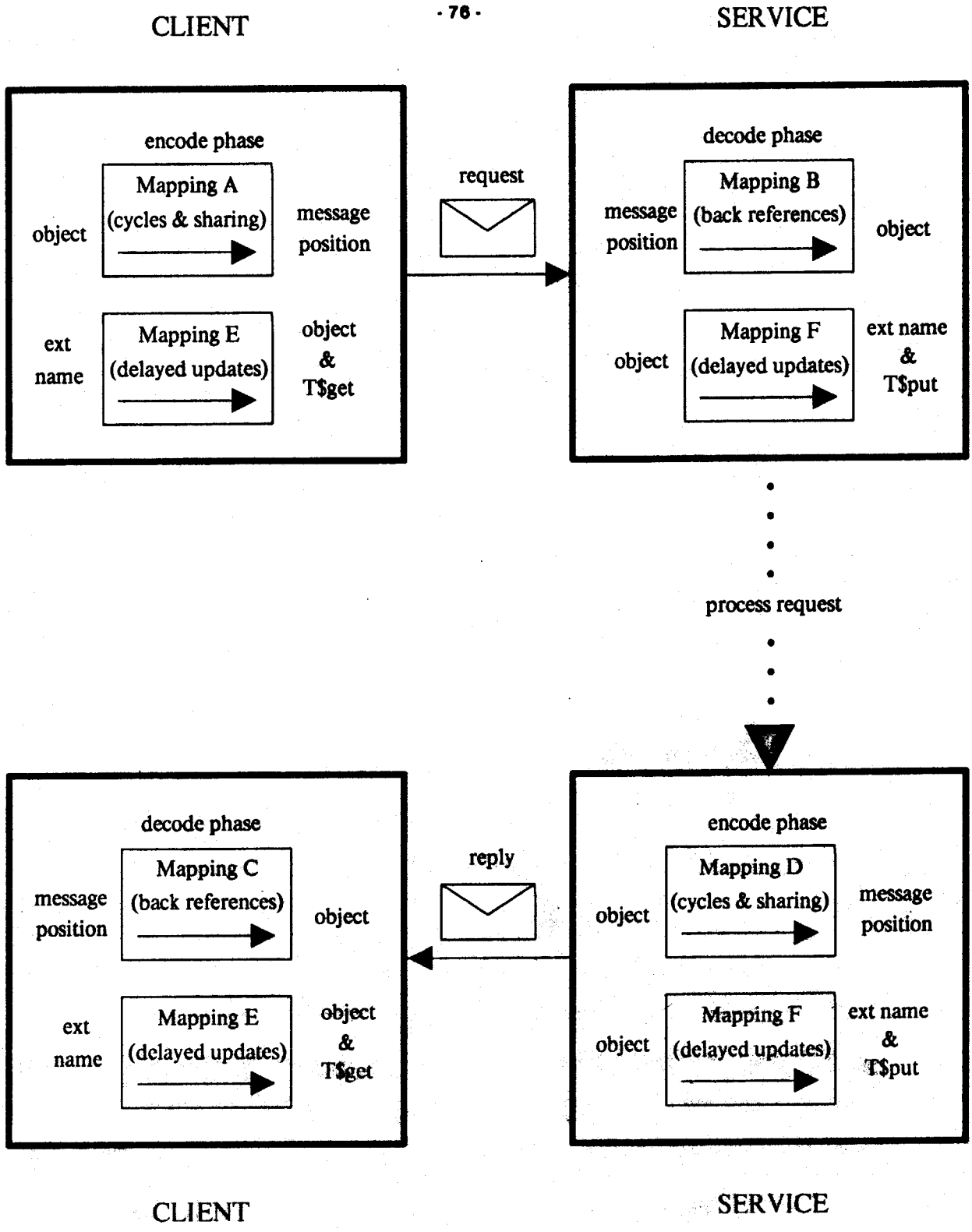


Figure 3-13: An implementation for call by value-overwrite.

- O *is mutable and has already been inserted in the message.* Since the delayed update will be done when the client extracts O from the message, **putMutableArgs** does nothing. **PutMutableArgs** uses mapping D to decide whether an object has already been inserted in the message.

- O *is mutable and has not been inserted in the message.* **PutMutableArgs** consults mapping F to find the appropriate put routine and then uses it to insert O in the message. As a consequence, every object accessible from O is inserted in the message if it is not already in the message. No object is inserted in the message more than once, since the service uses mapping D and back references to preserve sharing.

Because **putMutableArgs** is somewhat complicated, we pause for a concrete example.

Figure 3-14 contains a procedure P, which we assume is relocated by an REV request. Array A, shown in the middle of the figure, is the argument to P. The argument objects are the array A, the IntBox B, and the integer 3. The first two argument objects are mutable, while the third is immutable. Procedure P puts a 4 in IntBox B, creates a new IntBox containing a 5, and appends this IntBox onto the high end of array A twice. The procedure **ab$addh** appends its second argument (an IntBox) onto the high end of its first argument (an array of IntBoxes), thereby modifying its first argument. The final state of the argument objects is shown at the bottom of the figure.

Since there are no explicit results in this example, only **putMutableArgs** inserts objects into the result message. We use two different orderings on the argument objects to show that the net result is order independent.

- Assume **putMutableArgs** considers A first, then B, and finally 3. Since A is mutable and has not been inserted in the reply message, **putMutableArgs** applies **ab$put** to A. This inserts A, B, 4, B', 5, and a back reference to B' into the reply message. Next, **putMutableArgs** considers B and does nothing since B is already in the message. Finally, **putMutableArgs** considers 3 and again does nothing because 3 is immutable. Note that only one copy of B' is inserted in the message.

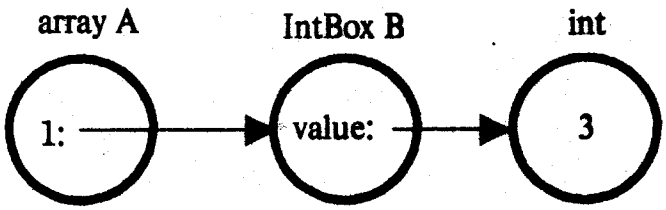
- Assume **putMutableArgs** considers the argument objects in the reverse order: 3 first, then B, and finally A. Since 3 is immutable, **putMutableArgs** does nothing. Since B is mutable and has not been inserted in the reply message, **putMutableArgs** applies **IntBox$put** to B. This inserts B and 4 into the message. Finally, **putMutableArgs** considers A and applies **ab$put** to it because A is not in the reply message. This inserts A, a back reference (to B), B', 5, and another back reference (this one to B') in the reply message. Note that only one copy of B is inserted in the reply message. The same holds for B'.

If any of the argument objects was also an explicit result, only one copy of the object would be inserted in the reply message. For instance, if P explicitly returned IntBox B, **IntBox$put** would insert B in the message before **putMutableArgs** was called. When **putMutableArgs** inserted A in the message, a back reference to B would be inserted. **PutMutableArgs** would do nothing with B during its processing of the argument objects, since B would already be in the message.

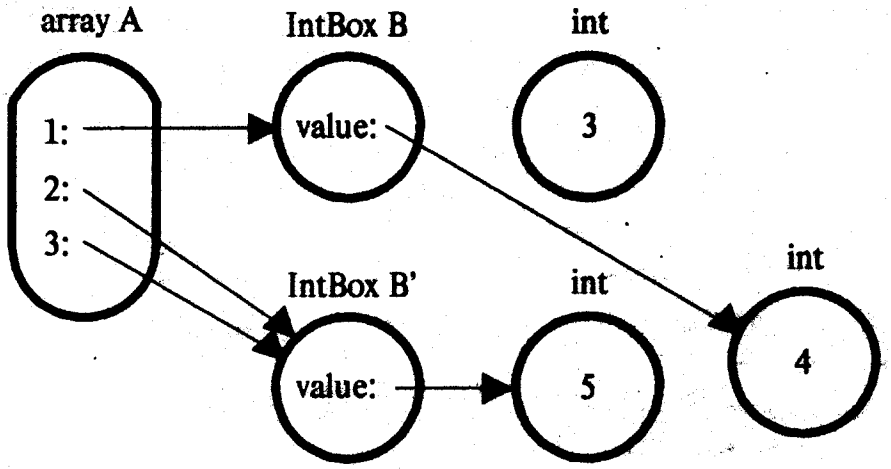
Although the compiler can tell which types can appear in the reply message, it does not know the

```
IntBox = record[value: int]
ab = array[IntBox]

P = proc (a: ab)
  a[1].value := 4
  newBox: IntBox := IntBox${value: 5}
  ab$addh(a, newBox)
  ab$addh(a, newBox)
end P
```

(a) The procedure relocated by an REV request.



(b) The arguments before P is called.



(c) The arguments after P is called.

Figure 3-14: An example that illustrates **putMutableArgs**.

number of objects of each type and their order in the reply message. The value of the arguments and the actions of the relocated procedure affect the contents of the reply message. Furthermore, the iteration order of **putMutableArgs** affects the order of the objects in the reply message. Because of

this, we keep **put** routines in mapping F. The role of the **get** routines in mapping E is explained below.

After calling **putMutableArgs**, the service sends the reply message to the client. As under call by value, the client extracts each explicit result using the appropriate **get** routine. Then the client calls **getMutableArgs**, which extracts argument objects until the message is empty. As we mentioned earlier, every result object is preceded by its external name in the reply message. Before **getMutableArgs** extracts an object, it examines the external name that precedes the object in the message and uses mapping E to select the appropriate **get** routine.

The delayed update for an argument object is done by the first **get** routine that extracts the object from the reply message. Suppose result object O with type T is the next object to be extracted from the reply message. What **T$get** does depends on the external name preceding the object in the message:

- O *is preceded by a -1.* This means O was not an argument object. **T$get** inserts the future name of the object in mapping C, calls **XT$get**, and then returns the object returned by **T$decode** as under call by value. There is no delayed update for O.

- O *is preceded by a valid message position.* This means O was an argument object. **T$get** uses mapping E to locate the original object and then updates mapping C appropriately. To perform the delayed update, **T$get** overwrites the argument object with the concrete state of the new object obtained from **T$decode**. After performing the delayed update, **T$get** returns the argument object. The object returned by **T$decode** is discarded.

If an object is extracted from a reply message several times, all occurrences but the first are back references. When a **get** routine extracts a back reference from the reply message, it uses mapping C to locate the proper object. No delayed update is done. If the object was a mutable argument object, its delayed update was done the first time it was extracted from the reply message.

We will illustrate **getMutableArgs** by describing how it handles the REV request in Figure 3-14. Assume **putMutableArgs** considers 3 first, then IntBox B, and finally array A. Thus the reply message contains B, 4, A, a back reference to B, B', 5, and a back reference to B'. **GetMutableArgs** examines the external name preceding the first object in the reply message (B) and uses mapping E to find **IntBox$get**. **IntBox$get** extracts the external name and the IntBox. Extracting the IntBox B also extracts the 4. Since B's external name is not -1, B was an argument object and **IntBox$get** performs the delayed update. Next, **getMutableArgs** examines the external name preceding the next object (array A) and uses mapping E to find **array[IntBox]$get**. This routine extracts the external name and the array. Since the array has three elements, **array[IntBox]$get** calls **IntBox$get** three times. The first call on **IntBox$get** uses mapping C to handle the back reference to B. The second call extracts B' and hence 5. Since B' was preceded by a -1, it was not an argument object, and no delayed update is done. The third call on **IntBox$get** uses mapping C to handle the back reference to B'. Once array A is extracted, **array[IntBox]$get**

performs the delayed update on A and returns to **getMutableArgs**. **GetMutableArgs** notices the reply message is empty and also returns. The final state at the client is shown at the bottom of Figure 3-14. The reader may find it useful to step through the **get** routines called by **getMutableArgs** if **putMutableArgs** had used the other ordering on mutable arguments. Of course the net result will be the same.

The following example shows how call by value-overwrite preserves sharing between arguments and results. Suppose procedure P in Figure 3-14 returns the IntBox in A[1]. In our example this is the argument object B. The explicit results are B and 4, and these two objects are placed in the reply message before the extra results. Subsequent references to B in the reply message are back references. For instance, when A is placed in the message as an extra result, it contains a back reference to B. At the client, the original IntBox B is the value of the REV request. After B's delayed update it refers to a 4, and after A's delayed update A[1] still refers to B. Note that the client has a single copy of B, just as if P were executed as a local procedure.

Because the reply message contains a sequence of objects whose structure is unknown at compile time, we keep **get** and **put** routines in mappings E and F. Each **put** routine at the client installs the corresponding **get** routine when inserting an entry into mapping E during the client's encode phase. Similarly, each **get** routine at the service installs the corresponding **put** routine when inserting an entry into mapping F during the service's decode phase.

Attempts to improve our technique for sending mutable argument objects to the client must avoid a subtle bug we encountered. Originally, the service encoded all objects accessible from the *final* value of the arguments. In other words, the service encoded the values bound to the formal arguments after the relocated procedure was executed. We introduced new variables to save the original arguments when formal arguments were assigned in the procedure. We did not worry about the objects originally accessible from the arguments passed to the procedure. The correct method, however, ensures that each mutable argument object is encoded. An earlier example (see Figure 3-4) shows the difference between the two approaches. In this example, the REV request modifies IntBox B and then removes it from array A. If B is still accessible to the program at the client, the changes to B must be reflected at the client. The correct scheme reflects the changes at the client. Our original design did not, since B is not accessible from the final value of A.

Although call by value-overwrite provides more advanced semantics than call by value, the implementation costs are not excessive. In most cases the space and time requirements for call by value-overwrite are at most twice those for call by value. Under call by value, the arguments and explicit results are transmitted once. Under call by value-overwrite, the arguments, explicit results, and extra results are transmitted once. We believe the size of the extra results will usually be comparable to the size of the mutable arguments. Call by value-overwrite uses six mappings for each REV request, while call by value uses four. The bookkeeping for call by value-overwrite is thus at most twice that for call by value.

3.4.1.3 Optimizations

The preceding section described a straightforward implementation of call by value-overwrite. We now discuss three optimizations for call by value-overwrite. First, immutable result objects that were also argument objects need not be sent in their entirety to the client. Since the appropriate abstract value already exists at the client, the service may send only the object's external name. On the other hand, immutable result objects that were not argument objects must be encoded normally. At the service, the put routine for each immutable type checks mapping F in Figure 3-13 to decide whether to send the abstract value or just the external name. Although it is unlikely that an immutable argument object O will be part of the explicit results, O might be part of the extra results. This happens whenever a mutable argument object refers to O at the end of the request.

Second, a mutable argument object that is not modified at the service does not have to be sent in its entirety to the client. Its external name is a suitable encoding. The value of this optimization depends on the overhead of detecting and remembering all service modifications to argument objects in comparison to the reduction in communications. This optimization could be limited to certain types exported by the service. A related optimization is to send only the incremental changes for modified argument objects.

Third, with the appropriate linguistic support and changes to the type system, the compiler could enforce an application programmer's declaration that an argument to an REV request was read-only. The access-control mechanism presented in [20] can provide this capability. Read-only argument objects, like immutable argument objects, do not need delayed updates.

The preceding optimizations attempt to reduce the amount of information the service sends to the client under call by value-overwrite. The remaining two optimizations apply to both call by value and call by value-overwrite. The next optimization concerns message positions and therefore external names. As every result object is preceded by its external name in a reply message for call by value-overwrite, we want to make external names small. Three of the six mappings in Figure 3-13 (and two of the four mappings in Figure 3-12) map message positions or external names to something else. We want to implement these mappings with small arrays rather than with hash tables, balanced trees, or large, sparse arrays. The approach described below accomplishes these objectives.

A message contains a sequence of objects. Each object may be identified by its position in the sequence or by its starting byte position. We favor the former, sequence-oriented approach over the latter, byte-oriented approach. Assume n objects are encoded and the resulting message is m bytes long. Typically m is much larger than n. With a sequence-oriented approach, valid message positions are the integers from 1 to n. Small arrays can implement the mappings described above. With a byte-oriented approach, valid message positions are sparse in the large interval from 1 to m. Depending on the sparsity, either a large array, a small hash table, or a balanced tree would implement each of the three mappings described above. The sequence-oriented approach can be

faster than a byte-oriented approach, since array accessing is typically faster than hashing or tree searching. The sequence-oriented approach uses less memory for a mapping than the alternative data structures. Finally, the sequence-oriented approach reduces message size by reducing the length of back references and external names. We do not have enough experience, however, to predict the significance of these advantages.

Finally, we favor *relative* back references in a message instead of *absolute* back references. We illustrate the difference with an example. Suppose a back reference inserted at message position 55 refers to the object at message position 17. With absolute back references, the back reference is 17, but with relative back references, the back reference is 55-17 = 38. If back references are absolute, a process encoding some objects must know the absolute message position of the first object it encodes. If back references are relative, a process encoding some objects does not need to know where its output will be positioned within the message.

This optimization may be important for immutable objects, such as routines without own variables, that are encoded at compile time. Relative back references let disjoint groups of objects be encoded at different times without retaining information from earlier encodings. This lets an REV mechanism encode the code portion of an REV request at compile time and the arguments at run time without having to retain compile-time information. Furthermore, relative back references let encodings be combined by concatenation. Consider a procedure that is an argument to several REV requests and assume it is not referred to by any other routine in the code portions of the requests. Being transmissible, the procedure can be encoded at compile time. At run time, its encoding can be blindly copied into the request message no matter if it is the first argument, the last argument, or one of the other arguments.

3.4.2 Code Transmission

Besides implementing call by value-overwrite, an REV mechanism must transmit routines without own variables from the client to the service. Herlihy's scheme for transmitting abstract values between nodes solves only part of the problem of transmitting code between nodes, as it handles only those details that do not depend on the type being transmitted. For example, references from a relocated procedure to client-supplied routines as well as references from a routine to itself (recursion) are handled automatically. In this section we focus on the remaining tasks: determining the representation for routines imported by an REV request; and determining the external representation for code.

Since every nontrivial REV request imports at least one service routine, we need an external representation for imported routines. One possibility is a pair of the form <**InterfaceName, RoutineName**>. The service could bind imported routines to their implementations when it extracts the code portion from the request message (static linking) or as it executes the request (dynamic linking). This representation for imported routines is simple but verbose. If the client and the service

agree on the ordering of interfaces, a more compact representation for imported routines is possible. In this case, each imported routine may be represented by a pair of the form <**InterfaceOffset, RoutineOffset**>.[6] The ordering on interfaces could be the one in the publically available, compiled version of the service definition. Alternatively, the client could send the ordering when establishing the binding.

Besides determining the external representation for imported routines, we must determine the external representation for code. As explained below, the choice of an external representation for code is a complicated trade-off involving execution efficiency, request message size, and security considerations. We first consider machine-dependent external representations for code and then consider machine-independent external representations.

A machine-dependent external representation for code involves compiled code or something close to it. Transmitting compiled REV requests is a viable option in homogeneous computing environments. Compiled code realizes a fairly compact encoding and achieves high performance. Nevertheless, transmitting compiled code raises security considerations, as a compiled REV request may have immediate access to peripherals, registers, and all (virtual) memory locations. If security considerations are important and compiled code is transmitted, a trusted compilation service could examine, compile, and attach a digital signature to REV requests at compile time. This would prevent a client from sending hostile requests to a service. Alternatively, each service could use conventional time-sharing protection mechanisms and provide a separate address space for each service capability it grants.

Transmitting compiled REV requests may not be the best solution in a heterogeneous computing environment, as a compiled REV request can not be executed by all processors. Machine-independent external representations for code, in contrast, let each REV request be independent of the processor that executes the request. Machine-independent external representations include character strings, parse trees, and bytecodes. Since dynamic compilation is probably too expensive, these code representations require an interpreter. An important advantage of bytecodes is that they realize compact encodings compared to source code and compiled code.

A hybrid approach may be useful in a heterogeneous computing environment, especially when a few machine architectures account for most of the processors. Under this approach, each REV request is compiled for the two or three leading processor types. The client sends compiled code whenever possible and uses the machine-independent representation for the remaining processor types.

We avoid problems caused by own variables by refusing to transmit code that refers to own variables.

---

[6] If an REV request is located in a module parameterized by a service definition, the REV request may be sent to instances of *different* services. The compiler can encode the REV request once for each service definition that instantiates the module and use the appropriate encoding at run time. Alternatively, the compiler can encode the request only once by using the verbose representation described above.

Every variable in a procedure transmitted between nodes is either an argument or an ordinary local variable. Such variables are not shared between processes, since each process executing an entire procedure has a private activation record.

3.4.3 Request Interpretation

When machine-independent code is transmitted between nodes, each service needs an interpreter to evaluate REV requests. One problem with this approach is maintaining identical semantics between interpreted code and compiled code. Many LISP implementations, however, offer a compiler as well as an interpreter and can evaluate an expression regardless of which routines are compiled and which are interpreted. Mixing direct execution with interpretation is not new.

3.4.4 Service Binding

Service binding involves locating an appropriate instance of a service and giving a service capability to the client. The only difference between RPC binding and REV binding is whether a client specifies a single interface (RPC) or a set of interfaces (REV).

Birrell and Nelson [4] describe how Grapevine [3] supports remote binding for RPC's. A node wishing to export an interface communicates its intent to a Grapevine server. If the node is authorized to export the interface, Grapevine updates its database of nodes and the interfaces each node exports. A client wishing to import a remote interface queries a Grapevine server, which normally returns a capability for an appropriate instance of the interface. The client then can use this capability to communicate with the remote node. This technique of using an intermediary to facilitate remote binding could be extended to do service binding for REV.

REV binding can be slightly harder than RPC binding, in that a client's needs may not exactly match any public service definition. In this case, the binding facility must consider all public service definitions that exceed the client's needs. We call this extension to remote binding *subset binding*.

Subset binding does not affect REV requests that are encoded with verbose pairs of the form **<InterfaceName, RoutineName>** for imported routines. Subset binding, however, does affect REV requests that are encoded with compact pairs of the form **<InterfaceOffset, RoutineOffset>** for imported routines. In this case the client and service must agree on the interface ordering, i.e., which interface corresponds to which **InterfaceOffset**. We let the client dictate the ordering and send it to the service when the binding is established. There are two reasons for this decision. First, subset binding lets the client execute the same REV request at nodes that export different services, as long as each service exceeds the particular service requested by the client. It is unlikely that these nodes will agree on the ordering of interfaces imported by the REV request. Second, the client already knows the ordering, which is chosen by the compiler when it encodes the code portion of the REV request.

3.4.5 Reliable Communication

A client and service communicate by using request-reply message pairs. An RPC communication mechanism converts an unreliable (packet-based) network into a reliable communication link that lets the client and service exchange messages of arbitrary length. Such a communication link may be implemented on top of a datagram service [41]. An REV communication mechanism must do the same. Furthermore, the RPC communication mechanism might use encryption to guarantee the security and integrity of data sent between nodes. Again, encryption-based techniques are directly applicable to REV communication.

3.4.6 Failure Recovery

RPC systems like Argus [25] use atomic transactions to tolerate node and communication failures while providing at-most-once semantics for RPC's. In these systems, a single transaction can span several nodes and last for an arbitrarily long time. An REV mechanism can use the same approach. Nested transactions can be included as an option.

A node failure can create *orphans*. An orphan process is a remote invocation (indirectly) initiated by a node that has since crashed [21]. Orphans may exist anywhere when REV is used, since nested REV requests can establish arbitrary communication paths between services. The orphan problem must be solved by any transaction-based system that supports RPC's. Remote evaluation does not appear to complicate the detection or extermination (i.e., killing) of orphans.

3.5 Discussion

This chapter compared an implementation for REV with a hypothetical RPC implementation. The main differences between an REV mechanism and an RPC mechanism are:

1. supporting call by sharing instead of call by value;
2. supporting code transmission, which might require an interpreter; and
3. verifying the validity of REV requests and determining their location independence.

As the bulk of an REV mechanism is an RPC mechanism, most of the techniques for tuning the performance of an RPC mechanism apply directly to an REV mechanism.

At this point we evaluate REV according to the constraints discussed at the end of Chapter 1:

- *Powerful Semantics:* We defined REV semantics so that relocating processing with a location-independent REV request has no effect on program semantics. In order to accomplish this goal, we imposed minimal constraints on an REV request. Each request must be a procedure without own variables. The body of the procedure must be known at compile time. Procedure variables are permitted in the request as long as the set of values for each procedure variable is known at compile time. A programmer can otherwise use the full power of the language, such as conditionals, loops, exceptions, and REV requests, to express an REV request. A request naming a routine that is not exported by the service can supply its own implementation.

- *Implementation Efficiency:* REV requests should be efficient. The code portion of an REV request can be checked for validity and encoded at compile time. In most cases, the space and time requirements for call by value-overwrite will be at most twice those for call by value.

- *Ease of Use:* REV is easy to use. A programmer can change a local invocation into an REV request by enclosing the service in two reserved words (AT and EVAL) and placing the expression before the invocation. This textual change has no effect on program semantics for location-independent REV requests.

- *Language Independence:* REV is language independent. Although we tailored the argument passing semantics to that for local invocations in CLU, similar mechanisms exist for other languages.

One area we can improve is ease of use. We currently require the programmer to write each REV request as a procedure. Repeated insertion and removal of REV requests can alter the way a program is decomposed into procedures. An easier way to introduce REV lets the programmer specify an REV request as a sequence of statements. The following chapter describes this extension.

Chapter Four

REV with Implicit Procedures

REV meets most of the requirements listed in Chapter 1. REV gives the programmer fine-grained control over the location of processing in a distributed application. Furthermore, location-independent REV requests relocate processing without affecting program semantics. An REV request, however, must relocate the execution of a single, complete procedure. Many REV requests will not represent a coherent idea but merely reflect the locations of particular objects. This could cause a proliferation of unnatural procedures, which in turn might make reading and maintaining programs more difficult.

We remedy this shortcoming by allowing *implicit* REV requests. Such requests relocate the execution of a sequence of statements instead of a procedure. We call the REV requests shown in earlier chapters *explicit* requests because the relocated procedure is explicit. While simplifying the insertion of REV requests into a program, implicit REV requests do not increase the power of a programming language with explicit requests. This chapter provides linguistic support for implicit REV requests, defines their semantics, and shows how to implement them.

4.1 Implicit REV Requests

An *implicit* REV request lets an application programmer relocate the evaluation of a *closure*. A closure consists of code and an environment in which to evaluate the code. Closures, which are not first-class objects in CLU, appear only as the body of iterators. The code for each closure in CLU is apparent at compile time. A programmer writes an implicit REV request by specifying a closure and the service that executes the request. Implicit requests are accommodated by changing the syntax for REV requests:

```
rev_expression ::= at expression eval body [expression] end
```

The optional *expression* following *body* lets an implicit REV request return a value. The reserved words EVAL and END delimit the closure. For clarity, we will assume the programmer writes implicit REV requests but not explicit requests. Later in the chapter we show how to convert an implicit request into an explicit request.

Our semantics for implicit REV requests will ensure that location-independent requests, which are defined below, preserve program semantics. An implicit request must execute as part of some atomic transaction that is aborted if a node or communication failure prevents the request from completing.

Implicit REV requests use call by sharing: as explained in Section 4.3, the arguments for an implicit REV request are part of the context.

We extend the definitions of validity and location independence to encompass implicit REV requests. As before, a valid REV request is one the compiler can encode and the service can execute, and a location-independent REV request is a valid request that relocates processing without altering program semantics.

An implicit request is *valid* if its code portion meets three conditions similar to those that define a valid explicit request. First, all routines imported by the code portion are exported by the service that executes the request. Second, all free variables in the closure are defined in the surrounding environment, and the remainder of the code portion can not have own variables. These two conditions ensure that the request is self-contained. Finally, the code portion of a valid request is apparent at link time. This lets the compiler encode the code portion and check the validity of the request.

An implicit REV request is *location-independent* if: the request is valid; every routine imported by the request is location-independent; and every variable accessed by the request is local to the transaction associated with the request.[7] The last restriction is needed because our transaction mechanism does not apply to variables. These restrictions and the semantics of implicit REV requests ensure that location-independent requests do not change program semantics.

4.2 An Example

Figure 4-1 recasts an earlier mail example (Figure 2-7) into a program using an implicit REV request. This program is equivalent to the earlier program that used an explicit REV request. There are two important points to note. First, introducing/removing an implicit REV request corresponds to inserting/deleting the two lines marked with asterisks. A programmer adding an explicit REV request, in contrast, would have to convert the closure into a procedure. Second, since the implicit request marked with asterisks is location-independent, it does not change program semantics. All the REV bookkeeping concerning arguments, results, and flow of control is done automatically, as explained in the next section. The programmer concentrates on producing a bug-free implementation instead of worrying about these details. With well-structured code, an optimizer can easily relocate execution when tuning program performance.

---

[7] If it is possible to have concurrency within a transaction, every variable accessed by the request must be local to the process executing the request.

```
ss = set[string]        % an equate

customize = proc (user, passWd, msg, registry: string, friends: ss)
  returns (ss) signals (NotAvailable)

  begin transaction
    postOffice: mail := Service[mail]$Lookup(registry) resignal NotAvailable

    at postOffice eval                                    % ***
      badNames: ss := ss$new()
      newMsg, firstName: string
      for friend: string in ss$elements(friends) do
        firstName := at postOffice eval registry$firstName(friend) end
          except when noSuchUser:
            ss$insert(badNames, friend)
            continue   % start the next iteration
          end
        newMsg := "Dear "||firstName||msg  % string concatenation
        at postOffice eval maildrop$send(user, passWd, friend, newMsg) end
      end   % loop statement
      return (badNames)       % return from REV request
                              % and customize
    end   % REV request                                   ***

  end except when failure (reason: string): signal NotAvailable end
end customize
```

Figure 4-1: An example of an implicit REV request.

4.3 Implementation

We implement an implicit REV request by converting it into an explicit request at compile time without altering program semantics. This transformation, which converts a closure into a procedure, is the opposite of in-line expansion. We call it *procedure folding*. Procedure folding has two tasks:

1. ensure the new procedure does not affect the flow of control; and
2. determine the arguments and results of the new procedure.

Each of these subtasks is described in turn.

4.3.1 Control Flow Preservation

Procedure folding must accommodate control constructs that terminate an implicit REV request. An implicit REV request may terminate in four ways:

1. an invocation in the request raises an exception that is not handled by the request;
2. the last statement executed causes a nonlocal transfer of control (i.e., a **return** or **signal** statement is executed);
3. the last statement executed causes a local transfer of control (i.e., a **continue**, **break**, or **exit** statement is executed); or
4. the last statement executed does not affect the flow of control.

A nonlocal transfer of control terminates the current activation, while a local transfer of control does not. We define the above CLU constructs in the following discussion, which considers each possibility in turn.

Procedure folding must not change which exceptions are handled. If an exception raised by an invocation in a closure is handled, it is handled in the routine containing the closure. Otherwise, the exception becomes a **failure** exception at the boundary of the routine containing the closure. The compiler can determine which exceptions the routine catches by examining the handlers in the routine. Those handlers whose scope contains the closure determine the exceptions that the anonymous procedure created by folding must resignal. We call these exceptions *client-handled* because the client handles them after the REV request. Other exceptions are not caught and become unhandled exceptions.

The example in Figure 4-2 shows the distinction between client-handled exceptions and unhandled exceptions. Evaluating a[b] or a[b + 1] might raise a **bounds** exception. This exception, which is caught by the handler in **SomeProc**, is a client-handled exception. The division routine might raise a **zero_divide** exception. Since there is no handler for this exception in **SomeProc**, it is an unhandled exception.

Folding the implicit REV request in Figure 4-2 yields the explicit request in Figure 4-3. The following section describes how we determine the arguments and results of the anonymous procedure (**G4250**). This procedure resignals the client-handled exceptions and ignores the unhandled exceptions. If the array access causes a **bounds** exception, the handler in **SomeProc** will catch it. In the corresponding program without the REV request, the same handler catches the **bounds** exception. If the division causes a **zero_divide** exception, the exception will not be handled. The unhandled exception becomes a **failure** exception at the boundary of procedure **G4250**. In the corresponding program without the REV request, the unhandled exception becomes a failure exception at the boundary of **SomeProc**. Both unhandled exceptions and **failure** exceptions abort all transactions they exit. Hence the REV request does not affect the meaning of an unhandled exception like **zero_divide**.[8]

Having discussed exceptions raised by invocations in an implicit REV request, we now focus on CLU constructs that transfer control out of an implicit request and terminate the current activation: **return** and **signal**. When such CLU constructs are present, the anonymous procedure created by folding returns a *oneof*, which is a tagged, discriminated union. Each possibility for the oneof corresponds to one way the implicit request may terminate. New code inserted after the REV request handles each possibility in the appropriate manner.

We again use an example to illustrate procedure folding. Figure 4-4 contains an implicit REV request

[8] When failure is a client-handled exception, unhandled exceptions must be accommodated in a different way.

```
SomeProc = proc (a: array[int], b: int) returns (int)
  begin transaction
    ans: int
    aService: built-ins := Service[built-ins]$Any()
    at aService eval
      ans := 3/a[b]
      ans := ans/a[b+1]
    end
    return(ans)
  end except when bounds: return(0) end
end SomeProc
```

Figure 4-2: An implicit REV request whose closure raises several exceptions.

```
SomeProc = proc (a: array[int], b: int) returns (int)
  begin transaction
    ans: int
    aService: built-ins := Service[built-ins]$Any()
    ans := at aService eval G4250(a, b)
    return(ans)
  end except when bounds: return(0) end
end SomeProc


G4250 = proc (a: array[int], b: int) returns (int) signals (bounds)
  begin
    ans: int := 3/a[b]
    ans := ans/a[b+1]
    return(ans)
  end resignal bounds
end G4250
```

Figure 4-3: The implicit request in Figure 4-2 after folding.

---

that can execute a **return** or a **signal** statement. Folding the implicit request yields the explicit request in Figure 4-5. Names unique in the current environment are automatically generated for the oneof type (G1010), the anonymous procedure (G1011), and the oneof variable (G1012). At most one arm of a **tagcase** statement is executed. The result returned by the REV request determines which arm of the **tagcase** statement in Figure 4-5 is executed.

The remaining CLU constructs that affect the flow of control are **break**, **continue**, and **exit**. These constructs cause a local transfer of control; i.e., they do not terminate the current activation. The **break** statement terminates execution of the smallest loop statement in which it appears. The **continue** statement terminates execution of the body of the smallest loop statement in which it appears. An **exit** statement is similar to a **signal** statement in that both raise an exception. **Signal** terminates the current activation, but **exit** does not. **Exit** statements are legal only when there is an enclosing handler of the appropriate type.

For these CLU constructs, determining the destination of the control flow is a straightforward

```
SomeProc = proc (a: array[int], b: int) returns (bool) signals (negativeArg(int))

  begin transaction
    ans: int
    aService: built-ins := Service[built-ins]$Any()
    at aService eval
      if b<0 then signal negativeArg(b)
      elseif b=0 then return(a[1]=0) end
      ans := a[b]
    end  % REV request
    return(ans=33)
  end  % transaction

end SomeProc
```

Figure 4-4: An implicit REV request with **signal** and **return** statements.

```
G1010 = oneof[normal: int,
              return: bool,
              negativeArg: int]

SomeProc = proc (a: array[int], b: int) returns (bool) signals (negativeArg(int))

  begin transaction
    ans: int
    aService: built-ins := Service[built-ins]$Any()
    G1012: G1010 := at aService eval G1011(a, b)
    tagcase G1012
      tag normal (i: int): ans := i
      tag return (b: bool): return(b)
      tag negativeArg (i: int): signal negativeArg(i)
    end
    return(ans=33)
  end  % transaction

end SomeProc


G1011 = proc (a: array[int], b: int) returns (G1010)
  if b<0 then return (G1010$make_negativeArg(b))
  elseif b=0 then return (G1010$make_return(a[1]=0)) end
  ans: int := a[b]
  return (G1010$make_normal(ans))
end G1011
```

Figure 4-5: The implicit request in Figure 4-4 after folding.

---

compile-time task. Assume one of these constructs appears in an REV request. If its destination is in the request, no problem occurs. If its destination is outside the request, the construct is handled in the same manner as a **return** or **signal** statement. An arm of the **tagcase** statement, which directly follows the original implicit request, transfers control appropriately. Since these constructs do not terminate the activation, procedure folding must preserve the environment. The values returned for these constructs, like the value returned for the "normal" tag in Figure 4-5, are used to update the

environment. The next section describes which values are returned and explains why they are returned.

4.3.2 Argument/Result Determination

Besides preserving the flow of control, procedure folding must determine the arguments and results of the procedure it creates. We begin with an example of procedure folding by converting the implicit REV request in Figure 4-6 into the explicit request in Figure 4-7. The arguments to the anonymous procedure (**G7345**) are *a* and *c*, since the closure reads these variables before updating them. Although the closure also uses the values bound to *b* and *d*, these values are computed by the closure. The results of the anonymous procedure are *c*, *d*, and *e*, since the closure defines these variables and **SomeProc** subsequently reads them. Although the closure also defines *b*, **SomeProc** redefines *b* before using it.

The anonymous procedure created by folding has no free variables, as the free variables in the closure are converted into arguments (*a* and *c*) or local variables (*b*, *d*, and *e*). The multiple assignment in **SomeProc** restores the minimal portion of the environment needed to preserve program semantics.

Although we use call by value-overwrite for objects, call by value-overwrite is not needed for variables. We use call by value-result for variables and can still claim that location-independent requests do not change program semantics. We need not worry about concurrency, because we have prohibited concurrency within a transaction and required that all request variables be local to the transaction. There is no aliasing that involves variables, because CLU objects and variables can not refer to variables.[9]

Use-definition analysis [1], a technique often used in optimizing compilers, lets us determine the arguments, results, and locals of the procedure created by folding. We begin with some terminology. A variable is *defined* by a program fragment if its value may be set by the program fragment. A variable is *used* by a program fragment if its initial value may be read by the program fragment. To simplify the discussion, we assume each variable is initialized when it is declared.

We determine the results of the anonymous procedure before the arguments, because the arguments may depend on the results. The *results* are those variables that are defined by the request and used by the code that dynamically follows the request. As shown in Figure 4-7, a **return** statement containing the result variables is appended to the closure. If the closure has several termination points, there may be several **return** statements. The anonymous procedure will in general return a oneof, as shown in Figure 4-5. The *arguments* to the anonymous procedure are those variables that

---

[9] For languages in which variables and objects can refer to variables, the current environment must be treated as an atomic object. Procedure folding can be extended to handle this case.

```
SomeProc = proc (a: int) returns (int)
  begin transaction
    aService: built-ins := Service[built-ins]$Any()
    b, c, d, e: int
    c := 3
    at aService eval
      b := a*a
      d := c*c
      c := d
      e := b+d
    end  % REV request
    b := e*e
    return(b*d*c)
  end  % transaction
end SomeProc
```

Figure 4-6: An implicit REV request.

```
SomeProc = proc (a: int) returns (int)
  begin transaction
    aService: built-ins := Service[built-ins]$Any()
    b, c, d, e: int
    c := 3
    c, d, e := at aService eval G7345(a, c)
    b := e*e
    return(b*d*c)
  end  % transaction
end SomeProc

G7345 = proc (a, c: int) returns (int, int, int)
  b: int := a*a
  d: int := c*c
  c := d
  e: int := b+d
  return(c, d, e)
end G7345
```

Figure 4-7: The implicit request in Figure 4-6 after folding.

---

are used by the program fragment consisting of the request and any **return** statements introduced by procedure folding. Finally, the *locals* are those variables that appear in the request but are not arguments.

4.4 Discussion

Although we restricted our attention to CLU constructs when discussing procedure folding, we believe that similar techniques exist for constructs found in other programming languages. For pedagogical purposes, we discussed procedure folding without discussing stub generation. To achieve high performance stubs, a production-quality REV mechanism could combine the two activities and use standard compiler techniques to optimize the flow of control.

Implicit REV requests directly support remote iterators. For example, assume the interface **mailbox** contains the following iterator, which lets a programmer process each message in a mailbox:

```
messages = iter (user: userID) yields (string) signals (noSuchUser, unreadable)
```

The application programmer can use an implicit REV request to execute the iterator entirely at the service containing the mailbox:

```
at postOffice eval
  for message: string in mailbox$messages(myUID) do
    .
    . % process the message
    .
  end  % iterator
end  % REV request
```

The compiler automatically converts the implicit REV request into an explicit REV request without altering program semantics.

One drawback of an implicit REV request is that its arguments and results are not readily apparent to a programmer reading, revising, or debugging a program containing the request. This information, while irrelevant to someone understanding the program, is crucial to someone tuning program performance. A useful compiler option would be the ability to list the arguments, results, and client-supplied routines for each implicit REV request.

4.5 Summary

Requiring that an REV request relocate the execution of a complete procedure burdens the programmer and may result in a proliferation of tiny procedures. This chapter extended the REV model by considering implicit REV requests in which the programmer designates a closure instead of a procedure for remote execution. This extension is for programmer convenience and program readability.

An implicit request uses call by sharing and executes as part of an atomic transaction that aborts if the request does not complete. A location-independent request has no effect on program semantics, and implicit requests can be arbitrarily nested. The constraints on implicit REV requests are similar to those defined for explicit REV requests, except that free variables defined in the surrounding environment are allowed. Implicit REV requests represent a slight compile-time enhancement of REV. They do not affect service definitions, type checking, remote binding, or run-time support.

Chapter Five

Remote Data Types

In our extended version of CLU, there is no built-in naming mechanism that lets a program running on one node refer to an object at another node. This naming mechanism is useful when we view a remote node as a repository of shared objects. For example, a file service consists of directories and files, and a programmer may want to manipulate the same file in several REV requests. We can simplify the programmer's task by letting the program refer to the remote file in between REV requests to the file service.

Because we feel such a naming mechanism will be useful in many distributed applications, we will incorporate it directly into the programming language. Although it is possible to construct such a naming mechanism outside the programming language, we provide direct support for reasons of convenience and expressive power. Our naming mechanism differs from conventional approaches based on global capabilities (e.g., Stroustrup's approach discussed in Section 1.3.4), because we meet the following requirements:

1. *Transience:* Once a client-service binding is broken, the client must not refer to service objects and vice versa. This relaxes our assumption concerning the disjointness of separate address spaces, but retains most of the advantages of the original assumption.

2. *Good Documentation:* The possibility that an object might exist at a remote node must be obvious to a person reading a program that involves the object. This is important because of both performance and semantic considerations. The communication costs incurred when manipulating an object at another node affect performance, while the above transience requirement affects program semantics.

3. *Convenience:* The naming mechanism must be easy to use and understand, which means the run-time system should manage most of the details.

4. *Safety:* We implement call by sharing for REV requests with call by value-overwrite. The naming mechanism must not invalidate the correctness of our implementation.

Our naming mechanism has two components: *global names* and *remote data types*. In this chapter we describe these components, show how they meet the above requirements, and sketch an implementation.

5.1 Global Names

A node uses a *local name* to refer to an object it contains. A local name is meaningful only to the node containing the object and therefore can not be passed between nodes. A *global name* also refers to an object, but it can be transmitted between nodes. If an object has a global name, any node may use the global name to refer to the object.

A global name consists of a service capability for the node containing the object as well as a unique identifier interpreted by that node. Only the node containing the object can convert between the global name and the object's local name. Since each global name contains a service capability, one node can not accidentally interpret a global name issued by another node. This requires run-time checking, but all errors of this kind can be detected. We assume that a node which generates global names for its objects has a mapping that converts between global and local names. In addition, we assume the garbage collector never reclaims such a mapping.[10] This mapping prevents the garbage collector from reclaiming any object whose global name is in the mapping, since such an object might be referenced by another node. Below we describe the conditions under which a node removes global names from its mapping.

Because we want to reclaim inaccessible objects without requiring a distributed garbage collector, we treat global and local names differently. A local name is always valid; i.e., if a program comes across a local name, the corresponding object is guaranteed to exist. A global name, in contrast, is not always valid. It becomes invalid when its service capability becomes invalid. This happens when either the client or service breaks the binding between them (see Section 2.1.3).

When a service capability becomes invalid, the service removes the invalid entries in its mapping between global and local names. This can make some of the service objects inaccessible and thus subject to garbage collection. Service objects that remain accessible, either by local names or by valid global names, will not be reclaimed. For example, files in a persistent directory will not be reclaimed, because they are accessible from the directory. A file removed from the directory and accessible only from an invalid global name can be reclaimed. Invalid global names do not cause dangling reference problems, since the service can not convert an invalid global name into a local name. Thus a program can not use an invalid global name to access an object reclaimed by the garbage collector.

Our lifetime definition for global names has three consequences. First, it meets our transience requirement. Second, passing global names implements call by sharing for REV requests, but it does so only while the service capabilities in the global names are valid. This is in contrast to call by value-overwrite, which implements call by sharing indefinitely. Third, each node can have an

---

[10] If client-service bindings (and hence service capabilities) survive node crashes, this mapping must also survive node crashes. Otherwise, the mapping can be kept in volatile storage.

independent garbage collector that is not concerned with other nodes. As long as client-service bindings are short, say on the order of minutes or hours, or if they deal with a small to moderate number of global names, a distributed garbage collector is not needed. For long-lived bindings that create and discard many large objects with global names, we recommend routines that release most of the resources held by such objects. For example, assume a file object is a file descriptor that refers to the disk pages associated with the file. Furthermore, assume **file$delete** deallocates the disk pages associated with the file without deallocating the file descriptor, which acts as a tombstone. The local name for a deleted file object refers to its descriptor and hence is not a dangling reference. The garbage collector reclaims the file descriptor when it becomes inaccessible.

Global names give a programmer fine-grained control over the location of data, and this control may be used to improve the performance of a distributed system. When objects are very large, encoding, decoding, allocating memory, reclaiming memory, and transmitting data can be expensive in both time and space. Transmitting the small global name of an object instead of its abstract value can be more efficient. The net effect on performance, however, depends on the number of times the object is accessed, as accessing the object must be done at the node containing the object.

An application programmer can use global names to protect client information manipulated by an REV request. While a communication channel using encryption can, with high probability, prevent a third party from viewing or modifying information sent between the client and the service, encryption does not help a client that sends sensitive information to an untrusted service that may violate the data abstraction that protects the information. For example, transmitting a client object by value-overwrite exposes its entire abstract state to the service, which could copy the abstract state or manipulate it in arbitrary ways. Sensitive information should therefore be kept at the client and transmitted by global name. A client that processes only simple REV requests (i.e., RPC's) and does the appropriate checking can prevent an untrusted service from indiscriminately accessing this information.

5.2 Remote Types

Our second requirement on the naming mechanism is good documentation. A person reading a program should be able to tell which objects are local and which objects may not be local. We meet this requirement by dividing abstract data types into two disjoint sets:

- A *local (data) type* is an abstract data type whose objects have local names but not global names. Each instance of a local type exists at some node, and no other node can refer to the instance. A local data type may be transmissible by value-overwrite; otherwise, it is nontransmissible. The programmer defining a local type decides whether it is transmissible.

- A *remote (data) type* is an abstract data type whose objects can have global names that

are meaningful at all nodes. We call an instance of a remote type a *remote object*. The operations a program can perform on a remote object are defined by the object's type. A remote type is automatically transmissible. A node transmits a remote object to another node by sending the object's global name to the other node. Unlike local objects, a remote object is never transmitted by value-overwrite.

An abstract data type is either local or remote; it can not be both. Later in the chapter we explain why we associate the idea of being remote with types instead of with individual objects or formal arguments to REV requests. Except for service capabilities, the abstract data types discussed so far in the thesis have all been local types. Scalar types, such as booleans, characters, integers, and reals, are local types. Records, arrays, strings, oneofs, procedures, and iterators are also local types.

Although there are no rules for a programmer deciding whether a transmissible type should be a local type or a remote type, we offer the following guideline. Types whose instances tend to be large or contain sensitive information should be remote types. Types whose instances are often shared by many users should also be remote types. Examples include include database relations, disk files, and mailboxes.

To specify a remote type T, the programmer creates an interface called T that is annotated by the new reserved word REMOTE. For instance, Figure 5-1 contains an interface for the remote type **mailbox**. A programmer implementing **mailbox** implements all the routines in Figure 5-1. As explained in Section 5.4, the compiler automatically generates **get** and **put**, which make **mailbox** be a transmissible type. The **mailbox** routines are location-dependent, since the semantics of each routine depends on the node executing the routine. Hence the application programmer must specify the node that executes each invocation of a **mailbox** routine.[11]

We shall assume it is the programmer's responsibility to manipulate a remote object only in an REV request executed by the node containing the object. This will make all the REV requests in our examples apparent to the reader. The program fragment in Figure 5-2 shows that the syntax for using a remote data type, such as **mailbox**, is the same as the syntax for using a local data type. The only way to tell whether a type is local or remote is to check the interface that specifies the type. Since Figure 5-1 declares **mailbox** to be a remote type, the client variable *mbx* contains a global name and not a local name between the two REV requests in Figure 5-2. In an REV request sent to the node containing the mailbox (i.e., *postOffice*), *mbx* contains a local name that directly refers to the mailbox. The translation between global names and local names is done automatically, as explained in Section 5.4.

This example shows why we have a complete separation between local types and remote types. If we

---

[11] A useful default is to execute each routine at the home node of the first argument if this argument has a remote type. For instance, the last six routines in Figure 5-1 could be relocated automatically to the home of their first argument, a mailbox. Since **create** and **open** have no arguments with remote types, the application programmer would use REV to relocate their execution.

```
mailbox = remote interface is

   create = proc (user, maintainer: userID) returns (mailbox) signals (userExists)
   % create a mailbox for the user at the node executing this procedure
   % userID = name & password

   open = proc (requester: userID, user: string) returns (mailbox) signals (noSuchUser)
   % open user's mailbox on the node executing this procedure
   % requester must be the mailbox owner, the system maintainer, or the mail system

   read = proc (mbx: mailbox) returns (array[string]) signals (unreadable)
   % read the contents of the mailbox -- fails if mailbox is remote

   messages = iter (mbx: mailbox) yields (string) signals (unreadable)
   % iterate through the messages -- fails if mailbox is remote

   addMessage = proc (mbx: mailbox, msg: string) returns ()
   % append message msg to the ones in the mailbox -- fails if mailbox is remote
   % used only by the mail system, since senders invoke a higher-level routine

   delete = proc (mbx: mailbox, msgNos: array[int])
   % remove the specified messages -- fails if mailbox is remote

   close = proc (mbx: mailbox)
   % prevent further manipulation of the real mailbox via this abstract object
   % fails if mailbox is remote

   destroy = proc (mbx: mailbox) returns ()
   % remove mailbox from node -- fails if mailbox is remote

end mailbox
```

Figure 5-1: The mailbox interface.

```
   .
   .
mbx: mailbox
msgs: array[string]
deleted: array[int] := array[int]$new()
postOffice: mail := Service[mail]$Lookup(registry)
at postOffice eval
  mbx := mailbox$open(userID)  % userID = name & password
  msgs := mailbox$read(mbx)
end
   .
   . % between REV requests the user reads mail
   . % and decides which messages to delete
   .
at postOffice eval
  mailbox$delete(mbx, deleted) % deleted contains message numbers
end
   .
   .
```

Figure 5-2: A portion of a simple mail reader.

did not have this separation, a transmissible type such as **mailbox** could have some instances

transmitted by value-overwrite and other instances by global name. What *mbx* refers to in between the REV requests would depend on how the mailbox is transmitted to the client.

- If the mailbox is transmitted by value, *mbx* refers to the client copy of the mailbox. The service sends the copy to the client at the end of the first REV request, the client sends the copy back to the service as an argument to the second request, and finally the service returns the copy to the client for its delayed update at the end of the second request. The program in Figure 5-2 is not correct because it does not modify the actual mailbox at the service.

- If the mailbox is transmitted by global name, *mbx* refers to the actual mailbox at the service. Any messages the user deletes are deleted from the actual mailbox. Once the client-service binding is broken, *mbx* can not be used to access the mailbox.

Note that how the mailbox is transmitted affects both program semantics and the amount of communication. We eliminate this form of ambiguity by having a single transmission strategy for each type. Because all mailboxes are transmitted by global name, the program in Figure 5-2 is not ambiguous.

A consequence of the strict separation between local types and remotes types is that the reserved word REMOTE can not be associated with formal arguments to an REV request. Each formal argument has a type, and the specification for the type indicates whether it is a local type or a remote type. The application programmer can not change this attribute of a type without redefining the type. If we let the programmer associate REMOTE with formal arguments, we would have to handle situations in which an object is simultaneously transmitted by value-overwrite and by global name. Completely separating local types and remote types rules out these anomalous situations, prevents ambiguity in programs, and meets our second requirement on the naming mechanism, good documentation.

Our third requirement on the naming mechanism is convenience. As we saw in the preceding example, compile-time type checking handles global names without any assistance from the programmer. The preceding example has also shown that remote types are easy to use and understand. Under our scheme, objects with a local type are always local, even in REV requests. Objects with a remote type, in contrast, are assumed to be remote. The programmer uses this information when deciding where to insert REV requests. If a remote object happens to reside at the node executing the program, the compiler or run-time system can improve performance without changing program semantics by short-circuiting REV requests directed to that node.

Besides being easy to use and understand, remote types are easy to define and implement. A programmer makes a type remote by including the reserved word REMOTE in the interface defining the type. As explained in Section 5.4, the system generates and transmits global names and hides their existence from the programmer. The only restriction on a programmer implementing a remote type is discussed below.

Our fourth and final requirement on the naming mechanism is safety. Call by value-overwrite

implements call by sharing for REV requests, and we do not want remote types to invalidate this fact. The following example shows the problems we encounter if we do not restrict remote objects. Although the example involves REV requests sent to the client, the same problems can arise in a request sent to the service. We shall return to this point in the next section.

Suppose we let the client implement remote types. Then we must let the application programmer send REV requests to the client, so the programmer can manipulate remote objects at the client during a request sent to a service. We call the requests sent to the client *call-backs*. A call-back is a nested REV request sent from the recipient of the outer REV request to the sender of the outer request. These call-backs are expected, since they are written by the programmer. If we do not restrict remote types, the run time system must support unexpected call-backs, as explained below.

Let R be a remote object kept at the client, and let L be a mutable client object transmissible by value-overwrite. Suppose R refers to L, and both R and L are arguments to an REV request, as shown in Figure 5-3. If the REV request performs an operation on R that accesses L, the request requires two nested call-backs:

- one from the service to the client to access R;
- and another from the client to the service to access L.

The first call-back, which is an REV request written by the programmer, is expected by the programmer. The programmer, however, does not expect the second call-back, which is caused by the run-time system. The second call-back occurs because the client's copy of L may not be up to date, and we want both remote types and call by value-overwrite to implement call by sharing. If L refers to R, the two objects from a cycle, and unexpected call-backs may nest to an arbitrary depth. Performance would probably be abysmal during such a *ping-pong* match in which the process bounced back and forth between the two nodes.

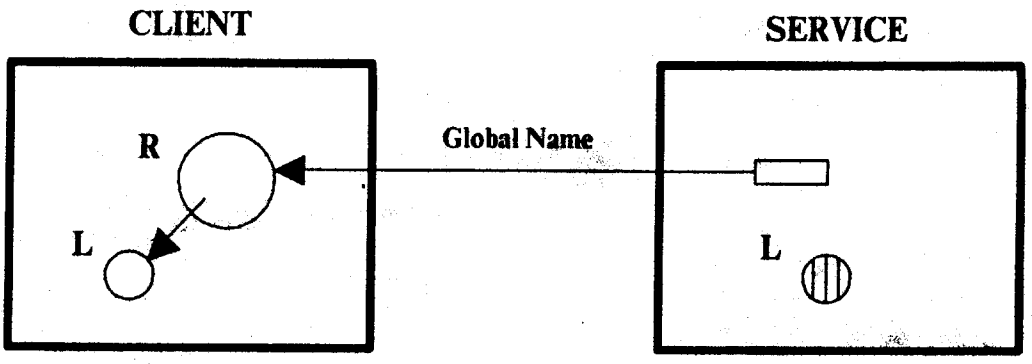


Figure 5-3: A scenario that could occur with unrestricted concrete representations for remote types.

We shall restrict the set of objects accessible from remote objects and own variables to avoid the inefficiency and complexity of unexpected call-backs. An unexpected call-back can occur only when

a mutable object transmitted by value-overwrite is accessible from a remote object or an own variable manipulated by a nested REV request.[12] Therefore, we prohibit remote objects and own variables from referring to a mutable object transmitted by value-overwrite. In other words, every object accessible from a remote object or an own variable must be immutable, nontransmissible, or a remote object. The next section provides two remote types, **RemoteRecord** and **RemoteArray**, which are useful to a programmer defining the concrete representation of a remote type. We also let programmers annotate interfaces and clusters with the new reserved word **IMMUTABLE**. This lets the compiler check the restriction on remote types and own variables at link time. If the client does not implement remote types and does not execute nested REV requests, this restriction does not apply.

5.3 System-Defined Remote Types

Records are extremely useful to a programmer choosing the concrete representation of a local type, since the representation often contains several components of different types. Arrays are also useful, especially when the number of components with some type can not be determined at compile time. A programmer choosing the concrete representation of a remote type will need similar type constructors, so we provide the following system-defined remote types: **RemoteRecord** and **RemoteArray**. **RemoteRecord** is identical to the **record** type except for transmissibility. Records are transmitted by value-overwrite; RemoteRecords are transmitted by global name. RemoteArrays differ from arrays in the same way.

RemoteRecords and RemoteArrays give the application programmer fine-grained control over the location of client data, which is an important consequence of global names. These types let the programmer keep client objects at a service between REV requests and refer to the entire collection of objects with a single global name. We illustrate this ability by using RemoteRecords, with the understanding that RemoteArrays can be used in a similar fashion.

RemoteRecords, like records, are declared as a list of components and their types. Assume a programmer wants to keep two real numbers and a string at a service. The following RemoteRecord declaration provides this capability:

```
info = RemoteRecord[x: real, y: real, name: string]
remData: info
```

RemData is a variable whose type is a three-component RemoteRecord. A RemoteRecord (or a record) is created by enclosing the initial values for all components in curly brackets. The following REV request creates a RemoteRecord at a graphics service:

```
xCoordinate: real := . . .   % initialize xCoordinate
aNode: graphics := Service[graphics]$Lookup("room 212")
at aNode eval remData := info${x: xCoordinate*5.5, y: 2.2, name: "Jones"} end
```

[12] Although a nested REV request can not access an own variable directly, it can call a client routine that accesses an own variable.

After the above REV request completes, the client variable *remData* contains the global name for the RemoteRecord at the graphics service. A programmer uses "dot" notation to access and update RemoteRecord components:

```
at aNode eval remData.x := remData.y + 2.0 end
```

Unless defaults are provided, a programmer can manipulate a RemoteRecord only in an REV request executed by the node containing the RemoteRecord. Note that the RemoteRecord lets the programmer group together all client objects at the service. If a single procedure deals with several services, this structuring of client data can help the programmer remember which object is at which service. This in turn can help eliminate errors that are difficult to detect before run time.

RemoteRecords let an REV programmer send information from the client to the service once, but access it many times. Consider a remote bitmap display that can show points, lines, and characters. Let **window** be a remote type. Assume the programmer wants to display characters in an unusual font that the display does not directly support. Furthermore, assume fonts are immutable and transmissible by value-overwrite. Instead of sending the unusual font with each REV request that displays characters, the programmer can send it once:

```
remoteInfo = RemoteRecord[value: font]

display: window
testFont: remoteInfo
newFont: font := . . .   % initialize newFont
displayService: superGraphics := Service[superGraphics]$Lookup("room 212")

at displayService eval
  display := window$create()
  testFont := remoteInfo${value: newFont}
end
```

The preceding REV request creates a window and a RemoteRecord at the service and returns only their global names to the client. If the programmer does not modify the RemoteRecord, the unusual font remains at the service until the client-service binding is broken. The next REV request uses this font to display a character in the remote window:

```
at displayService eval
    .
    .
  window$showChar(display, charCode, x, y, testFont.value)
    .
    .
end
```

The client transmits only a global name (and not the entire font) when the variable *testFont* appears in an REV request.

Besides reducing communication from the client to the service, RemoteRecords can reduce communication in the other direction. RemoteRecords let a programmer keep the results computed by an REV request at the service and use these results during later REV requests. Without RemoteRecords and global names, the results are returned to the client and sent to the service with each REV request that accesses the data.

If RemoteRecords and RemoteArrays are not built into the programming language, programmers can get fine-grained control over the location of client data by constructing similar mechanisms. Such mechanisms are useful in constructing distributed systems, and we supported them directly to enhance their convenience and expressive power.

We end this section with a reminder. To meet the safety requirement on our naming mechanism, we restricted the objects accessible from a remote object. A remote object can not refer to a mutable object that can be transmitted by value-overwrite. An application programmer using RemoteRecords and RemoteArrays must be aware of this compiler-enforced restriction.

5.4 Implementation

In addition to the support provided for every abstract data type, the system has two extra tasks for each remote type **T**:

1. hide global names from the programmer implementing T; and
2. generate and manage global names for objects of type T.

This section describes each task in turn.

To simplify programming, we completely hide global names from someone implementing a remote type. A programmer implementing a remote type T defines **LocalRep**, which is the representation for local instances of T. **LocalRep** does not involve global names, but the compiler uses **LocalRep** to generate the actual concrete representation that does involve global names:

```
rep = oneof[local: LocalRep,
            remote: GlobalName[T]]
```

The actual concrete representation for a remote type is a oneof type, since the representation for a remote object depends on whether the current node is its home. The system-defined type **GlobalName** is parameterized by the type of object that corresponds to the global name.

Before discussing the conversions between concrete objects and abstract objects for remote types, we review the conversions for local types. Let **S** be a local data type. Two routines, which are automatically generated by the compiler, convert between the concrete viewpoint (**rep**) and the abstract viewpoint (**S**):

```
S$up = proc (rep) returns (S)

S$down = proc (S) returns (rep)
```

These routines, which do not cause any run-time computation, are available only in a cluster implementing type S. They inform the compiler that an object of type S is going to be viewed abstractly or concretely. For example, a programmer that needs to manipulate an argument of type S can use **S$down** to view the concrete object. The programmer can later use **S$up** to convert the concrete object into an abstract object before returning it as a result.

Up and down for remote types require run-time computation and have different type specifications than the corresponding routines for local types. In a cluster implementing remote type T, a programmer may use the following conversions:

```
T$up = proc (arg: LocalRep) returns (T)

T$down = proc (arg: T) returns (LocalRep) signals (NotLocal)
```

Figure 5-4 shows how to implement these routines. Although the implementations violate the type system, the violations are acceptable because these routines are compiler-generated. **T$up** creates and returns a oneof with a "local" tag that refers to its argument. **T$down** checks a oneof and returns the component if the object is local. Otherwise, the oneof refers to a global name, and **T$down** raises an exception. Neither **T$up** nor **T$down** involves REV.

```
up = proc (arg: LocalRep) returns (T)
   concrete: rep := rep$make_local(arg)
   % the next statement does not type-check, since we should "up" concrete first
   % this routine is compiler-generated and therefore OK
   return (concrete)
end up

down = proc (arg: T) returns (LocalRep) signals (NotLocal)
   % the next statement does not type-check, since we should "down" arg first
   % this routine is compiler-generated and therefore OK
   tagcase arg
      tag local (obj: LocalRep): return(obj)
      tag remote signal NotLocal
   end
end down
```

Figure 5-4: Up and down for a remote type T.

Besides defining the local representation (**LocalRep**), a programmer implementing remote type T implements the routines defined by interface T. The programmer, who can use **T$up** and **T$down** in these routines to convert between the abstract viewpoint (T) and the concrete viewpoint (**LocalRep**), does not need to know that the actual concrete representation is a oneof.

Because down for a remote type can raise an exception whereas down for a local type never raises an exception, implementing a remote type is slightly different than implementing a local type in CLU. Assume a remote type contains a binary operation (**T$op**) that may involve two objects at different nodes. Furthermore, assume **T$op** must be executed at the home of the first argument. Then the programmer implementing **T$op** can not always apply **T$down** successfully to the second argument. If the second argument is remote, the programmer must view it as an abstract object and use REV requests to manipulate its abstract state. The programmer would use the following system-defined operation to determine the node containing the argument:

```
T$home = proc (arg: T) returns (G2776),
```

where G2776 is a name that is unique in the current environment and bound to the following service definition:

```
G2776 = service is T end
```

Such a service exports T and, by default, the built-in types.

Having discussed how the system hides global names from programmers, we now consider how service capabilities and global names are transmitted between nodes. A service capability consists of a node name and an identifier interpreted by the node:

- **NodeID**: a system-wide unique name
- **CapabilityID**: an identifier unique to the above node for all time

NodeID must contain enough information for the client to be able to send a message to the service. For instance, a host name or an internet address is acceptable. **CapabilityID** lets the client invoke REV requests at the service. If different service capabilities correspond to different address spaces, **CapabilityID** also selects the appropriate address space.

A global name, which is the external representation for each remote type, consists of a service capability and an identifier interpreted by the service:

- **ServiceCapability**: defined above
- **UniqueID**: an identifier unique to the above service

Recall that a global name lets one node refer to an object kept on another node.

A service implementing remote types maintains two mappings:

- one mapping converts remote objects to global names (mapping G); and
- the other mapping converts global names to remote objects (mapping H).

Mappings G and H are inverses. When a service capability becomes invalid, all entries in G and H that refer to the service capability are removed from the mappings. We explain below how the mappings are used to encode and decode remote objects.

The **encode** routine for a remote type, which is automatically generated, converts the concrete representation for a remote object (a oneof) into the external representation for the object (a global name). If the oneof has a "remote" tag, the oneof refers to a global name. In this case, **encode** simply returns the global name. Otherwise, the oneof has a "local" tag and refers to an object. In this case, **encode** uses G to determine if the remote object already has a global name under the current service capability. If so, **encode** returns the global name. Otherwise **encode** generates a new global name, inserts the appropriate entries into G and H, and then returns the new global name.

The **decode** routine for a remote type, which is also automatically generated, converts the external representation for a remote object (a global name) into the concrete representation for the object (a oneof). If the service capability for the global name is not the current service capability, **decode** returns a oneof with a "remote" tag that refers to the global name. The validity of the global name is not determined. If the service capability for the global name is the current service capability, the global name is valid by definition. **Decode** uses H to map the global name into an object that exists at the service and then returns a oneof with a "local" tag that refers to the object.

The system, rather than the programmer, makes a remote type transmissible. The **encode** routines for remote types differ only in the types of the remote objects and oneofs they deal with. Because the encoding algorithm is always the same, the compiler can generate **encode** for each remote type. The same applies to **decode** routines for remote types. Since the compiler also generates **get** and **put** for all transmissible types, the compiler makes each remote type transmissible without any help from the programmer implementing the remote type.

5.5 Summary

We developed a naming mechanism based on global names that lets one node refer to an object residing at another node. We associated the ability to have a global name with types, rather than with individual objects or formal arguments to REV requests, and achieved the following objectives:

1. *Transience:* Once a client-service binding is broken, neither node refers to objects at the other node. This means, however, that remote data types implement call by sharing for an REV request only while the relevant service capabilities are valid.

2. *Good Documentation:* Every object that may be remote is apparent to someone reading a program.

3. *Convenience:* Remote types are easy to use and implement.

4. *Safety:* Call by value-overwrite still implements call by sharing for REV requests. A compiler-enforced restriction on the types of objects accessible from a remote object guarantees this objective.

A programmer makes a type remote by including the reserved word **REMOTE** in the interface that defines the type and clusters that implement the type.

An important consequence of global names is that they give the application programmer fine-grained control over the location of client data. **RemoteRecord** and **RemoteArray** are system-defined remote types that can transform an immutable local type, such as **integer**, into a remote type at the cost of an explicit indirection. These two remote types also let a programmer keep a collection of objects at a service and refer to them with a single global name. A programmer using these types can send information from the client to the service once, but access it during many different REV requests. A programmer using these types can also keep the results of an REV request at a service beyond the completion of the request.

Chapter Six

An Extended Example

To familiarize the reader with REV requests and remote objects, we present an extended example of their use. In this chapter we define a service called **news** by specifying the interfaces it exports. Each news service contains a database of newspaper articles. An authorized user at another node can query this database and examine the articles selected by the query. The program fragments in the chapter show the usefulness of REV compared to RPC's for applications that support queries to such a remote database. The program fragments also show how RemoteRecords give the application programmer fine-grained control over the location of client data, which is important because remote types can reduce the amount of communication between nodes. We begin by describing the interfaces exported by each news service.

6.1 Declarations

Recall that a two-level description applies to services. A service definition is a set of interfaces, and an interface specifies a collection of routines. The following statement declares the interfaces each news service exports:

```
news = service is article, articleDB, time&date, set, RemoteRecord, built-ins end
```

We discuss the more interesting interfaces in this service definition below.

Figure 6-1 contains the **article** interface. An **article** instance is an object with five components: subject, author, priority, timestamp, and body. The priority of an article describes its overall importance. We assume that a priority is a small positive integer. Because an article body might be very long, we provide an additional routine (**shortBody**) that returns only the beginning of the article.

Figure 6-2 contains the **articleDB** interface, which defines a remote type. Thus a program running at a client can refer to an instance of this type that resides at a service. An instance of type **articleDB** is a repository of articles that efficiently performs certain searches. Some of the routines in the interface generate a new articleDB from an existing articleDB and an attribute value (**artByPriority** through **artSince**). Other routines return information about a particular articleDB (**subjects** through **articles**). The reserved word IMMUTABLE at the top of the interface tells us that an articleDB can not change its state. There are no routines that modify an articleDB, and the only way to create one is to use **accessDB**.

```
article = immutable location_independent interface is

  create = proc (string, string, int, time&date, string) returns (article)
    % args are subject, author, priority, timestamp, & body

  % routines to access components of an article

  subject = proc (article) returns (string)
  author = proc (article) returns (string)
  priority = proc (article) returns (int)
  timestamp = proc (article) returns (time&date)
  fullBody = proc (article) returns (string)  % return the entire body of the article
  shortBody = proc (article) returns (string)  % return only the first three lines

  % value-overwrite transmission

  encode = proc (article) returns (string)
  decode = proc (string) returns (article)

end article
```

Figure 6-1: The article interface.

The remaining interfaces exported by a news service consist of common data types and system-defined data types. The **time&date** and **set** interfaces, shown in Figures 6-3 and 6-4, are the usual ones. The **RemoteRecord** type is a remote type that is identical to the **record** type except for transmissibility. **Built-ins** is the set of types provided in every implementation of the programming language. It includes integers, reals, booleans, characters, records, arrays, strings, and procedures.

6.2 Sample Programs

We present five examples that use a news service. These examples are largely independent, but together they show how REV and remote types are useful to an application programmer dealing with one or more news services.

Our first example, which concerns a user who is authorized to access a database of articles from the New York Times, shows the general template for interacting with such a database. Consider a program that accepts a query from the user, transforms the query into an REV request, sends the request to the news service, and finally presents the selected articles to the user. A straightforward implementation of this task has several problems, especially when a query selects a large number of long articles. The user must wait for all the selected articles to be sent to the client, and the client must have enough (secondary) memory to store the selected articles. We can improve response time and reduce the storage burden on the client by returning only part of the results to the client. The client initially receives a collection of abbreviated articles, and the user can subsequently request the full version of any article in the collection.

```
articleDB = remote immutable interface is

  availableDBs = proc () returns (set[string])
  accessDB = proc (string, string, string) returns(articleDB) signals (accessDenied)
    % arguments are DB name, user name, password

  % each of the following six routines returns a subset of its first argument
  % articles that "match" the second argument form the result
  % neither argument is modified

  artByPriority = proc (articleDB, int) returns (articleDB)
  artBySubject = proc (articleDB, string) returns (articleDB)
  artByAuthor = proc (articleDB, string) returns (articleDB)
  artByKeyword = proc (articleDB, string) returns (articleDB)
  artBefore = proc (articleDB, time&date) returns (articleDB)
  artSince = proc (articleDB, time&date) returns (articleDB)

  % the following routines determine what a DB contains; each returns
  % the number of articles with each value of the attribute

  subjects = proc (articleDB) returns (set[record[subject: string, number: int]])
  authors = proc (articleDB) returns (set[record[author: string, number: int]])
  priorities = proc (articleDB) returns (set[record[priority: int, number: int]])
  keywords = proc (articleDB) returns (set[record[keyword: string, number: int]])

  % additional information about a DB

  timeRange = proc (articleDB) returns (time&date, time&date)
  noiseWords = proc (articleDB) returns (set[string])
  size = proc (articleDB) returns (int)
  fetch = proc (articleDB, int) returns (article) signals (bounds)
  articles = iter (articleDB) yields (article)

end articleDB
```

Figure 6-2: The articleDB interface.

```
time&date = immutable interface is

  now = proc () returns (time&date)
  within = proc (time&date, time&date, time&date) returns (bool)
    % determine whether the first pair of times differs by at most the third time
  before = proc (time&date, time&date) returns (bool)
  toString = proc (time&date) returns (string)
  fromString = proc (string) returns (time&date) signals (BadFormat)

  % value-overwrite transmission

  encode = proc (time&date) returns (int)
  decode = proc (int) returns (time&date)

end time&date
```

Figure 6-3: The time&date interface.

We illustrate this enhancement with the following program fragment, but for simplicity, we do not discuss the conversion from user queries to executable program fragments. Figure 6-5 shows the

```
set = location_independent interface [t: type]
  where t has equal: proctype (t, t) returns (bool)
  is

  create = proc () returns (set[t])
  insert = proc (set[t], t)
  delete = proc (set[t], t)
  member = proc (set[t], t) returns (bool)
  elements = iter (set[t]) yields (t)
  size = proc (set[t]) returns (int)
  any = proc (set[t]) returns (t) signals (empty)
  union = proc (set[t], set[t]) returns (set[t])
  intersection = proc (set[t], set[t]) returns (set[t])
  difference = proc (set[t], set[t]) returns (set[t])

  % value-overwrite transmission

  encode = proc (set[t]) returns (array[t]) where t has
    put = proctype (REVcontext, t)
  decode = proc (array[t]) returns (set[t]) where t has
    get = proctype (REVcontext) returns (t)

end set
```

Figure 6-4: The set interface.

---

code corresponding to the query "What articles containing the word 'movie' were written by Vincent Canby?" After locating a New York Times database, **queryProc** opens the database and can access (via *articles*) all the articles in the database. Because the user does not want all the articles, **queryProc** successively refines the set of articles that it returns to the user. First, **queryProc** determines the articles written by Canby. From this set, it determines the articles that contain the word "movie." Finally, **queryProc** creates a set of abbreviated articles by applying **shortBody** to each selected article. The procedure **as$addh** appends its second argument (a string) onto the high end of its first argument (an array of strings), thereby modifying its first argument. The client variable *shortText* has a local type, an array of strings. After the REV request completes, this variable refers to objects residing at the client. The client variable *articles*, in contrast, has a remote type. After the REV request completes, *articles* contains a global name that refers to the results at the news service. Thus the service sends the client an array of strings (i.e., an array of abbreviated articles) and only the global name for the articleDB containing the full articles.

We assume an articleDB keeps the articles it contains in some fixed order, such as reverse chronological order. Fetching the full version of the *ith* article requires only the short REV request in Figure 6-6. The client sends a global name (*articles*) and an integer (*i*) to the news service, which returns the body of the specified article in the variable *full*.

This example has shown the general template for interacting with a news service. The program opens the database, selects articles according to a query, performs some computation, and finally returns the results. In this example, the computation involved fetching the abbreviated version of each

```
DB = articleDB              % some equates
ss = set[string]
as = array[string]

queryProc = proc (user, password: string) returns (DB, as) signals (NotAvailable)

  begin transaction
    nyt: news := Service[news]$Lookup("new york times") abort resignal NotAvailable
    articles: DB
    shortText: as

    at nyt eval
      articles := DB$accessDB("new york times", user, password)
        except when accessDenied: abort signal NotAvailable end
      articles := DB$artByAuthor(articles, "Vincent Canby")
      articles := DB$artByKeyword(articles, "movie")
      shortText := as$create()
      for art: article in DB$articles(articles) do
        as$addh(shortText, article$shortBody(art))
      end
    end

    return (articles, shortText)

  end except when failure (s: string): signal NotAvailable end

end queryProc
```

Figure 6-5: A program to determine movie reviews by Vincent Canby.

```
    .
    .
full: string
    .
    .
at nyt eval
  desired: article := DB$fetch(articles, i)
  full := article$fullBody(desired)
end
    .
    .  % use variable full
    .
```

Figure 6-6: An REV request that fetches a full article body.

selected article and collecting the abbreviated articles in an array. We used REV to compose several articleDB operations into a single request, and we used a remote type (**articleDB**) to avoid returning all the information to the client immediately.

Our second example shows how REV supports general queries much more efficiently that an RPC approach. The procedure in Figure 6-7 answers the question "Who wrote the *New York Times* articles with the highest priority?" Once the client-service binding is established, the REV approach

requires one remote invocation. An RPC implementation requires three remote invocations: the first to open the database, the second to get the priorities, and the third to fetch all articles with the highest priority.[13] The RPC implementation returns all articles with the highest priority, while the REV implementation returns only the authors of these articles. Unless the **articleDB** interface or the **news** service is extended, RPC programs can not have the service project the results of a query onto one or more article components. An REV programmer, in contrast, can combine primitive operations in an arbitrary way to have the service do the projection. This reduces communication and increases performance.

```
queryProc = proc (user, password: string) returns (set[string]) signals (NotAvailable)

  DB = articleDB            % some equates
  ii = record[priority: int, number: int]
  sii = set[ii]
  ss = set[string]

  begin transaction
    authors: ss
    nyt: news := Service[news]$Lookup("new york times") abort resignal NotAvailable

    at nyt eval
      articles: DB := DB$accessDB("new york times", user, password)
        except when accessDenied: abort signal NotAvailable end
      priorities: sii := DB$priorities(articles)
      if sii$size(priorities)=0 then abort signal NotAvailable end
      info: ii := sii$any(priorities)
      max: int := info.priority
      for info: ii in sii$elements(priorities) do
        if info.priority>max then max := info.priority end
      end
      articles := DB$artByPriority(articles, max)      % ***
      authors := ss$create()
      for art: article in DB$articles(articles) do
        ss$insert(authors, article$author(art))
      end
    end  % REV request

    return (authors)

  end except when failure (s: string): signal NotAvailable end

end queryProc
```

Figure 6-7: A program to determine authors of high priority articles.

The following approach, which an RPC programmer may use, sends less data than the preceding RPC approach but requires an additional remote invocation. The extra invocation occurs just after the assignment statement marked with three asterisks in Figure 6-7. This invocation uses

[13] In addition to the procedures in the **articleDB** interface, we let an RPC programmer have a single remote procedure that returns all articles matching a given priority, subject, author, keyword, and timestamp combination.

articleDB$authors to return to the client the authors and the number of high priority articles each wrote. The REV approach, however, will outperform this RPC approach because it does the same processing but requires fewer remote invocations and transmits less data to the client.

This example has shown how REV can outperform an RPC approach when the "right" remote procedure is not exported by the service. An REV programmer can construct such a procedure and have it execute at the service. An RPC programmer, in contrast, must use the routines exported by the service and do the remaining processing at the client. The RPC approach often requires several remote invocations, while the REV approach often requires only one. Because an REV request can often eliminate unneeded information at the service, an REV approach usually needs less communication than an RPC approach.

Our next example is the first one to use a RemoteRecord, and it does so to keep client data at the news service between REV requests. Suppose the user always wants to disregard certain articles. For instance, the user may want to avoid all news summaries, front page layouts, photo captions, and articles containing certain keywords. One method of implementing this capability is to have the client augment each query with these standard restrictions before using REV to execute the query. The disadvantage of this approach is that the client sends the same restrictions to the service with each request. A more efficient method is to install the standard restrictions at the service once and access them when necessary.

Assume the standard restrictions consist of a procedure with the following type specification:

```
standardQuery = proc (articleDB) returns (articleDB).
```

The following REV request installs **standardQuery** at the service:

```
filter = proctype (articleDB) returns (articleDB)
rFilter = RemoteRecord[proc: filter]     % remote filter
sq: rFilter
articles: DB
   .
   .
at nyt eval sq := rFilter${proc: standardQuery} end
```

The next REV request restricts a query by invoking the procedure in the RemoteRecord on the articles that match the query:

```
at nyt eval
  articles := . . .          % initialize articles from user query
  articles := sq.proc(articles)  % apply standardQuery
     .
     .
end
```

Once installed at the service, the copy of the procedure **standardQuery** remains there until the client or service breaks the binding between them. The client transmits only a global name (and not the entire procedure) when the variable *sq* appears in an REV request.

This example has shown how RemoteRecords let the application programmer keep client data at a

service between REV requests. We used a RemoteRecord to avoid sending the same object to the service with every REV request. RemoteRecords may also be used to keep (part of) the results computed by an REV request at the service.

The remaining two examples show how REV can integrate different information sources. Suppose a user can access three news services: the New York Times, Associated Press, and United Press International databases. REV can hide the location of articles by sending each query to all three services and then merging the responses. Figure 6-8 shows part of a program that combines articles from all three news services. The program respects the autonomy of individual news services, which may be supported by completely different organizations. As long as the nodes export the same service (i.e., news), integrating disjoint databases for queries is straightforward. REV lets the query language be independent of the news services, which simplifies application programming.

```
DB = articleDB                  % an equate
   .
   .
nytUp, apUp, upiUp: bool := false
nytData, apData, upiData: DB

nyt: news := Service[news]$Lookup("new york times")
  except when NotAvailable: nytUp := false end
ap: news := Service[news]$Lookup("AP")
  except when NotAvailable: apUp := false end
upi: news := Service[news]$Lookup("UPI")
  except when NotAvailable: upiUp := false end

% If no accessible databases, stop here
if bool$not(nytUp | apUp | upiUp) then signal NotAvailable end
   .
   .  % Submit queries to all news services that are up.
   .  % As before, fetch full articles on demand.
   .
```

Figure 6-8: Integrating three different news services.

If the three nodes exported different different kinds of news services, REV could be used to hide their differences from the end user, assuming each node exported enough general operations. An REV programmer can construct new "remote procedures" that execute at a service, while an RPC programmer can not.

Our final example uses a nested REV request and compares information kept in different databases. A user who wants to know what subjects are currently covered by the AP database but not by the UPI database would execute the code in Figure 6-9. The outer REV request is sent to the AP service, while the nested REV request is sent from the AP service to the UPI service. Relocating the execution of procedure **subjects** strips the article count at each news service. An implementation without REV has higher communication costs because it also sends the article count information to the client.

```
ss = set[string]   % an equate

subjects = proc (db: articleDB) returns (set[string])
  subjCount = record[subject: string, number: int]
  scts: set[subjCount] := articleDB$subjects(db)
  DBsubjects: ss := ss$create()
  for sct: subjCount in set[subjCount]$elements(scts) do
    ss$insert(DBsubjects, sct.subject) end
  return (DBsubjects)
end subjects
    .
    .
begin transaction
  answer: ss
  at ap eval
    apArticles: articleDB := DB$accessDB("AP", user, password)
        except when accessDenied: abort signal NotAvailable end
    APsubjects: ss := subjects(apArticles)
    UPIsubjects: ss

    at upi eval
      upiArticles: articleDB := DB$accessDB("UPI", user, password)
          except when accessDenied: abort signal NotAvailable end
      UPIsubjects := subjects(upiArticles)
    end  % nested REV request

    answer := ss$difference(APsubjects, UPIsubjects)
  end  % outer REV request
  .
  .
end except when failure (s: string): signal NotAvailable end
```

Figure 6-9: Comparing AP and UPI subjects.

This example shows how nesting two REV requests might reduce communication in comparison to two separate REV requests. If two separate requests are used, the AP subjects and the UPI subjects are sent to the client, which calculates the difference between these sets. If the requests are nested, the UPI subjects are sent to the AP service, which calculates the difference between the UPI subjects and the AP subjects and returns this difference to the client. If the difference is significantly smaller than the set of the AP subjects, nesting the requests can significantly reduce the size of the second result message. The first request message, however, is larger when the requests are nested, as this request message must contain the code portion of the nested request.

6.3 Discussion

The examples in this chapter attempted to convey two points. First, REV gives the programmer fine-grained control over the location of processing. Once REV has been used to give a program the desired semantics, the programmer can partition the program into fragments for local and remote processing to improve performance and reduce communication. The ability to nest REV requests can improve performance and reduce communication even further.

Second, remote objects give the programmer fine-grained control over the location of data, which is very useful when the programmer must reduce the amount of communication between nodes. Either the service or the client may introduce remote objects. A service introduces remote objects by exporting a remote data type such as **articleDB**. Such remote data types support the partial transmission of results. This technique, which can improve response time, reduces communication when a user does not examine all the results in detail. A client can introduce remote objects by using RemoteRecords and RemoteArrays, which let an REV programmer keep an object at the service between REV requests. An REV programmer can use these types to keep the results of an REV request at a service or to avoid sending the same data with each request to the same service.

REV and remote types are simple mechanisms that give an application programmer control over how and where a program executes in a distributed environment. Some of this control is used to give a program the desired semantics, but the rest is available for tuning performance.

Chapter Seven

Experience and Evaluation

This chapter summarizes our experience with REV. It describes our prototype REV implementation, presents representative performance measurements, and offers advice to future REV implementors. We also give advice to service programmers supporting REV and application programmers using REV. Finally, we evaluate the novel ideas in the thesis and present areas for further work.

7.1 A Prototype Implementation

We constructed a prototype implementation of an REV mechanism to facilitate experimentation, debug algorithms, and evaluate the amount of work required to implement a production system. Absolute performance was not a concern. Our computing environment consisted of two Symbolics 3600 Lisp Machines connected by a local area network. Both machines supported the LISP dialect ZetaLisp [45]. We used LISP because of its flexibility and its uniform treatment of code and data. For simplicity, we assumed the existence of strong type checking and prohibited exceptions. Conditional statments were allowed but loops were not. Our experimentation language contained the following built-in types: integers, characters, booleans, strings, records, arrays, and procedures. All these types were local, transmissible types. We added new types as necessary. Some of the new types, such as services and windows, were remote types.

Most of the compile-time requirements for REV were implemented. We built a use-definition package and converted implicit REV requests into explicit REV requests. The code portion of each REV request was type-checked against the definition of the service class that would execute the request. Stubs were generated for both the client and the service. We did not support client-supplied routines or the compilation of interface and service descriptions. A small database of procedure, interface, and service specifications was built manually. Our procedure folding algorithm did not handle exceptions or local transfers of control.

Except for a name-lookup service and a transaction mechanism, the run-time requirements for REV were supported. We implemented call by value-overwrite and supported remote types by generating and transmitting global names. Source-level procedures were transmissible. For simplicity, we did not transmit compiled code. The external representation for source code was a compressed form of list structure. The remaining portion of the run-time environment consisted primarily of existing Lisp Machine software. The chaos datagram package transferred packets from one node to another.

Ensuring that the code portion of every decoded request was compatible with ZetaLisp obviated the need for a new interpreter. Requests were executed by calling **apply**, which invoked the resident ZetaLisp interpreter.

Although absolute performance was not a concern, we measured the time to perform simple remote operations:

`packet exchange`	`20-25 msec.`
`null REV request`	`100 msec.`
`x := x*x`	`121 msec.`

Our REV implementation was based on the ability to send a string from one Lisp Machine to another and receive a string reply. When both strings were empty, the round-trip time for a packet exchange was 20-25 msec. A null REV request, which has no arguments, no results, and an empty procedure body, took a tenth of a second. Executing a remote integer multiplication took slightly longer. We used six hash tables to implement call by value-overwrite: three at the client and three at the service. Creating a small hash table took about 10 milliseconds, which shows that hash table creation accounted for much of the overhead on each REV request.

To test our thesis that REV may be viewed as an optimization, we compared the time to execute an REV request with the time to execute the corresponding collection of RPC's. Since we had no RPC package, we simulated an RPC with a simple REV request and used the following formula to estimate its performance;

```
RPCtime = 0.5 * (REVtime - NETWORKtime) + NETWORKtime
```

NETWORKtime is 20 milliseconds, which is the time for a packet exchange. Except for the packet exchange, we assumed that the REV request was twice as expensive as an RPC. For the objects we transmitted between nodes, this is an upper bound on the performance difference between call by value and call by value-overwrite.

Figure 7-1 contains a procedure with three REV requests. An array of size three (*smallArray*) was sent to the service twice and returned to the client twice. The average time to execute P, which incurs three REV overheads, was 466 milliseconds. Using the above formula, we estimated that the average time to execute P with three RPC overheads was 263 milliseconds. If we assume the RPC's use call by value, the array is sent to the service twice and never returned to the client. Executing P as a single REV request, as shown in Figure 7-2, incurs only one REV overhead. The array is sent to the service and returned to the client only once. The average time to execute P in this case was 183 milliseconds. Figure 7-3 summarizes the differences between the two REV executions and the estimated RPC execution.

In discussing the compile-time and run-time requirements of an REV mechanism, the previous chapters have conveyed much of the detailed structure of our prototype implementation. At this

```
P = proc (aNode: built-ins, a: array[int], x, y: int) returns (int)
  x := at aNode eval a[x] end
  y := at aNode eval a[y] end
  x := at aNode eval x*y end
  return(x)
end P

   .
   .
   .

smallArray: array[int] := . . .   % initialize smallArray
aNode: built-ins := Service[built-ins]$Any()

P(aNode, smallArray, 1, 2)
```

Figure 7-1: A procedure with three RPC's.

```
P = proc (a: array[int], x, y: int) returns (int)
  x := a[x]
  y := a[y]
  x := x*y
  return(x)
end P

   .
   .
   .

smallArray: array[int] := . . .   % initialize smallArray
aNode: built-ins := Service[built-ins]$Any()

at aNode eval P(smallArray, 1, 2) end
```

Figure 7-2: A single REV request instead of the three RPC's.

---

program	arrays transmitted	integers transmitted	average time
3 REV's	4	7	466 msec.
3 RPC's	2	7	263 msec. (estimated)
1 REV	2	3	183 msec.

Figure 7-3: An estimated comparison between REV and RPC's.

point, we present our reflections on implementing an REV mechanism. Whenever possible, REV

implementors should augment an existing RPC implementation instead of beginning from scratch. Much of the compile-time and run-time support for RPC, including any modifications done for high performance, can be used for REV. Communication primitives and protocols for RPC are applicable to REV. Stub generation is similar in both schemes. For immutable types, call by value is equivalent to call by value-overwrite. Unless optimizations are desired, no changes are necessary to their **put** and **get** procedures. Slight changes have to be made to the **put** and **get** procedures of mutable types using call by value-overwrite. The encoding and decoding contexts (**REVcontext**) must be extended to manage external names, as explained in Section 3.4.1. Excluding the need for an interpreter, the remainder of the conversion process consists of implementing procedure transmission and accommodating syntax extensions, service definitions, and implicit REV requests (use-definition analysis) in the compiler.

The introduction of REV need not degrade RPC performance. REV requests that are really RPC's could be recognized and treated as such. The stub generator can avoid much of the encode and decode overhead by capitalizing on REV requests that have simple argument or result types. In short, application programmers should not have to pay for the generality of REV unless REV is required.

7.2 Hints for the Service Programmer

The following discussion contains our advice to service programmers supporting clients that use REV. We first discuss the kinds of routines a service should export and then discuss how the service programmer can constrain the application programmer. Finally, we consider the relationship between service programmers and application programmers.

When RPC's are used, a service exports routines tailored to a particular application. Because not all applications will use the full power of REV, services should continue to export these specialized routines.

In addition to specialized routines, REV services should export general routines and let the application programmer compose them arbitrarily. Even inexpensive routines can be exported, since good application programmers will structure their code to minimize REV overhead. Services should also export routines that take routines (or closures) as arguments, since they let the application programmer customize existing service routines. Iterators, for example, can and should be exported.

An application programmer, who often wants maximum flexibility when partitioning a program, wants the service to export as many interfaces as possible. On the contrary, a service designer does not want to burden each service node with extraneous requirements. A useful compromise is to export (some of) the interfaces whose implementations are guaranteed to exist at the service.[14] Although

[14] As explained below, a service designer may withhold service interfaces to constrain the application programmer.

this approach seems to put implementation before design, the relevant interfaces are known at design time. A simple analysis of the interfaces exported by a service yields a fair amount of information, under the assumption that a service able to contain an object of type T implements interface T. By this assumption, the service implements all types mentioned in the exported interfaces, not only the types defined by the interfaces. The additional types mentioned in the interfaces for the unexported types must also exist at the service, and so on. In technical terms, the net result is the reflexive, transitive closure of type dependence beginning with the exported interfaces. A set of interfaces is *closed* if and only if it equals its type closure. Exporting this closed interface set places no new requirements on a service yet provides application programmers with a moderate to large collection of relevant routines. The routines are relevant because a program that encounters an object of type T is likely to invoke routines from interface T. We suggest that services export closed interface sets. Taking the closure of a service definition never burdens a service yet may improve the utility of the service to an application programmer.

Having discussed some of the reasons for enlarging a service definition by taking its type closure, we now consider some restrictions on service definitions and the particular benefit each provides. Service designers may not want to burden their nodes with nested REV requests that can not be short-circuited. The syntax for service definitions could be extended to include such a restriction. Another restriction is to remove the **RemoteRecord** and **RemoteArray** interfaces from a service definition. This prevents programmers from storing client data at the service between REV requests.

The responsibility for providing a useful computational environment to end users rests with both service programmers and application programmers. REV permits a flexible division of labor, depending on their relative numbers. For example, service programmers should view application programmers as independent service programmers who might unexpectedly provide new ideas or implementations. Heavily used REV requests are candidates for new service routines. Because the design, implementation, and debugging of these routines are already done, a service programmer merely examines existing candidates instead of developing code from scratch.

7.3 Hints for the Application Programmer

We now present our recommendations to application programmers using REV. The main rule is to *assume that everything is remote and scattered*. Different service capabilities should be thought of as denoting different services, and each remote object should be thought of as existing at a distinct service. Capitalizing on REV under these worst case conditions requires a programmer to group together operations on a single remote object. The next step is to group together operations on objects known (or thought) to be at the same service. The desirability of an intervening operation that manipulates client data depends on the probability that it is exported by an arbitrary service as well as its effect on data communication.

The degree to which these guidelines are followed depends on the trade-off between readability and performance. The structure of an REV-based program may look unusual to a traditional programmer, who might object to the rigid discipline we are apparently advocating. Two points need to be remembered. First, the guidelines apply only to instances of remote types. In most parts of most programs, we expect such objects will be in the minority. Second, what appears unusual at first glance may become acceptable once we gain experience with REV. In fact, the additional discipline mandated by performance considerations may actually improve programming style. Operations involving a particular remote object will be limited to certain sections of the program. Although programs may be harder to write (initially), readability and maintainability should improve.

Adding REV requests to an existing program will often require local code rearrangement. Reordering statements wherever possible to meet the above guidelines will help. Other source-to-source transformations may also be applicable. For instance, consider a single loop containing two REV requests to two different services. Furthermore, assume the two services are the same for each iteration of the loop. Loop splitting should be used in an attempt to create two REV requests and two loops, such that each REV request contains one loop.

7.4 REV Drawbacks

An REV mechanism poses three problems that must be addressed:

1. implementation overhead;
2. computer security; and
3. lack of improvement in performance and functionality.

Production-quality solutions exist for the first two problems, but the third problem depends on the computational environment and the distributed applications.

First, REV has compile-time and run-time costs. REV complicates binding and type checking. Implicit REV requests require use-definition analysis and procedure folding. Every service node must have an interpreter for the language in which the REV requests are written. The alternative, dynamically compiling and then executing a request, will often take longer than using RPC's. Requests could be precompiled, but the feasibility of precompilation depends on the homogeneity of the environment.

A second problem is computer security. REV and remote types exacerbate the security issues that must be considered in an RPC framework, such as service availability and protection. An RPC service programmer protects a node by controlling the routines that a client may invoke. An RPC service need not isolate concurrent requests from different clients; a single, shared address space will suffice. Because a trusted team of programmers implements the service routines, nonterminating RPC requests are not a problem.

On the contrary, REV service programmers must be suspicious of REV requests. An REV request

may run for a long time; in fact, there is no guarantee that it will terminate. To prevent program bugs or malicious clients from interfering with the requests of other clients, protection mechanisms developed for time-sharing systems will be necessary. Separate address spaces, authorization checks, resource accounting, and preemptive scheduling may be needed.

Finally, REV may not improve performance or provide any new functionality. Consider RPC's whose execution takes substantially longer than the REV overhead. Coalescing several such RPC's into a single REV request will not improve performance and may actually degrade performance because of the space and time overhead needed to support the advanced semantics of REV. If the RPC's form a functionally complete set of routines, REV yields no new functionality. The utility of REV depends on the communication channel, the exported routines, and the intended applications.

7.5 REV Advantages

REV is a simple mechanism that we expect will provide three benefits. Remote evaluation may:

1. increase performance by reducing processing time or network communication;
2. induce service designers to export additional routines that give application programmers new power; and
3. simplify the construction of distributed applications.

We discuss each of these benefits in turn.

7.5.1 Increased Performance

REV, which lets the application programmer partition a program for distributed execution in many ways, can realize a substantial reduction in communication requirements or time scales. An RPC programmer, in contrast, must accept the unique partitioning of a program implicitly imposed by service programmers.

REV may reduce the amount of communication between a client and a service, which can extend the apparent capability of clients with limited processing power or storage capacity. The reduction in communication may be substantial for database applications. A user extracts information from a remote database by submitting a request and then waiting for a response. A request that typically examines a sizable component of the database yet returns a comparatively small amount of information is called a *filter*. Consider an application that needs a filter that is not directly supported by the database. A programmer can use REV to construct a customized filter that executes at the database. If the size of the REV request is negligible, an REV implementation of a filter requires much less communication than the corresponding implementation without REV. If the communications network is slow or expensive, or if the client node has limited storage capacity, REV is the preferred mechanism for implementing filters. In these cases, an application programmer using REV can *relocate* processing to reduce communication. Unlike data compression schemes, REV does not require more processing to encode and decode the data that is transmitted.

Some filters select those components of a database that satisfy a predicate. If the routines exported by the database easily express the predicate, REV should be used. For instance, a user of a news service might be interested in all movie reviews by a famous reviewer. Other filters reduce the amount of information returned to the client by summarizing the information contained in each (selected) component. For instance, given the set of high priority articles at a news service, a user might be interested in the authors of those articles.

Another example of filtering concerns dictionary maintenance. Dictionary editors add new words to the dictionary and drop unused words as the language changes. Although reading newspapers and periodicals is a useful method for spotting new words, the designers of a news service probably never thought (seriously) about supporting dictionary maintenance. If the service designers had the foresight to support REV, the database of articles is immediately useful to dictionary editors. Although discovering novel uses for existing words is a difficult task to automate, discovering brand new words is not. Consider the brute force approach. Every month an REV request, which examines every article that appeared in the previous month, is sent to the news service. The request adds each word in each article to a set of words and returns the word set to the client. The client sorts the word set and then compares it with an on-line dictionary. The set of new words, along with close matches for the unavoidable spelling errors, is finally presented to the dictionary editors. An editor could see the articles in which each new word appeared by submitting the appropriate query to the news service. A client using only RPC's must fetch the entire collection of last month's articles and then compress the information into the word set. The REV approach filters the information at the news service instead of at the client.

It may be possible to avoid the brute force implementation in this example. A news service supporting keyword access to articles probably maintains a mapping from keywords to articles. Such a mapping would be extremely useful to the dictionary editors, even if commonly occurring "noise" words are not present. If the mapping is accessible to clients, an REV request that iterates over all pairs in the mapping and saves the word component of each pair generates the word set.

This example, which combined a newspaper database with a dictionary to assist dictionary editors, shows how REV facilitates the smooth integration of independent databases. Combining a dictionary with a thesaurus yields a service that is better than either alone. Associating a "yellow pages" database with an electronic street map is yet another example of how simple information sources may be combined to provide a new service. Joining independent databases for a new application typically requires that data be extracted or updated in a manner not expected by the service designers. While an application that uses several independent databases could be implemented with RPC's, their lack of generality will often hamper the application programmer and impose intolerable performance penalties on the resulting implementations.

Besides reducing the amount of communication between a client and a service, REV may improve performance by reducing the number of times control is passed between the two nodes. An

application without REV may repeatedly access a particular service because of its specialized hardware or data. If communications overhead dominates the execution time for most of the routines, combining the routines into a single REV request may substantially improve performance. REV amortizes the communications overhead over several former remote invocations.

One example is a service with an array processor. If an application performs four array operations in a row and needs only the final result, REV should be considered. Similarly, if an application performs four operations on the same array and needs all four results, REV could again be used. The same argument applies to four operations on four different arrays. In these examples, REV reduces the number of times control passes between the client and the service. In the first two examples, REV also reduces the total amount of data communicated across the network.

Another example in which an application repeatedly accesses a service involves a on-line dictionary. Assume the dictionary service exports **dictionary$lookup** as well as the built-in types. **Dictionary$lookup** takes a string and either returns the word's definition or raises an exception when no such word exists. One problem with conventional dictionaries is that a person looking up an unknown word is often forced to look up one or more words that appear in the definition of this word. REV efficiently supports a smart dictionary that performs expected lookups in advance. When a user asks for the definition of word **X**, an REV request could look up **X** as well as the (unusual) words in the definition of **X**. Another extension using REV transforms the dictionary service into a spelling corrector. In an RPC implementation, each attempt at correcting a misspelled word requires a separate remote invocation. Correcting a word with REV requires only a single remote invocation. If the dictionary service exports the **RemoteRecord** interface, the client can store the correction algorithm at the service for the duration of the binding between the client and the service. Transmitting a sizable correction algorithm only once may be useful during a session in which the user interactively corrects many words.

Our final example of an application that repeatedly accesses a remote node involves an airline flight reservation system. Each traveler has preferences regarding the itinerary, dates, departure and arrival times, stopovers, airlines, ticket class, and cost. Efficiently expressing all possible preferences and combinations of preferences with only RPC's is nearly impossible. On the other hand, REV permits a wide variety of preferences to be sent to the airline service. In more technical terms, arbitrary algorithms approximate preferences better than a fixed "universal" preference algorithm. A client using RPC's has two options: use the service's preference algorithm or extract the necessary information and evaluate all possibilities locally. Using REV to send the user's preference algorithm to the service generates trip plans more efficiently by eliminating unacceptable plans early. A more focused search may let the REV implementation find better plans in less time.

The dictionary and airline examples reflect search problems in which the next action depends on the presence or absence of data as well as the current state of the search. Unless the search strategy is built into the service, applications without REV are limited. Since REV makes code transmissible, the particular search strategy can always be sent to the service and applied to the database.

7.5.2 New Capabilities

Besides improving performance, REV may give the application programmer new capabilities. Although theoretically REV is no more powerful than RPC's, REV may induce service designers to support a wider variety of routines including those with procedures or closures as arguments. Iterators are one example.

REV reduces the effect of performance on the choice of service routines. A simple routine whose execution time is dwarfed by the RPC overhead is an unlikely candidate for a traditional remote procedure. An application programmer using REV, however, can often use such a routine efficiently. For example, a traditional mail service may not export routines that parse the header of a message in a mailbox. A client using RPC's that wants to retrieve certain messages based on their headers must retrieve all the messages. A client using REV, in contrast, can examine each message header at the service and retrieve only the desired messages.

7.5.3 Effect on Distributed Programming

REV simplifies the design and implementation of services, because service programmers do not have to provide all the software that executes at a service. Application programmers can construct customized facilities for the applications they write. Thus service software can be smaller, more stable, and easier to debug.

REV is also useful to application programmers. REV is more powerful than RPC's but introduces only minor syntactic extensions to a programming language. Like RPC implementations, an REV mechanism hides the bookkeeping and communication details from the application programmer. Location-independent REV requests assist a programmer tuning the performance of a distributed system. These requests are recognized by the compiler and do not change program semantics. This contributes to network transparency and facilitates reconfiguration. In short, REV is a simple but powerful technique for implementing and tuning a distributed system.

7.6 Key Ideas

This thesis has proposed and investigated a new primitive for constructing loosely-coupled distributed systems. Remote evaluation is the ability to send an expression to a remote node and evaluate it there. We assumed that the overhead for invoking a remote procedure was much greater than the overhead for invoking a local procedure and that a transaction mechanism spanned the distributed system. Our goal throughout the thesis has been to give the programmer fine-grained control over the location of processing and client data in a distributed application. A key constraint was that this relocation of processing and data should not affect program semantics for location-independent requests. The solution, REV with implicit procedures and remote types, is moderately

complex. Since simpler systems achieve many of the benefits REV provides, we describe the set of ideas behind our solution:

- *Transmissible Procedures:* Procedures with no free variables may be sent to a service and executed by the service as long as all routines accessible from the procedure are known at link time. Furthermore, every routine invoked by the procedure at the service must accompany the procedure or exist at the service executing the procedure. The programmer can use such a procedure as an argument to a remote procedure or as a new remote procedure. (Chapter 2)

- *Services:* We characterize a remote node by the set of interfaces it exports. This information lets the compiler verify the validity of a transmissible procedure and encode it. Because the application programmer can compose several service routines into a single REV request, services can export general routines in addition to routines tailored to a specific application. (Chapter 2)

- *REV Semantics:* Explicit REV requests let a programmer relocate the execution of a transmissible procedure from the client to a service. We decouple the transaction structure from the way a program is partitioned into components for local and remote execution. Each REV request must run as part of some transaction that is aborted if the request does not complete. REV requests, like ordinary procedure calls, use call by sharing. (Chapter 2)

- *Location Independence:* Service programmers declare the location independence of exported service routines. A routine is location-independent if its semantics does not depend on the node that executes it. A location-independent REV request imports only location-independent routines and preserves program semantics. (Chapter 2)

- *Call by Sharing in a Distributed System:* We implement call by sharing with call by value-overwrite, transactions, and compiler-enforced restrictions on service programmers. (Chapter 3)

- *Implicit REV Requests:* We let the programmer relocate the evaluation of a closure whose code is apparent at compile time. Procedure folding converts each implicit REV request to an explicit REV request at compile time. Implicit REV requests simplify the use of REV without impairing program readability. They directly support remote iterators and other routines with closures as arguments. Finally, implicit requests bring us one step closer to an automatic program partitioner. (Chapter 4)

- *Remote Data Types:* We developed a naming mechanism that lets one node refer to an object kept on another node. The naming mechanism is integrated into the type system via remote data types. Only the global name of a remote object, rather than its abstract value, is transmitted between nodes. Compiler-enforced rules for implementing remote types guarantee that call by value-overwrite implements call by sharing and that location-independent REV requests preserve program semantics. (Chapter 5)

- *RemoteRecords and RemoteArrays:* These system-defined remote types are identical to the **record** and **array** types except for transmissibility. They give the application programmer fine-grained control over the location of client data. (Chapter 5)

Although these ideas mesh nicely, designers of programming languages and distributed systems do

not have to incorporate all of them to realize substantial benefits. The appropriate ideas depend on the computational environment, the sophistication of the programmers, and the intended applications.

7.7 Areas for Further Research

Many questions concerning REV remain unanswered. Once a production system has been built and practical experience with REV is gained, alternatives we rejected may need to be reexamined. Some of the areas we feel deserve more attention are primarily theoretical:

- *Enhanced Compile-Time Analysis:* Use flow analysis techniques to determine which mutable arguments to REV requests are read-only and which global names accompanying a request denote objects at the service executing the request.

- *Transmission of Clusters:* Let clusters, which are modules that implement abstract data types, be transmitted between nodes. Investigate the implications of sending an instance of a type to a service that does not export the type.

- *Multiple Languages:* Let applications written in different languages use REV to interact with the same service. Specify the semantics of inter-language communication and investigate procedure transmission. Design a method for specifying the control constructs that a service "exports."

- *Automatic Program Partitioning:* Formulate a cost model of distributed computation and let the compiler relocate processing with location-independent REV requests.

The remaining extensions to REV are practical:

- *Practical Experience:* Design, build, and tune a production REV system. Implement real applications and evaluate the usefulness of various REV features, such as implicit requests and remote data types.

- *Optimizations:* Let the service cache the code portion of repeated REV requests. Investigate the utility of suggesting that a service do background compilation or dynamic compilation of particular requests. Let the programmer or compiler make these suggestions and measure the performance changes.

- *Precompilation:* Evaluate the level of service protection when requests are precompiled by the client. Measure the changes in code size, communication time, and performance. Evaluate the utility of a trusted compilation service that attaches a digital signature to the code it compiles.

7.8 An Evaluation

In this chapter we used many realistic examples to illustrate the advantages of REV. We usually outlined an RPC implementation of a system and then described an application for which the procedures exported by the service were not useful. The REV-based solution we proposed naturally solved the problem. A staunch believer in remote procedures might declare that a similar solution using only RPC's is possible. While conceding this point, we would continue the debate by describing another application for which the extended RPC interface was not useful. If the RPC defender augmented the extended interface, we would again propose another problematic application. This endless debate reflects a key motivation for REV. Because of unexpected applications, an "optimal" RPC interface probably does not exist for many kinds of services. Moreover, the user community is likely to be much larger than the service implementation team. Therefore, for most new applications, users should take the initial responsibility for providing useful service routines.

Conventional programming languages offer a useful analogy. Early languages had a fixed set of built-in types but gave the programmer little or no opportunity to construct new types. The inability to define new types was eventually recognized as a serious shortcoming. Language designers solved this problem by including simple but powerful mechanisms that let a programmer define new types that have equal standing with the built-in types.[15] In a distributed computing environment, an RPC service exports a fixed interface of routines. On the other hand, REV lets the application programmer compose service routines to create new routines that have equal standing with the exported routines.

One goal of this thesis has been to convince others that remote evaluation, which is by now fairly well understood, is both feasible and desirable in a number of situations. REV can improve performance and provide generality for distributed applications. Besides being easy to use, REV has powerful semantics and an efficient implementation. REV supports reconfiguration because location-independent requests have no effect on program semantics. REV increases network transparency, because both local invocations and relocated invocations use call by sharing. Finally, REV is more powerful than RPC's: an RPC is a simple REV request that invokes only a single service routine.

Adding REV to a system that supports RPC's is not a difficult task, because the bulk of an REV mechanism is an RPC mechanism. Earlier in the chapter we listed the set of ideas behind implicit REV requests and remote types. We hope that these ideas, either alone or in combination, will make it easier to design and implement distributed computer systems that need both generality and good performance.

---

[15] Most languages, however, do not permit a programmer to redefine special type constructors such as **record**.

Bibliography

[1] Allen, F. E., and Cocke, J.
A Program Data Flow Analysis Procedure.
Communications of the ACM 19(3):137-147, March, 1976.

[2] Atkinson, M. P.
IDL: A Machine-Independent Data Language.
Software Practice and Experience 7(6):671-684, November-December, 1977.

[3] Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M. D.
Grapevine: An Exercise in Distributed Computing.
Communications of the ACM 25(4):260-274, April, 1982.

[4] Birrell, A. D., and Nelson, B. J.
Implementing Remote Procedure Calls.
ACM Transactions on Computer Systems 2(1):39-59, February, 1984.

[5] Birtwistle, G. M., Dahl, O.-J., Myhrhaug, B., and Nygaard, K.
Simula Begin.
Auerbach Publishers Inc., Philadelphia, 1973.

[6] Burton, F. W.
Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs.
ACM Transactions on Programming Languages and Systems 6(2):159-174, April, 1984.

[7] Daniels, D., Selinger, P., Haas, L., Lindsay, B., Mohan, C., Walker, A., and Wilms, P.
An Introduction to Distributed Query Compilation in R*.
In *Proceedings of the Second International Symposium on Distributed Databases*. September, 1982.
Available as IBM Research Report RJ3497.

[8] Donahue, J. E.
Complementary Definitions of Programming Language Semantics.
In Goos and Hartmanis (editors), *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1976.

[9] Eastlake, D. E., III.
Tertiary Memory Access and Performance in the Datacomputer.
In *Proceedings of the Third International Conference on Very Large Data Bases*, pages 259-267. October, 1977.

[10] El-Dessouki, O.
Program Partitioning and Load Balancing in Network Computers.
PhD thesis, Illinois Institute of Technology, December, 1978.

[11] El-Dessouki, O., Huen, W., and Evens, M.
Towards a Partitioning Compiler for a Distributed Computing System.
In *Proceedings of the First International Conference on Distributed Computing Systems*, pages 296-304. October, 1979.

[12] Farrell, J.
The Datacomputer -- a Network Data Utility.
In *{First} Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 352-364. May, 1976.

[13] Gaines, R. S.
An Operating System Based on the Concept of a Supervisory Computer.
Communications of the ACM 15(3):150-156, March, 1972.

[14] Gifford, D. K.
Information Storage in a Decentralized Computer System.
PhD thesis, Stanford, June, 1981.
Also available as Xerox PARC Technical Report CSL-81-8.

[15] Good, M. D., Whiteside, J. A., Wixon, D. R., and Jones, S. J.
Building a User-Derived Interface.
Communications of the ACM 27(10):1032-1043, October, 1984.

[16] Gordon, M. J. C.
The Denotational Description of Programming Languages: An Introduction.
Springer-Verlag, New York, 1979.

[17] Gray, J. N.
Notes on Data Base Operating Systems.
In Goos and Hartmanis (editors), *Lecture Notes in Computer Science*, pages 393-481. Springer-Verlag, Berlin, 1978.

[18] Herlihy, M.
Transmitting Abstract Values in Messages.
Master's thesis, Massachusetts Institute of Technology, April, 1980.
Also available as MIT Technical Report MIT/LCS/TR-234.

[19] Herlihy, M. and Liskov, B.
A Value Transmission Method for Abstract Data Types.
ACM Transactions on Programming Languages and Systems 4(4):527-551, October, 1982.

[20] Jones, A. K., and Liskov, B. H.
A Language Extension for Expressing Constraints on Data Access.
Communications of the ACM 21(5):358-367, May, 1978.

[21] Lampson, B. W. and Sturgis, H. E.
Crash Recovery in a Distributed Data Storage System.
Xerox PARC, Palo Alto, CA, April, 1979.

[22] Leverett, B. W., Cattell, R. G. G., Hobbs, S. O., Newcomer, J. M., Reiner, A. H., Schatz, B. R., and Wulf, W. A.
An Overview of the Production Quality Compiler-Compiler Project.
Technical Report CMU-CS-79-105, CMU Department of Computer Science, February, 1979.

[23] Lindsay, B. G., Haas, L. M., Mohan, C., Wilms, P. F., and Yost, R. A.
Computation and Communication in R*: A Distributed Database Manager.
ACM Transactions on Computer Systems 2(1):24-38, February, 1984.

[24] Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, J. C., Scheifler, R. and Snyder, A.
CLU Reference Manual.
In Goos and Hartmanis (editors), *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1981.

[25] Liskov, B. and Scheifler, R.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
ACM Transactions on Programming Languages and Systems 5(3):381-404, July, 1983.

[26] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C.
Abstraction Mechanisms in CLU.
Communications of the ACM 20(8):564-576, August, 1977.

[27] Mamrak, S. A., Leinbaugh, D. and Berk, T. S.
A Progress Report on the Desperanto Research Project -- Software Support for Distributed Processing.
Operating Systems Review 17(1):17-29, January, 1983.

[28] Mamrak, S. A., Maurath, P. Gomez, J., Janardan, S. and Nicholas, C.
Guest Layering Distributed Processing Support on Local Operating Systems.
In *Proceedings of the Third International Conference on Distributed Computing Systems*, pages 854-859. October, 1982.

[29] Marill, T. and Stern, D.
The Datacomputer -- A Network Data Utility.
In *Conference Proceedings, 1975 NCC*, pages 389-395. AFIPS, 1975.

[30] Marshall, H. Z.
The Linear Graph Package, A Compiler Building Environment.
SIGPLAN Notices 17(6):294-300, June, 1982.
(Proceedings of the SIGPLAN '82 Symposium on Compiler Construction).

[31] Minsky, N.
Files with Semantics.
In J. B. Rothnie (editor), *Proceedings of the International Conference on Management of Data*, pages 65-73. ACM-SIGMOD, 1976.

[32] Mitchell, J. G., Maybury, W., and Sweet, R.
Mesa Language Manual.
Technical Report CSL-79-3, Xerox PARC, April, 1979.

[33] Morris, J. H.
Towards More Flexible Type Systems.
In Goos and Hartmanis (editors), *Lecture Notes in Computer Science*, pages 377-384. Springer-Verlag, New York, 1974.

[34] Nelson, B. J.
Remote Procedure Call.
PhD thesis, Carnegie-Mellon University, May, 1981.
Also available as Xerox PARC Technical Report CSL-81-9.

[35] Schneiderman, B. and Shapiro, S. C.
Toward a Theory of Encoded Data Structures and Data Translation.
International Journal of Computer and Information Sciences 5(1):33-43, March, 1976.

[36] Shapiro, M.
The Design of the GIROLLE Remote-Procedure Call Protocol.
CMIRH, Paris, November, 1983.

[37] Strom, R. E. and Yemini, S.
NIL: An Integrated Language and System for Distributed Programming.
SIGPLAN Notices 18(6):73-82, June, 1983.
(Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems).

[38] Stroustrup, B.
On Unifying Module Interfaces.
Operating Systems Review 12(1):90-98, January, 1978.

[39] Stroustrup, B.
An Inter-Module Communication System for a Distributed Computer Sustem.
In *Proceedings of the First International Conference on Distributed Computing Systems*, pages 412-418. October, 1979.

[40] Swinehart, D., McDaniel, G., and Boggs, D.
WFS: A Simple Shared File System for a Distributed Environment.
In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 9-17. December, 1979.

[41] Tanenbaum, A. S.
Computer Networks.
Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[42] Wallis, P. J. L.
External Representation of Objects of User-Defined Type.
ACM Transactions on Programming Languages and Systems 2(2):137-152, April, 1980.

[43] Weihl, W. E.
Specification and Implementation of Atomic Data Types.
PhD thesis, Massachusetts Institute of Technology, March, 1984.
Also available as MIT Technical Report MIT/LCS/TR-314.

[44] Weihl, W. and Liskov, B.
Implementation of Resilient, Atomic Data Types.
ACM Transactions on Programming Languages and Systems 7(2):244-269, April, 1985.

[45] Weinreb, D. and Moon, D.
Lisp Machine Manual.
MIT Artificial Intelligence Laboratory.

[46] White, J. E.
A High-Level Framework for Network-Based Resource Sharing.
In *Conference Proceedings, 1976 NCC*, pages 561-570. AFIPS, 1976.

[47] White, J. E.
Elements of a Distributed Programming System.
Computer Languages 2(4):117-134, 1977.

[48] *Courier: The Remote Procedure Call Protocol*
Xerox Corporation, Stamford, Connecticut, 1981.
Available as Xerox System Integration Standard XSIS-038112.

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: MIT/LCS/TR-354	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): Remote Evaluation	5. TYPE OF REPORT & PERIOD COVERED: Ph.D. Dissertation May 1982-January 1986
	6. PERFORMING ORG. REPORT NUMBER: MIT/LCS/TR-354
7. AUTHOR(s): James W. Stamos	8. CONTRACT OR GRANT NUMBER(s): DARPA/DOD N00014-83-K-0125
9. PERFORMING ORGANIZATION NAME AND ADDRESS: MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: DARPA/DOD, 1400 Wilson Boulevard, Arlington, VA 22209	12. REPORT DATE: January 1986
	13. NUMBER OF PAGES: 136
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): ONR/Department of the Navy, Information Systems Program, Arlington, VA 22217	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

atomic transactions, call by sharing, call by value-overwrite, computer networks, distributed computing, evaluation, global names, message passing, procedure call, programming languages, remote evaluation, remote procedure call, remote types

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A new technique for computer to computer communication is presented that can increase the generality and performance of distributed systems. This technique, called Remote Evaluation, lets one computer send another a request in the form of a program. A computer that receives such a request executes the program in the request and returns the results to the sending computer.

Remote evaluation provides a new degree of flexibility in the

design of distributed systems. In present distributed systems that use Remote Procedure Calls, server computers are designed to offer a fixed set of services. In a system that uses remote evaluation, server computers are more properly viewed as programmable soft abstractions. One consequence of this flexibility is that remote evaluation can reduce the amount of communication that is required to accomplish a given task.

Our thesis is that it is possible to design a remote evaluation system that permits the processing of a program to be distributed among remote computers without changing the program's semantics. In support of this thesis our proposal for remote evaluation uses the same argument passing semantics for local and remote procedure invocations (call by sharing); it provides atomic transactions to mask computer and communication failures; and it provides a static checking framework that identifies procedures that can not be relocated from computer to computer.

We discuss both the semantics of remote evaluation and our experience with a prototype implementation. The idea of a remote data type is introduced to let one computer name objects at a remote computer. A detailed discussion of the compile-time and run-time support necessary for remote evaluation is provided, along with a detailed sample application.