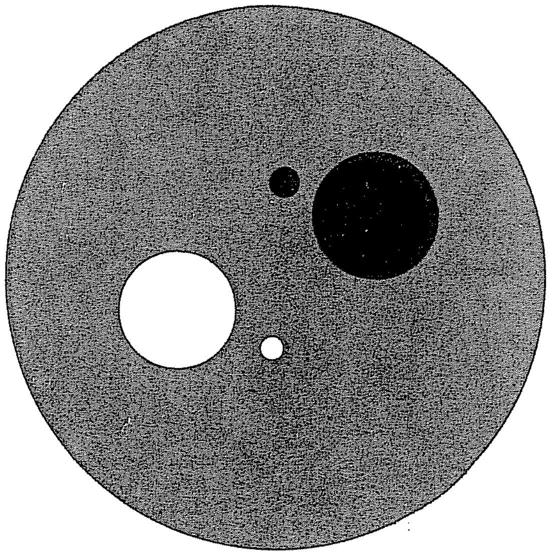

THE DESIGN AND PERFORMANCE OF HIGH-LEVEL LANGUAGE PRIMITIVES FOR DISTRIBUTED PROGRAMMING

by

Thomas John LeBlanc

Computer Sciences Technical Report #492

December 1982

THE DESIGN AND PERFORMANCE OF HIGH-LEVEL LANGUAGE PRIMITIVES FOR DISTRIBUTED PROGRAMMING

by

THOMAS JOHN LEBLANC

A thesis submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

1982

ABSTRACT

We consider a distributed computing environment in which a high-level distributed programming language kernel, as contrasted with a distributed operating system, is sufficient support for programming applications and in which performance is of primary concern. We propose programming language support for such an environment and present the performance results of an implementation.

Using the distributed programming language StarMod as a basis, we describe communication primitives which provide interprocess communication, broadcast communication, remote invocation, and remote memory references. Each form of communication is integrated into StarMod in a consistent fashion maintaining the properties of transparency, full functionality, and modularity. The costs and benefits associated with the various models of communication are analyzed based on the results of an implementation that runs on 8 PDP 11/23 microprocessors connected by a one megabit/second network.

We conclude with a comparison of the communication primitives, including design, performance, ease of implementation, functionality, and tractability. We discuss general lessons learned in constructing distributed programming language kernels for "bare" machines and suggestions for the organization of an architecture that supports the efficient implementation of high-level language communication primitives.

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, Bob Cook. Technically, Bob is my thesis advisor; in actuality, he is a friend and colleague without whom I could not have written this dissertation. He was an ideal advisor: supportive during crises of confidence, critical whenever required by circumstances, and always willing to lose a tennis match when I needed a win.

I would also like to thank the other members of my committee, David DeWitt, Charles Fischer, James Goodman, and Charles Kime. In particular, I would like to thank Dave DeWitt who, using cardboard, scotch tape and bubble gum, kept the PDP 11/23's running long enough for me to perform my experiments, and Charlie Fischer who has been a fountain of knowledge during my stay at Wisconsin; he is the reason why, to this day, I prefer compilers to operating systems.

Special thanks are due Bob Gerber who developed the Modula kernel and was instrumental in the early development of the StarMod kernel. His efforts saved me months of tedious programming; his congeniality saved my sanity on more than one occasion.

Further support along the way was also provided by the following people: Julius Archibald, Carol Smith, the "secretaries", the "boys in the lab", and, of course, my parents. Their contributions to my education have not gone unnoticed.

Finally, and most importantly, I want to thank my wife, Anne, for those lonely nights she spent at home while I was in the lab. Put a light in the window Slim, I'm finally coming home.

TABLE OF CONTENTS

Chapter 1 - Introduction 1

1.1 Introduction and Motivation 1
1.2 Scope and Goals 3
1.3 Methodology 4
1.4 Implementation Environment 4
1.5 Dissertation Plan 5

Chapter 2 - Language Primitives for Distributed Programming 7

2.1 Introduction 7
2.2 Concurrent Programming Languages 7
2.3 Distributed Programming Languages 8
2.3.1 Distributed Processes 8
2.3.2 Communicating Sequential Processes 9
2.3.3 Liskov's Primitives 9
2.3.4 Ada Tasks 10
2.3.5 Synchronizing Resources 10
2.4 Nelson's Remote Procedure Call 11
2.5 Spector's Remote Operations 11
2.6 Summary 13

Chapter 3 - An Overview of StarMod 14

3.1 Introduction 14
3.2 Processes 14
3.3 Signals 15
3.4 Modules 15
3.4.1 Interface Modules 16
3.4.2 Processor Modules 16
3.4.3 Network Modules 17
3.5 Ports 18
3.6 The Completion Statement 18
3.7 Summary 19

Chapter 4 - Message- and Procedure-Oriented Communication 21

4.1 Introduction 21
4.2 Message-Oriented Communication 22
4.2.1 Port Declarations 23
4.2.2 Region Statement 25
4.2.3 Port Call Completion Semantics 26
4.2.4 Implementation 29
4.2.4.1 Implementation Description 29
4.2.4.2 Implementation Performance 30
4.2.4.3 Comparisons with Charlotte 35

4.2.4.4 Improving Performance 36
4.3 Remote Procedure Call 38
4.3.1 Essential Properties of an RPC Mechanism 39
4.3.1.1 Uniform Call Semantics 39
4.3.1.2 Type Checking 40
4.3.1.3 Full Parameter Functionality 40
4.3.1.4 Concurrency Control and Exception Handling 40
4.3.1.5 Distributed Binding 40
4.3.2 Orphan Computations 41
4.3.3 Implementation 43
4.3.3.1 Implementation Description 43
4.3.3.2 Implementation Performance 44
4.3.3.3 Comparisons with Nelson's RPC 44
4.3.3.4 Improving Performance 46
4.3.4 Remote Process Activation 47
4.4 Broadcast Messages 48
4.4.1 Broadcast Port Declarations 49
4.4.2 Broadcast Semantics 50
4.4.2.1 Broadcast Replies 50
4.4.2.2 Broadcast Completion Semantics 51
4.4.2.3 Multicast Messages 51
4.4.3 Implementation 52
4.4.3.1 Implementation Description 52
4.4.3.2 Implementation Performance 53
4.5 Summary 59

Chapter 5 - Remote Memory References 61

5.1 Introduction 61
5.2 The Data Abstraction Conflict 62
5.3 Synchronization Issues 63
5.4 Remote Variable References 65
5.4.1 Transparency 65
5.4.1.1 Transmitting Complex Types 66
5.4.1.2 Qualified References 66
5.4.1.3 Pointer Types 69
5.4.1.4 Type-Based Synchrony 69
5.4.1.5 Failure and Retry 70
5.4.2 Synchronization 71
5.4.3 Address Binding 73
5.4.4 Implementation 74
5.4.4.1 Implementation Description 74
5.4.4.2 Implementation Performance 75
5.4.4.3 Comparisons with Spector's Remote References 77
5.4.4.4 Effects of Qualification on Performance 79
5.4.4.5 Effects of Synchronization on Performance 80
5.4.4.6 Improving Performance 80
5.4.5 Protection and Autonomy 83
5.5 Summary 84

Chapter 6 - A Family of Communication Primitives 86

6.1 Introduction 86
6.2 Comparing Models of Communication 86
  6.2.1 Performance 86
    6.2.1.1 Network Contention 87
    6.2.1.2 Interaction Between Communication Primitives 88
    6.2.1.3 Local Vs. Remote Communication 89
    6.2.1.4 Overhead of Marshalling Parameters 90
  6.2.2 Ease of Implementation 90
  6.2.3 Functionality 91
  6.2.4 Familiarity of the User Interface 92
  6.2.5 Formal Tractability 92
6.3 Lessons in Kernel Construction 93
  6.3.1 Balance Simplicity and Performance 94
  6.3.2 Simple Protocols Are Not Simple 94
  6.3.3 Don't Increase Traffic to Improve Performance 95
  6.3.4 Use Simple Packet Structures 96
  6.3.5 Avoid Expensive Instructions 96
  6.3.6 Avoid Mutual Exclusion and Synchronization 97
  6.3.7 Use a High Resolution Line Clock 98
6.4 An Architecture for a Family of Primitives 98
  6.4.1 Processor 98
  6.4.2 Communication Adapter 99
  6.4.3 Memory 101
  6.4.4 The Network 102
    6.4.4.1 Network Topology 102
    6.4.4.2 Network Bandwidth 103
    6.4.4.3 Processor Homogeneity 103
    6.4.4.4 Language Homogeneity 105
6.5 Summary 106

Chapter 7 - Conclusion 107

7.1 Summary 107
7.2 Future Research 108
  7.2.1 Delayed Bindings 108
  7.2.2 High-Performance, Processor-Synchronous Operations 108
  7.2.3 Network Contention 108
  7.2.4 Network Architectures 109
  7.2.5 Dynamic Mapping 109
  7.2.6 User Experience 109

Appendix 111

Bibliography 112

LIST OF FIGURES

Figure 4-1	Execution Time (in ms.) for Asynchronous and Synchronous Port Calls	30
Figure 4-2	Timing Program for Local Port Calls	31
Figure 4-3	Timing Program for Remote Port Calls	32
Figure 4-4	% Network Bandwidth Utilization for Remote Port Calls	34
Figure 4-5	RPC Performance and Bandwidth Utilization	44
Figure 4-6	RPC Performance and Bandwidth Utilization Without Argument Copy	47
Figure 4-7	Execution Time (in ms.) for Asynchronous Broadcast Port Call To Multiple Destinations	54
Figure 4-8	Speedup Factor of Asynchronous Broadcast Communication vs. Point-to-Point Communication	55
Figure 4-9	TBR(N) Projected Time (in ms.) and Speedup Factor vs. Point-to-Point Communication for Asynchronous Broadcast Port Call To Multiple Destinations	56
Figure 4-10	Execution Time (in ms.) for Synchronous Broadcast Port Call To Multiple Destinations	58
Figure 4-11	Speedup Factor of Synchronous Broadcast Communication vs. Point-to-Point Communication	59
Figure 5-1	Remote Variable Reference Performance and Network Bandwidth Utilization	76
Figure 5-2	Instruction Breakdown for Remote Variable Reference	78
Figure 5-3	Remote Variable Reference Performance Without Copy	81
Figure 5-4	Projected Timing Results (in Microseconds) for Higher Bandwidth Networks	83
Figure 6-1	Ratio Between Remote and Local Execution Time for Various Communication Models with 2 Byte Arguments	89
Figure 6-2	Projected Execution Time (in ms.) for Higher Bandwidth Networks and Higher Performance Processors	104

Chapter 1

INTRODUCTION

1.1. Introduction and Motivation

This dissertation considers programming language support for distributed programming, with particular emphasis on execution efficiency. The approach is to provide program structuring forms and a spectrum of communication primitives within the context of a high-level, distributed programming language implementation.

Numerous factors have influenced the evolution of distributed systems including site autonomy, decentralization, reliability, performance, availability, resource sharing, growth potential, and geographical distribution [34]. Depending on the relative importance of each factor, distribution of processors might span a few inches (e.g., for performance or reliability) or thousands of miles (e.g., for geographical distribution).

In the past, primitives for communication in distributed systems have frequently been developed in an environment that precludes both a variety of communication models and high performance. For example, Liskov's emphasis on robustness lead to the selection of remote procedure call as the primitive of choice [41], regarding other communication primitives as inadequate for constructing robust software. Distributed operating systems frequently use message-based communication because the process structure provided by the typical operating system does not allow code or data sharing. In situations such as these, the optimized performance of a particular implementation of the communication model may be an important consideration. However, the performance of the entire class of primitives, as compared with some other communication model in the same environment, is seldom an issue. In this dissertation we take the view that multiple forms of communication are necessary to satisfy both the program structuring and performance requirements of distributed programs.

Our work was motivated by several considerations. We chose to investigate distributed programming issues from a programming language perspective because we believe that a distributed programming language kernel is sufficient support for a large class of programs, particularly programs distributed to enhance performance. Since a language-based distributed programming system can be developed more quickly and transported more readily than most distributed operating systems, development and maintenance costs are greatly reduced. In addition, a programming language kernel can optimize performance for that class of programs the language is designed to address. More general implementations, layered protocols for example, tend to introduce significant overhead during execution [10]. The static bindings and compile-time type checking inherent in many high-level programming languages provide run-time protection without run-time overhead. More important, a programming language provides a consistent model of program organization and communication, a structured view frequently unobtainable in a language that implements all communication using explicit subroutine calls to the operating system.

Further motivation was provided by the work of Spector [49] and Nelson [44]. An important aspect of Spector's work was to show that one class of remote references, remote memory references, can be executed efficiently on a local-area network. Nelson focused on the desirability of remote procedure call (RPC) as a communication primitive for constructing programs for distributed systems. An experimental implementation demonstrated that RPC can also be executed efficiently in the context of a specialized implementation. Both Spector and Nelson acknowledge that programming language support is a necessary attribute for the practical application of their somewhat specialized results.

We do not concentrate on a particular model of communication because we believe that a communication system based on a single model of communication will execute some large class of programs inefficiently. This impression is reinforced by the results of Spector and Nelson. Two extremes in the spectrum of communication primitives, remote memory reference and remote procedure call, were shown to be desirable forms of communication that can be executed efficiently. We consider our work to be an extension, and perhaps a unification, of Spector's and Nelson's work.

1.2. Scope and Goals

The theme of this dissertation is that a high-level, distributed programming language implementation that provides a rich set of communication primitives is a reasonable and efficient mechanism for programming high-performance, local-area networks. We characterize the computer networks of interest as single-user, single-address space processors executing programs with modest I/O and security requirements that are distributed primarily to enhance performance. Such attributes are commonly associated with microprocessor-based, local-area networks. While we do not strictly limit discussion to single-user, single-address space processors, we are primarily concerned with microprocessor-based, local-area networks.

The comparative performance of various communication models is a central issue in our research. Therefore we examine an environment in which performance is not limited by current communications technology. Most long-haul networks are limited by the bandwidth of the telephone network; saving a few milliseconds in processing time will not significantly improve performance of overnight network mail services. Thus, we specifically exclude consideration of applications such as inter-network mail servers.

We propose that the programming language provide the user with a uniform and complete view of the system. No additional explicit operating system support is envisioned. The resultant virtual machine is not intended to completely address I/O requirements, protection, and other issues to the extent typically found in operating systems.

Our goals are two-fold. The first is to demonstrate the advantages of a programming system in which a variety of communication primitives are supported. Our second goal is to show that communication primitives subsumed in a high-level language implementation can be implemented very efficiently, especially in comparison with systems that present a more general user interface, for example, distributed operating systems and layered network protocols.

1.3. Methodology

Using the programming language StarMod [12] as our context, we explore various models of communication between remote processors. StarMod provides a uniform program structure and philosophy consistent with the construction of distributed programs. Each communication primitive is considered in three steps: (1) a language design within the StarMod context, including syntax and semantics, (2) a run-time kernel implementation, and (3) performance analysis and fine-tuning of the implementation for efficient execution.

1.4. Implementation Environment

Throughout this dissertation, performance results derived from an implementation are presented. To evaluate those results and to place them in proper perspective, it is essential that the implementation environment be well understood.

The implementation executes on a network of 8 Digital Equipment Corporation PDP 11/23 microprocessors connected by a Computrol megalink, a one megabit/second, carrier-sense (no collision detection), broadcast network. The PDP 11/23 processor executes about 170,000 instructions per second. A typical instruction requires 6 microseconds. The microinstruction cycle time is 300 nanoseconds and the interrupt service time is 8.2 microseconds [15]. An appendix lists the time required to execute some common machine language instructions and certain frequently executed language primitives found in the run-time kernel (e.g., procedure call, process switch).

Each PDP 11/23 processor contains a hardware clock with 1/60 second resolution used to measure timeout intervals and performance results. Clock interrupts maintain the system clock and also permit the scheduling algorithm to execute. The clock interrupt handler is always present and the associated overhead is included in all timing results.

Test results are, for the most part, best case performances on an unloaded network. At times, artificial loads were generated on the network for performance evaluation in the presence of contention. In any event, there were no other users on the network when the test results were generated.

The implementation was developed on a "bare" machine. The entire communications kernel resides within the language kernel. No other system software is used, thus all software overhead is our own.

1.5. Dissertation Plan

The organization of this dissertation is as follows. Chapter 2, a summary of related work, contains a survey of programming language proposals for distributed systems. Chapter 3 presents an overview of the programming language StarMod, a distributed programming language which serves as a starting point for the remainder of the dissertation. In Chapter 4, basic communication using messages and procedures is introduced within the context of a high-level language; syntax, semantics, and performance are described. We also extend the notion of point-to-point communication to permit broadcast communication, whereby a single processing node may communicate with an entire network partition using a single invocation, and compare the performance attained with the expected performance. In Chapter 5, a more primitive operation, remote memory reference, is considered. We examine the advantages and disadvantages of such a primitive operation, propose a language design incorporating transparent remote variable references, and present results describing the observed performance. In Chapter 6, we use qualitative and quantitative criteria to compare the various communication primitives. We also discuss lessons learned during construction of the kernel and consider architectural support for the development of high-level language kernels for distributed programming. Chapter 7 concludes the dissertation with a summary and suggestions for future research.

Chapter 2

LANGUAGE PRIMITIVES FOR DISTRIBUTED PROGRAMMING

2.1. Introduction

Linguistic support for parallel algorithms can be divided into two classes, concurrent and distributed programming languages. Concurrent programs require multiple processes to execute; distributed programs require multiple processors for their execution. Languages designed for concurrent programming often assume that the programs execute on a single processor. Such languages may allow shared variables to be used for process communication and may not provide a mechanism for message communication between processes. This is in contrast to languages for distributed programming that need to address issues of process communication without shared memory, communication protocols, remote references and operations, and network reliability. The difference between the two classes is a subjective one since, conceptually, it is possible to write distributed programs using a concurrent programming language. Considerable effort must be expended, however, on developing underlying software designed to map concurrent programs onto distributed systems.

In this chapter, we review some of the important language proposals designed to support concurrent and distributed programs, as well as other relevant work that influenced our language development.

2.2. Concurrent Programming Languages

Both Concurrent Pascal [7] and Modula [56] are Pascal-based languages designed for concurrent programming. Processes and monitors (interface modules) are used to provide concurrency and synchronization. Neither language was designed to address the particular issues associated with distributed programming.

For example, Wirth's implementation of Modula, designed for a uniprocessor, disables the hardware interrupt mechanism to implement mutual exclusion [55]. This technique does not extend to multiprocessor systems. In addition, both languages assume the existence of shared memory and neither provides explicit constructs for message-based communication. The primary advantage of these languages is that they have been implemented and, therefore, provide an environment for experimentation with concurrent programming. Additionally, the kernels associated with their run-time packages are small and simple to write since few run-time operations are supported.

2.3. Distributed Programming Languages

2.3.1. Distributed Processes

Distributed Processes (DP) [8] was proposed by Brinch Hansen as a language concept for distributed programming without shared variables. As such, it was a first attempt to address the problem of language design for distributed programming. The concept was intended for real-time applications executing on microcomputer networks. Remote procedure calls that may return results are the fundamental operation used for communication; procedure argument lists are used to transmit information between processors. Synchronization of processes is achieved using guarded regions [16].

DP was designed for the specific purpose of programming real-time applications on microcomputer networks and is somewhat inflexible as a general-purpose language for distributed programming. Limitations include: (1) a program composed of distributed processes consists of a fixed number of concurrent processes, (2) each processor is dedicated to a single process, (3) there are no common variables between processes, a direct result of the requirement that each processor be dedicated to a single process, and (4) remote procedure calls, the only form of inter-machine communication available, are considered indivisible operations; the calling process must wait until the remote operation is finished.

In spite of these limitations, DP is well-suited to its application environment, microprocessor networks designed for real-time applications. In addition, the design of DP had considerable influence on the development of StarMod, upon which much of the work reported in this dissertation is based.

2.3.2. Communicating Sequential Processes

Communicating Sequential Processes (CSP) [25] assumes input and output as the fundamental programming operations and composition of parallel processes as the basic structuring method of programs. This language proposal uses Dijkstra's guarded commands [16] to provide nondeterminism, a **parbegin** command to introduce concurrency, and input/output commands for process communication. In order for two processes to communicate, each must name the other as the object of an input/output command. Each process must wait for the other to execute its input/output command.

The primary disadvantages of CSP are that communication is symmetric (each process must name the other to communicate), static binding is used to name communicating processes, and there is no facility for asynchronous communication. Nevertheless, the numerous program examples in [25] demonstrate the flexibility of the proposal.

2.3.3. Liskov's Primitives

Liskov has suggested primitives for distributed programming that support modularity, communication, and robust behavior [38,39,40,41]. A **guardian** is an abstraction that represents a logical processor; guardians are dynamic in nature in that the population of guardians may increase or decrease during program execution. Processes communicate by sending messages to **ports**, one-directional gateways into guardians [38]. The type of communication supported has ranged from the simple no-wait send [38] to the more complex remote procedure call [41].

The primary thrust of Liskov's work is towards robust software for distributed systems. The approach is very high level; Clu abstraction techniques combine with atomic transactions, which are most frequently associated with distributed databases, to form a powerful, high-level tool for software construction.

2.3.4. Ada Tasks

Ada tasks [52] support concurrency using the procedure call model. An **entry** procedure within a task differs from a normal procedure in that the entry procedure is executed by the enclosing task, not the calling task, which is suspended during execution of the entry procedure. After execution of the entry procedure, both tasks continue in parallel. This *rendezvous* facility was designed to provide both process communication and synchronization. The calling task calls an entry procedure, actual parameters are bound to formal parameters, and the calling task waits for the called task to execute an **accept** statement. After the rendezvous, the called task executes the entry procedure.

In Ada, the calling process must name the rendezvous point within the receiving process. The naming operation is implicit in that a specific entry point for rendezvous must be given and each entry point is associated with only one task. Since task qualification may be used to specify the entry point, including an access type specification (pointer to task), communication paths between individual instantiations of task types may not always be determined statically.

2.3.5. Synchronizing Resources

Andrews's view of distributed systems as a set of resources providing service to clients led to the development of a communication and synchronization proposal called Synchronizing Resources (SR) [1]. A resource is a set of processes, together with local data, that define some operation; processes are synchronized by boolean expressions in guarded commands. Process communication may take the form of remote invocation (call), blocking the caller until the operation has completed execution, or message transmission (send), blocking the sending process only long enough to buffer the arguments. An **in** statement, analogous to the accept statement of Ada, is used to schedule execution of the operation.

2.4. Nelson's Remote Procedure Call

Nelson's thesis on remote procedure call (RPC) [44] addresses the desirability of the remote procedure call as a model of process communication and control, the properties such calls should exhibit, and the efficient implementation of remote calls. Ideal properties for an RPC mechanism are said to include uniform call semantics, binding and configuration, type checking, complete parameter functionality, and concurrency control and exception handling. (It is worth noting that any communication primitive incorporated into a high-level programming language should, to a great extent, realize these ideal properties. These properties are reconsidered in detail in later chapters with regards to the specific primitives we discuss.)

Nelson's thesis presents the design of a transparent RPC mechanism in considerable detail and then evaluates the performance of a family of mechanisms based on the design. Special attention is paid to maintaining semantics in the presence of processor or communication failures, via orphan algorithms, and to parameter transmissions involving complex types, two of the most difficult issues associated with RPC. Performance measurements are presented that show that even a complex primitive such as RPC can be implemented efficiently.

2.5. Spector's Remote Operations

Spector's work on remote references/remote operations [49] describes a model of semantics for remote operations in a distributed system. The model, independent of any particular language or architecture, considers both low-level operations performed by system implementors and user-level operations.

Spector's model considers five attributes that apply to remote operations: (1) reliability - is the operation guaranteed to occur and how many times might it occur in the presence of communication or processor failures? (2) return value - does the reference require a return value? (3) flow control - are resources available at the site of the remote operation? (4) caller/callee synchronization - must the caller's process or processor wait for a reply? (5) operation types - is the operation sufficiently low level to be executed by the communications interface? This taxonomy of references is useful for characterizing communication in distributed systems. The programmer may specify the exact nature of the reference and only the overhead necessary is incurred. For example, one reliability attribute guarantees that an operation will be performed exactly once. This operation requires stable storage [30] to implement the semantics of the operation. Considerable overhead is often associated with such storage; only those operations requiring this level of reliability should use it. Distributed systems that provide a rich set of remote operations, as exemplified by this model, allow the user to choose a primitive whose performance and semantics closely match the intended usage.

An additional aspect of Spector's work was the implementation of a particular class of primitives defined by the model. Processor-synchronous, value-returning, remote memory load and store operations were implemented using Alto computers connected by a 2.94 megabits/second Ethernet. The primary result was that such references can be made extremely efficient on a local network, 155 microseconds per reference using Spector's microcode implementation. This promising result was a major motivational factor in our work in which we attempt to incorporate efficient communication primitives based on Spector's communication model into a high-level language.

2.6. Summary

A survey of relevant work suggests that, for the most part, there have been few attempts to design general-purpose, distributed programming languages that can be used to build higher levels of software support in a distributed system. Each of the many language proposals we have surveyed addresses some particular aspect of programming distributed systems. Distributed processes were developed for a specific set of application programs, Liskov's primitives primarily address one specific requirement (robustness), and Nelson considers one specific form of communication (RPC). Our work is an attempt to fill the gaps between these various approaches.

Chapter 3

AN OVERVIEW OF STARMOD

3.1. Introduction

The programming language StarMod [12] grew from the desire to extend Modula [56] to include facilities for distributed computing. The additions and modifications to Modula primarily focus on those aspects of the language intended to support this type of programming. In most other respects, the structure of StarMod programs is inherited from Modula. Since much of the work in this dissertation is based on StarMod, we describe in detail those features of StarMod that support concurrent and distributed programming. We also propose a new statement for StarMod that provides a timeout mechanism, a necessary addition used extensively by the kernel implementation.

3.2. Processes

A StarMod process declaration describes a procedure, complete with local declarations and statements, which is to execute concurrently with other processes. No assumption can be made about the relative speed of a process except that it is greater than zero.

```
process <identifier> (<formal parameters>) : <result type>;
  <local declarations>
  begin
    <statement list>
  end <identifier>;
```

A process is activated just like a procedure, using the procedure call statement. The only difference is that the calling program continues to execute in parallel with the newly activated process. The number of concurrent processes at execution time is limited only by the amount of available memory. There may be many activations of the same process executing concurrently; each instance of the process may have a different set of actual parameters. A process terminates when an **exit process** statement is encountered or when the end of the process is reached.

A process may return a value (the result type in the process declaration is optional), just as a procedure may be a value-producing function. For functional processes, the calling process must wait for the return value.

3.3. Signals

In general, processes communicate via shared variables, usually declared within an interface module. For long term scheduling, however, signals should be used. A variable of type signal is not a variable in the usual sense; there are no values that may be assigned to a signal. The only valid operations on variables of type signal are the following predefined procedures:

wait(s,p) - delay a process until it receives the signal s. The process is given priority p, imposing an ordering on processes waiting for a specific signal.

send(s) - send signal s to that process waiting for s with the highest priority.

awaited(s) - a boolean function that returns true if there is a process waiting for signal s.

3.4. Modules

A module encapsulates a set of declarations and procedures into a closed lexical scope. The interactions between modules must be explicitly stated in the interface specification section. Thus, the module is a syntactic construct for the implementation of abstractions and their interfaces.

```
module <identifier>;
  <interface specification>
  <local declarations>
  begin
    <statement list>
  end <identifier>;
```

A module describes a *closed* scope. That is, no identifiers declared in an enclosing scope are visible within the module unless explicitly imported. Similarly, no identifiers declared within the module are visible in the enclosing scope unless explicitly exported. These restrictions force the programmer to declare the interactions between modules. The compiler enforces the protection of data within a module, providing secure abstractions with well-defined interfaces.

As in Modula, a prefix in the module declaration specifys semantic interpretations for the module. Thus, the module can be used to delineate scope, specify synchronization, enclose logical processors, and describe network topology.

3.4.1. Interface Modules

A module prefixed by **interface** specifies that the module is to act as a monitor [24]. That is, only one process may execute within the module at one time, preventing simultaneous access to encapsulated data. Thus, interface modules provide a mechanism for short-term scheduling of processes.

3.4.2. Processor Modules

A module prefixed by **processor** describes the environment for a logical processor. In general, a processor module corresponds to a physical processor and a single shared memory. It encapsulates all processes that are to execute on a single processor. A processor module may encapsulate a number of processes that have access to the shared data of the processor module. Communication between processes within different processor modules, potentially executing on different physical processors, is in the form of messages.

It is possible for more than one virtual processor, as defined by a processor module, to share a single physical processor. Similarly, the implementation may map processes within a processor module to different physical processors to improve the parallelism. Nevertheless, processor modules provide a mechanism whereby the programmer can partition his distributed program into collections of processes, reflecting natural divisions between physical processors.

3.4.3. Network Modules

A module prefixed by **network** encapsulates programs for execution on a network of processors without shared memory. A network module is made up of processor modules and global (network-wide) declarations. The global declarations may include constants, types, procedures, and processes. No global variables or module initialization are allowed since we do not assume a shared memory exists for all processors. Network modules are useful for defining network-wide protocols, via constant, type, procedure, and process declarations, and ensuring standardization between processor modules.

A network module may specify the topology of the underlying hardware configuration. The links facility allows the programmer to specify what direct communication connections exist between processor modules. A link declaration specifies a source processor and a series of destination processors; communication may flow from the source to the listed destinations. For example, the following network module declares the communication connections for a five node star network:

```
network module star;
  (* Processor module communication links *)
  (center, north, south, east, west),
  (north, center),
  (south, center),
  (east, center),
  (west, center),
  <interface specification>
  <local declarations>
  end star;
```

This feature provides a compile-time check for communication between processes within different processor modules. Messages may flow between processor modules only if a communication link exists between them.

3.5. Ports

In StarMod, processes within different processor modules communicate via messages. Message communication is based on the concept of a port [4]. The port mechanism is described in detail in Chapter 4.

3.6. The Completion Statement

A fundamental way in which distributed programs differ from concurrent programs is in the notion of failure of an operation. If an operation within a concurrent program running on a single processor is unable to proceed without outside intervention, the entire system fails. An advantage of distributed systems, however, is the fault-tolerant nature of such systems; processors and processes are expected to continue execution in spite of the failure of other processors. Thus, it is imperative that remote operations recover from communication and processor failures.

Most distributed systems are constructed so that a processor does not wait indefinitely for a response from a remote site. Typically, a remote operation results in (1) completion with correct results, (2) completion with incorrect results, or (3) failure to complete within time t. Such a capability can be provided by a timeout specification. This does not mean that a timeout mechanism is available at the user level. Nelson has argued that, at the application level, timeouts should only be used for reasons of performance and not as a generalized error handler [44]. An exception handling capability is more appropriate for handling errors.

One method frequently used to express a timeout is to associate a timeout argument with each operation. This approach is common in operating systems in which operations are expressed as system procedure calls. For example, the command to send a message and wait for a reply might be:

```
Send (message, timeout) : reply;
```

This approach, used in RIG [32], has the disadvantage that a timeout facility has been provided at the expense of syntactic transparency; not all basic operations are expressed as procedure calls, so only certain operations may be time-dependent. We prefer to allow the user to specify timeouts for user-defined operations, not system-defined operations. In addition, the timeout mechanism should not be restricted to any particular set of operations, for example, remote operations.

To specify timing constraints within StarMod, we will use the completion statement, which takes the following form:

```
in time t do
   <statement list_1>
otherwise
   <statement list_2>
end;
```

The unit of time may be supplied by the user, but may not require more resolution than that provided by the implementation. If *<statement list$_1$>* has not completed execution within time t, *<statement list$_2$>* is executed. It may cause the operation to be reexecuted, modified, or aborted.

The completion statement has applications beyond the purposes we envision for the kernel. Since it is not attached to any fixed set of operations, it may be used to impose timing constraints on both local and remote operations. While we will use this statement as a rudimentary exception handler within the kernel, wherein a timeout is translated into a higher-level exception for the user, it is also applicable at the user level.

3.7. Summary

StarMod extends the modular philosophy of Modula to the distributed programming environment. Virtual networks are composed of virtual processors that communicate over specified communication channels. We will use these abstractions as tools in the further development of the StarMod language and its kernel to support various communication primitives for distributed systems.

Chapter 4

MESSAGE- AND PROCEDURE-ORIENTED COMMUNICATION

4.1. Introduction

Ever since the construction of the first network, it has been natural to model communication between processors using messages. The fundamental unit of communication at the hardware level is a packet or message; further interpretation is unnecessary for basic communication between processes. In effect, all communication between processors in a loosely-coupled, distributed system is, at some level, based on messages. We are primarily interested in the user's view of communication, however. We define message-oriented communication to mean that the user is explicitly aware of the messages used in communication and the mechanisms used to deliver and receive messages. On the other hand, procedure-oriented communication assumes that the user processes communicate using procedure calls; the underlying message communication is transparent to the user.

Many systems have been designed that use messages as the basic form of communication. Demos [5], Thoth [9], Medusa [45], RIG [32], and Arachne [20] are all examples of distributed operating systems in which processes communicate via messages. Programming languages that incorporate messages include PLITS [19], CSP [25], and CLU [38]. Various forms of remote procedure call are considered in Liskov's more recent work [40, 41], Distributed Processes [8], and RIG [32].

Lauer and Needham [33] argue that, within the context of their empirical model for communication in operating systems, messages and procedures have equivalent functionality. They conclude that a choice between these two forms of communication should be based not on the intended application, but on the lower level support for implementing the two primitives. While we agree that the primitives are essentially equivalent, we do not believe the resulting conclusion should be extended to the programming language environment. For example, if the ease with which the transition is made to the distributed environment is an overriding factor, one might choose remote procedure call as the preferred method of communication because many existing concurrent programs could be easily modified for a distributed environment without drastic changes to the program. Thus, in spite of the message/procedure duality, the decision on which primitive to use should be made by the user, not the language designer.

In this chapter we consider both message-oriented and procedure-oriented communication in StarMod, including declarations, semantics, and performance results of an implementation. We compare and contrast these two analogous forms of communication in an attempt to clarify their basic similarities and differences.

4.2. Message-Oriented Communication

Message communication between processes in a high-level language typically uses one of two techniques: the sending process designates either a fixed destination process or a fixed location for receipt of the message. In the first approach, the receiving process may also designate a willingness to receive the message, as in CSP and Ada.

If the sending process must name the receiving process, process names must either be static, as in CSP, or possibly, names are formed using a static process name and an instance qualifier. If process instances are created and destroyed dynamically, but process references are static, as in Modula, it is a distinct advantage that no process must name another to communicate with it. The communicating processes need only agree on a rendezvous location where messages are deposited and retrieved.

In StarMod, as in Modula, a single process declaration forms a template for (potentially) many instances of the process; each instance has the same name. To explicitly name a process before communicating with it, some mechanism would have to be introduced in the language to dynamically name processes. So, message communication between StarMod processes uses an approach based on the port mechanism of Balzer [4].

Ports provide a facility for interprocess communication that is especially appropriate for communicating processes residing on different processors. A port declaration defines the form of strongly-typed messages; a region statement acts as an independent message handler.

4.2.1. Port Declarations

A port declaration serves to define a queuing point for messages, that is, the interface between the sender and receiver. Ports are not bound to processes. Any process that defines or inherits a port name may access the port.

To emphasis the duality between the procedure call model and message communication model, a port call and procedure call have equivalent syntax. Thus, only procedure and port declarations need be modified if the model of communication is changed. This is an important factor that eases the problem of modifying concurrent programs for the distributed environment.

The syntax of a port declaration is:

port *<identifier>* (*<formal parameters>*) : *<result type>*;

The rules for associating actual parameters with formal parameters are the same as those used with procedures. A port may return a value; the type of the return value is specified in the port declaration. Ports that return a value are called functional ports or synchronous ports, since further execution of the sending process is synchronized with the execution of the receiving process. Ports that have no return value are called asynchronous ports.

All port parameters are passed by value. There are several good reasons for this requirement; only minor inconvenience results from its enforcement. (We assume here that the alternative is call-by-value-result. Thus, the overhead of a copy operation imposed by call-by-value is required in any case.) While we don't agree completely with Liskov that "*call-by-reference is not very useful in a distributed program*" [40], we do believe that the combination of asynchronous execution and reference parameters leads to potential aliasing problems and dangling references that are extremely unsettling. One unfortunate side-effect of requiring call-by-value semantics is that existing concurrent programs that make heavy use of reference parameters cannot be easily distributed by simply changing procedure calls to port calls.

To implement call-by-value semantics, a copy of each parameter is sent to the destination machine. Complex variables, especially those containing pointer fields, need to be *marshalled*. We refer to the packaging of a variable into a form amenable for transmission as marshalling. The basic data types of StarMod (e.g., integer, real, boolean) require little or no marshalling. Structured variables created by type constructors (e.g., array, record) and containing only simple data types are similarly easy to transmit. A more difficult case is a variable containing pointer fields. Since pointers represent local addresses not meaningful to a remote processor, the pointer's referent must be transmitted, not the pointer itself. A local copy of the referent is created and the result of the reference is a pointer to that copy. In the general case, this process could involve tracing a series of pointers through a data structure recursively, replacing each occurrence of a pointer with its referent. In Herlihy's thesis [23], mechanisms for transmitting cyclic and acyclic list structures are presented.

Not all distributed systems are made up of homogeneous processors, nor is every instance of an abstract data type implemented using the same representation. It may be necessary to marshall all data types between different processors if each processor uses a different implementation for each abstract type. Herlihy and Liskov [22] have examined the problem of translating abstract representations of data into external representations suitable for transmission.

Note that there is no body of code associated with a port declaration. The port only defines the interface between the process sending a message and the process that receives it. A region statement processes the messages associated with a port.

4.2.2. Region Statement

A region statement defines the message handler for one or more port declarations. It processes input messages (arguments to port calls) and generates return values for functional ports. The syntax of a region statement is:

```
region
  <port name_1> : begin <statement list_1> end;
        :          :          :             :
        :          :          :             :
  <port name_n> : begin <statement list_n> end;
end region;
```

Within the statement list associated with a port name, the formal parameters given in the port declaration are visible and may be referenced as if the statement list were the body of a port declaration. Similarly, the port name may be the object of assignment if the port is a functional port.

The semantics of the region statement specify the order in which messages are processed. If only one port identifier is specified in a region statement, the statement will wait until a message to that port arrives, after which the statement list is executed with the message body substituted for the formal parameters of the port. If more than one port is given within a region statement, the statement will select that port containing the message that arrived first (the statement will wait for a message if no port contains a message), and process that message. If more than one process is within a region statement waiting for a particular port, the region that will process the message, when it arrives, is chosen at random.

When a region statement completes, a reply message is returned to the process that sent the original message. The reply is of the type given in the port declaration and is referenced within the region statement using the port name.

One important difference between the ports/region scheme of communication and remote invocation is that a receiving process is not statically specified when a message is sent to a port. One can achieve the effect of remote invocation by assigning a single, static process to handle a functional port. The effect is then equivalent to a remote procedure call, although the semantics may vary in the presence of processor or communication failures.

4.2.3. Port Call Completion Semantics

The semantics of a functional port call specify that the calling process blocks until a reply is received from the process that accepted the message. Therefore, a functional port call fails to complete if no reply arrives within an acceptable period of time. The timeout interval is, of course, application specific and may be specified using the completion statement introduced in chapter 3.

It is not as apparent when a port call without an expected reply has successfully completed its execution. Consider the following situations:

Processor A regularly and frequently sends state information to processor B for load balancing purposes. The information requires sending short messages.

Processor A regularly, but infrequently, sends more complete state information to processor B, again for load balancing purposes. The information requires sending long messages.

Processor A sends state information to processor B with the expectation that, if the load balancing process on B is functioning, it will be processed soon, although processing may continue for some time (the message is large). Processor A expects no reply, but needs to know if the load balancing process on B is currently active.

Processor B requests more complete state information from processor A. Processor A replies with the requested information.

Each successive situation suggests that the port call completes later in time, representing four distinct points in the execution of a port call at which we can state the call has finished executing and, hence, the calling process may continue execution. The four stages (checkpoints) in the execution of a port call are: (1) when the message has been delivered to the network kernel on the local processor, (2) when the message has been accepted by the network kernel on the remote processor, (3) when the message has been delivered to a process executing a region statement on the remote processor, and (4) when the region statement processing the message has completed execution.

In the absence of communication and processor failures, and assuming a local network in which the average transmission delay is of short duration, checkpoints 1 and 2 are, from the user's perspective, equivalent. Internally, however, the two are not equivalent since checkpoint 1 requires that the message be buffered locally, introducing significant overhead for large messages. If the maximum packet size allowed on the network is large, the potential for long transmission delays exists for checkpoint 2. In addition, packet collisions may cause retransmissions that also delay the sending process.

The completion time at checkpoint 3 may be quite different from that at checkpoint 1 or 2 because the time interval between message arrival and message processing is unbounded. This is an attribute of the ports/region construct that is not a factor in remote invocation, since remote process representatives are, in general, instantly available if buffer space for the arguments exists. We are not sure if the difference is significant; we include it for completeness. Completion at checkpoint 3 is easily simulated by having a reply sent immediately on message receipt, although, within the context of the region statement, this would require buffering the message in the user process's address space before processing it. Alternatively, a separate reply message can be sent within the region statement using a port call to the originator of the message, requiring slightly more overhead than if checkpoint 3 were recognized automatically.

Checkpoint 4 is alreauy avcilable to the user, as it is equivalent to a functional port call.

To allow the user maximum flexibility in specifying the semantics of a port call, we extend the port declaration to include all 4 checkpoint possibilities. The termination specification of a port call, declared by a prefix to the port declaration, can be **immediate**, **on-receipt**, **on-delivery**, or **on-return**, representing checkpoints 1-4 respectively. (A null specification defaults to **on-return** if a result type specification is given, otherwise the default is chosen by the implementation.) An **immediate** port call will always complete if local buffer space is available. All other types of port call may not complete in the presence of communication failures or failures in the remote processor. Port calls that return **on-receipt** require only that the communications network and remote kernel process function correctly; **on-delivery** calls are not affected by errors in the receiving process.

In most message-passing systems, only one checkpoint is visible to the user. If performance is critical, no single checkpoint implementation is always satisfactory. Either long messages are needlessly buffered, introducing significant overhead on many microprocessors, or processes that send frequent, short messages may be delayed each time a message is sent while a long message passes by on the network.

The checkpoints we have identified apply, for the most part, to any message-based system. They are not peculiar to the ports/region construct, although, if processes must rendezvous to communicate, **on-delivery** and **on-receipt** may be equivalent. In any case, we believe the difference in the performance of primitives that use different checkpoints may be significant, so much so that the user should specify what level of performance is required.

4.2.4. Implementation

In this section, we describe an implementation of the StarMod ports/region mechanism and present some performance results. We compare our performance measurements with results obtained by others and offer suggestions for improving performance.

4.2.4.1. Implementation Description

Remote port calls are executed through the cooperation of three distinct StarMod processes: the user process, a network multiplexor process, and a network device driver process. The network multiplexor process implements the protocol that, among other things, guarantees that all messages are reliably received. (Reliability is dependent on a hardware checksum facility. Software checksums were used in initial tests, but were not used in the final tests reported on here.)

A user process that executes a functional port call blocks until a reply has been received. For port calls with no reply expected (non-blocking or asynchronous port calls), a user process posts a message for the multiplexor and suspends until that message has been accepted by the kernel of the destination machine. That is, if no checkpoint is specified in the port declaration, checkpoint 2 is assumed. We made this choice to avoid excess data movement which is extremely time-consuming on the PDP 11/23, especially for large messages.

In both cases, arguments to a port call are pushed onto the run-time stack just like procedure parameters. One difference is that extra space is pushed preceding the message for network packet overhead. This way, the run-time stack of the user process sending the message acts as an outgoing network packet, avoiding the necessity of copying each message into a different buffer. The disadvantage is that a process not expecting a reply cannot proceed immediately, but must wait for the message to be received by the destination processor. In our implementation, that waiting period is small and is deemed a worthwhile trade-off for improved performance.

In the absence of a result specification by the user in a port declaration, a straightforward modification to the compiler would allow it to generate two different kernel calls depending on the size of the argument list. Short messages could be buffered with little or no overhead, while long messages are transmitted directly from the user process's stack.

Our implementation does not allow all StarMod parameter types. In particular, pointer types and structures that contain pointer types were not implemented. No marshalling routines were implemented.

4.2.4.2. Implementation Performance

In Figure 4-1, the time required to execute local and remote port calls is given, both for functional (synchronous) ports and (asynchronous) ports with no return value. The return value for all functional ports was a simple completion indicator. The message body (port argument) was varied, from a simple integer to an array of

Execution Time (in ms.) for Asynchronous and Synchronous Port Calls

Message Size (in bytes)	Asynchronous Port Calls		Synchronous Port Calls	
	Local	Remote	Local	Remote
2	3.61	11.11	4.76	20.73
4	3.63	11.24	4.90	20.87
8	3.69	11.28	4.98	20.99
16	3.78	11.38	5.04	21.11
32	4.04	11.57	5.16	21:36
64	4.35	11.94	5.40	21.85
128	5.13	12.69	5.87	22.84
256	5.81	14.19	6.82	24.82
512	7.80	17.18	8.74	28.77
1024	12.38	23.17	12.53	36.65

Figure 4-1

characters 1K bytes in size, to measure the effect of message size on the performance. Figure 4-2 shows the program that was used to derive the local synchronous port call timing results; Figure 4-3 contains the program used for remote synchronous port calls. The hardware clock used has 1/60 second resolution. Loop overhead is included in Figure 4-1; it is less than 50 microseconds per message. The asynchronous port call results were derived similarly, except that the total time is computed by the receiving process, not the sending process. The two processes began executing (almost) simultaneously for those tests.

It is important to realize what has been measured. Owing to our inability to accurately time a single operation using a clock with 1/60 second resolution, an averaging technique was used to derive the timing results. While the time required

Timing Program for Local Port Calls

```
processor module TestLocal;
 type ArgumentType = ...;
 on-return port p ( arg : ArgumentType ) : integer;
 var ActualArg : ArgumentType;
 process sender();
  var CompletionIndicator : integer;
  begin
    StartTime := TIME();
    for i := 1 to 32000 do
      CompletionIndicator := p(ActualArg);
    end for;
    AverageTime := (TIME() - StartTime) div 32000;
  end sender;
 process receiver();
  begin
    loop
      region p : begin end; end region;
    end loop;
  end receiver;
 begin
   receiver(); sender();
 end TestLocal.
```

Figure 4-2

to execute a single port call and receive a reply is accurately portrayed in Figure 4-1, it is not clear what is meant by the time required to execute a port call with no reply expected. Since the port call is asynchronous, there are operations that are overlapped in time on the two processors. Figure 4-1 shows the average time required to receive a message given that the sender and receiver are devoted exclusively to the communication being measured. For large messages, that average indicates the performance one can expect when many messages are sent to a destination; it does not indicate the time required to send and receive a single message. This may be attributed to the execution overlap on the two machines involved in the communication; the sending process can begin to assemble a new message on its stack for transmission, while the receiving process is copying the last message received onto the user's stack for processing. Since block copy

Timing Program for Remote Port Calls

```
network module TestRemote;
 processor module reader;
  export p, ArgumentType;
  type ArgumentType = ...;
  on-return port p ( arg : ArgumentType ) : integer;
  begin
   loop
    region p : begin end; end region;
   end loop;
  end reader;
 processor module writer;
  import p, ArgumentType;
  var ActualArg : ArgumentType;
  begin
   StartTime := TIME();
   for i := 1 to 32000 do
    CompletionIndicator := p(ActualArg);
   end for;
   AverageTime := (TIME() - StartTime) div 32000;
  end writer;
end TestRemote;
```

Figure 4-3

operations are expensive on the PDP 11/23, this overlap in execution distorts the results for a single large message transmission. The time spent from the moment a process begins to assemble a message until a destination process is able to manipulate the message is probably 4 ms. or so greater than the time given, about the time required to copy 1K bytes. Nevertheless, the time spent before the sending process is allowed to continue execution (assuming an implementation of checkpoint 2) is correctly represented.

One observation of note is that the time required to execute a remote port call with no arguments and to receive a null reply is about 20 ms. This is the same figure that has been frequently reported by others [46]. Our hypothesis for this anomaly is as follows. A message-based communication system is sufficiently complex to admit many possible optimizations. Work on performance is important until an acceptable level of efficiency is achieved. According to Peterson [46], "*a time of about 20 milliseconds was quoted as the round trip time to send a (null) message and receive and answer on the Xerox Alto systems, with similar numbers put forth for IBM systems and Multics.*" Thus, most implementations will struggle to reach this performance goal, and then struggle no further. Nevertheless, we do not believe that is the case with our implementation because the remote port call facility has been streamlined for efficiency.

In any case, the similarity between the execution efficiency of our implementation and the often quoted average of 20 ms. is misleading. The Xerox Alto system, one of the message passing systems mentioned in [46], was implemented on a processor twice as fast as the PDP 11/23's used in our implementation and a communications medium, a 2.94 megabits/second Ethernet, with almost three times the bandwidth of our medium. The similarity in execution speed implies that our implementation is probably 2 to 3 times more efficient than the Xerox system.

Figure 4-4 illustrates the bandwidth utilization for both asynchronous and synchronous port calls for various message sizes. Bandwidth utilization is greater for synchronous port calls than for asynchronous calls if the message and reply are roughly equal in size. Since sending a message requires more (constant) processing time than sending a reply, many small messages utilize less bandwidth than many replies of equal size. If messages are large and replies are small, as is often the case when a reply is used as a completion indicator, bandwidth utilization is considerably greater for asynchronous calls.

As the figure indicates, communication using small messages does not consume a significant percentage of the available bandwidth, even though the transmission rate of the communications network is only one megabit/second. This means that in our implementation, the processor is the crucial bottleneck in message communication; even very large messages utilize only about 1/3 of the total bandwidth.

% Network Bandwidth Utilization for Remote Port Calls

Message Size (in bytes)	Asynchronous Port Call	Synchronous Port Call
2	1.44	1.54
4	1.57	1.61
8	1.84	1.75
16	2.39	2.05
32	3.46	2.62
64	5.49	3.73
128	9.20	5.81
256	15.45	9.48
512	24.68	15.29
1024	35.98	23.18

Figure 4-4

4.2.4.3. Comparisons with Charlotte

Charlotte, a descendent of Arachne [20], is a distributed operating system under development at the University of Wisconsin. It offers a unique opportunity to compare the performance of two different implementations, each with a different approach to the problem of support for distributed programming. Since Charlotte is currently being developed on the same network of PDP 11/23's that was used for our tests, it is especially interesting to compare the performance of the two implementations. (We should note that the Charlotte implementation will continue to undergo significant modifications. The performance results are preliminary.)

Message communication in Charlotte is based on send/receive primitives. The send primitive is nonblocking, that is, the sender may continue execution once the message has been buffered. The implementation determines whether buffering takes place locally or on the remote destination; currently, the sender blocks until an acknowledgement has been received by the remote destination. The current implementation uses an almost identical stop-and-wait protocol for communication as was used in the StarMod kernel.

The time required to send a message from one user process to another on a different machine under Charlotte is about 47 ms.; a process sending messages to itself requires about 29 ms. to send and receive a message [42]. The figures are not significantly affected by message size, for all sizes from 0 to 30 bytes. The same timing figures for the StarMod implementation are about 11 ms. and 3 ms. respectively.

The time required to send a message under Charlotte may be roughly divided into inter-process and inter-machine communication. Preliminary test results show that approximately 29 ms. per message is required for inter-process communication; inter-machine communication requires about 18 ms. per message. Thus, a process sending messages to itself takes about 29 ms. per message, without any context switching, while the same process requires an additional 18 ms. per message to send the messages to a remote process, again without context switching.

It is not surprising that Charlotte communication is significantly more expensive than our port call implementation. The underlying protocol in Charlotte is a layered protocol that tends to introduce considerable overhead. Other factors that contribute to the difference in performance include: (1) Charlotte is written in C and Modula, hence different compilers were used in the two implementations. The quality of the code produced by the various compilers could be a factor in the performance comparison. (2) The StarMod kernel is constructed using Modula-like processes. Charlotte contains a single system process and interrupt handlers; language-level context switches do not occur in Charlotte. (3) Charlotte provides dynamic communication links that require run-time processing. Nevertheless, we believe the structure of the two kernels is the primary reason for the disparity in performance. StarMod message communication exhibits a factor of 4 speedup over similar communication in Charlotte, using the same stop-and-wait protocol. The Charlotte kernel is a layered protocol that explicitly reflects inter-process and inter-machine communication. The StarMod kernel is tailored for the type of communication supported by StarMod and was designed for high performance.

4.2.4.4. Improving Performance

As we have previously stated, one of the most costly operations performed by the processor is copying large messages. Each time a message copy is avoided, performance can be greatly improved. One copy was eliminated from the outset; messages are transmitted onto the communications line directly from the user's stack without resorting to a system buffer. Thus, the only remaining data transfers of consequence occur from the sending process's stack to the destination processor and from the buffer area on the destination processor to the receiving process's stack. Transmission onto the communications line is unavoidable;

however, we can eliminate the second copy operation.

In StarMod, a message is received via a region statement. This statement delineates the scope of a port's parameters and, hence, the message itself. In effect, the user's stack buffers the message during the execution of the region statement. If the compiler generates code to reference the message body relative to a general purpose register, rather than the stack pointer register, then the message can remain in the original buffer and need not be copied onto the user process's stack. This modification improves the performance of synchronous remote port calls containing large messages (1K bytes) by 10.3% and local port calls for the same message by 30.8%. The improvement had little effect on the measured transmission rate for asynchronous remote port calls. Again, owing to the overlap in execution of the copy operations on the two communicating processors, the copy operation on the receiver was "free", as far as our measurements were concerned. In actuality, the user process would receive the message about 4 ms. sooner than before; however, certain memory organizations may preclude the use of this optimization.

If memory is organized into kernel and user address spaces, we would expect incoming messages to be buffered in kernel space until the user destination process is identified. The kernel process would then copy the message into the appropriate user process's space. In general, user programs are not allowed to reference kernel space, preventing any optimization that requires user programs to reference network buffers directly. Fortunately, microprocessor-based, local-area networks frequently employ single-user, single-address space machines as in, for example, process control systems. In these systems, it's conceivable that all programs run in the equivalent of kernel space, making it possible for all processes to reference network buffers directly.

Further significant performance improvements beyond those discussed would require more powerful hardware. For large messages, 25% of the execution time is spent on network transmission. The time required to send a 1K byte message to a remote site and receive a reply would decrease from 36.65 ms. to 25.5 ms. if the previous optimization were implemented on a 10 megabits/second network. Substantial improvements would also be expected if a faster processor were used as well.

4.3. Remote Procedure Call

Using Nelson's definition [44], "*remote procedure call is the synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel*". Even though remote invocation and message passing are analogs, message-oriented systems and systems constructed using remote invocation often differ in the relationship between communicating processes. Typically, messages are passed between cooperating processes that act as partners in a computation. Communication is frequently two-way, especially if messages are used to synchronize distributed processes. Remote invocation, in particular RPC, is based on a master/slave relationship. The invoking process is a master requesting some service and communication is one-way, from master to slave. (Although a remote call may return a value, it is only in response to a specific request. Control flows from the master process to the slave process and returns.)

In this section, we consider the design of an RPC mechanism for StarMod, highlighting those important features identified by Nelson as essential for a transparent RPC mechanism, and also present the performance results of an implementation.

4.3.1. Essential Properties of an RPC Mechanism

Nelson [44] identified five essential properties that an RPC mechanism should exhibit: uniform call semantics, type checking, full parameter functionality, concurrency control and exception handling, and distributed binding. Pleasant, but nonessential, properties of an RPC mechanism include good performance, atomic transactions, respect for autonomy, type translation, and remote debugging. We briefly consider each of the essential properties with regards to an RPC mechanism for StarMod.

4.3.1.1. Uniform Call Semantics

The primary advantage of the remote procedure call is that its semantics are inherited from the local procedure call, allowing the programmer to disregard the complications that typically arise in the execution of remote operations. A transparent RPC implementation must, to the fullest extent possible, maintain the same semantics used for local procedure calls. Otherwise, remote and local procedures must be coded differently, requiring the programmer to make decisions about program distribution early in the life cycle of the program.

Remote procedure calls are easily integrated into StarMod. Conceptually, a remote procedure call is a call to a procedure declared in some other processor module. Again, we do not insist that all processor modules are mapped onto individual physical processors, but a procedure call that appears at compile-time to be a remote call will be executed by the kernel at run-time. That is, the kernel will map the call to a physical processor at run-time. This may result in the execution of a local procedure call if more than one logical processor shares a single physical processor. In any case, the syntax and semantics are exactly the same for local and remote calls in the absence of machine crashes. (We assume that the kernel provides reliable messages.)

4.3.1.2. Type Checking

A remote procedure call is treated exactly like a local procedure call by the compiler. Processor modules are the unit of compilation, but the compiler supports strong type checking across compilation boundaries. Thus, the same level of static type checking applied to local procedure calls applies equally well to remote procedure calls.

4.3.1.3. Full Parameter Functionality

There is no conceptual or practical problem in allowing all basic data types of StarMod as parameters to a remote procedure call. StarMod contains all the basic types found in Modula, as well as pointer types. While pointer types do present certain difficulties for the system implementor, as will any type containing local addresses, transparency requirements force the admissibility of these types in remote procedure parameter lists. Dynamic arrays and procedure parameters, such as those found in Mesa [43], do not exist in StarMod.

4.3.1.4. Concurrency Control and Exception Handling

Concurrency control and exception handling are not a fundamental aspect of the RPC mechanism itself; however, reasonable support from the programming language in which the RPC mechanism resides is highly desirable. In StarMod, concurrency is provided by the process declaration inherited from Modula. A timeout facility was proposed in Chapter 3. That facility, together with a reasonable exception mechanism, such as that found in Ada, are sufficient support for a transparent RPC mechanism.

4.3.1.5. Distributed Binding

A programming language system that supports RPC must have some means of compiling, binding, and loading distributed programs onto a network. For our implementation, a rudimentary distributed binder was constructed that accepts as input a list of previously compiled processor modules and a mapping of virtual processors to physical processors. The binder modifies the specified object files so that internal tables are consistent with the processor mapping, then loads each processor module onto the appropriate physical processor. The binder currently supports only a 1-to-1 mapping between logical and physical processors.

4.3.2. Orphan Computations

One issue of importance with regards to the reliability of an RPC mechanism is how the system recovers from a remote procedure call that fails. To illustrate some of the difficulties involved, consider a distributed program running on three machines labeled A, B, and C. If procedure P_1 on machine A calls procedure P_2 on machine B, execution of P_1 is suspended until P_2 completes. If machine B crashes during execution of P_2, P_1 will remain suspended indefinitely. If we assume that the RPC implementation includes a timeout mechanism, then P_1 will eventually be allowed to continue and may invoke P_3 on machine C to perform the same task P_2 was assigned. Before P_3 completes, machine B might restart and attempt to recover from the failure. Recovery allows P_2 to complete execution, causing P_1 to receive results from a computation it had assumed no longer existed. The problem occurs because an *orphan* computation, namely P_2, was not exterminated when machine B crashed. Determining when to abandon a remote computation and ensuring that the results of an abandoned computation are not used is an important part of any RPC implementation.

Different techniques have been used to deal with orphans. For the most part they are variations of two different methods, extermination and expiration. These two approaches differ primarily in the amount of computation and stable storage required by the algorithm per remote procedure call and when a machine is restarted after a crash. We will present an overview of the two approaches here, and direct the reader to Lampson's work [31] and Nelson's thesis [44] for details.

Extermination is the process of finding and aborting orphan computations that result from a crash. Each processor records in stable storage its outstanding remote calls and the calls it is currently processing. After a crash, the remote calls being processed locally are aborted and machines processing outstanding calls are requested to abort those calls.

Expiration (deadlining), a technique analogous to timeout, is used to determine when a computation has existed beyond its expected lifetime; such computations are likely to be orphans. Each remote call carries with it an expiration time, the point at which the computation will automatically abort. Determination of the expiration time is a crucial parameter that will have significant effects on the performance of both remote calls and recovery. An expiration time that is unrealistically short will cause non-orphan processes to abort; too long an expiration time will slow recovery from crashes. A variant of this approach allows an expired process to request a postponement of its deadline, preventing non-orphan computations from aborting needlessly.

One advantage of an RPC mechanism over a message-oriented scheme is that, in many situations, the programmer would be required to duplicate the effect of an RPC implementation were it not supplied as a primitive. An RPC mechanism that provides at-most-once semantics (on return from a remote procedure call, the programmer is guaranteed that the procedure was executed exactly once) establishes a reliability goal that is difficult to attain using only messages. Unfortunately, orphan algorithms assume the existence of some form of stable storage. It may not be cost effective to provide such storage for each processor in a microprocessor-based network. If the network is required to reference stable storage, the overhead associated with a remote call may be significantly increased, even if little data is required by the orphan algorithm.

4.3.3. Implementation

4.3.3.1. Implementation Description

The RPC implementation was incorporated into the kernel that supports the port implementation previously described. In particular, a remote procedure call is implemented with the cooperation of a user process, a network multiplexor process, and a network device driver process.

The differences between a synchronous port call and a remote procedure call are minor; the code executed by a remote procedure call is a subset of the code executed by a synchronous port call. RPC does not need to examine and modify the complex data structures needed to support synchronous and asynchronous message communication. Some machine mapping functions are avoided because only remote (nonlocal) procedures are handled by the kernel, whereas both local and remote port calls are joint kernel operations. (If virtual processors are not mapped 1-to-1 with physical processors, this would no longer be true.)

As with port calls, a remote procedure call causes a network packet to be constructed on the calling process's stack containing the argument list. A mapping table determines the machine and address of the remote procedure and sends the argument packet to that site. The calling process then suspends. A representative process is created on the remote site that invokes the remote procedure with the appropriate arguments. On termination of the procedure, the remote representative process returns a value (possibly just a completion indicator) to the calling process and then dies.

We should note that a realistic RPC implementation must provide detection and recovery of failures. Our implementation does contain timeout detection, but we did not implement full recovery from remote machine failures. We would expect the performance to degrade somewhat, especially if repeated references to stable storage [30] are required to support the recovery algorithm. Nonetheless, the implementation provides a reasonable measure of RPC performance in the absence of machine crashes.

4.3.3.2. Implementation Performance

In Figure 4-5, the time required to execute a remote procedure call, with a null body, is shown for various size argument lists. Marshalling of parameter lists was not implemented, therefore, argument size is the most important variable factor in the performance of the remote procedure call.

4.3.3.3. Comparisons with Nelson's RPC

Nelson's thesis [44] was devoted entirely to the study of the remote procedure call. The detailed design of a transparent RPC mechanism is presented along with the performance results of a family of RPC mechanisms. Nelson's implementation executes on Dolphin processors (a successor to the Alto) connected by a 2.94 megabits/second Ethernet. Some tests were also performed using Dorado processors, a successor to the Dolphin that executes Mesa about 8-10 times faster.

RPC Performance and Bandwidth Utilization

Size of Argument List (in bytes)	Ms. Per Call (null body)	% Bandwidth Utilization
2	20.13	1.75
4	20.17	1.82
8	20.22	1.98
16	20.36	2.28
32	20.60	2.87
64	21.09	4.02
128	22.08	6.16
256	24.05	9.91
512	28.00	15.83
1024	35.88	23.77

Figure 4-5

Nelson's family of mechanisms includes an implementation that runs at each level of the Pup bytestream protocol [6]. A stub generator translates a Mesa interface into an RPC implementation that uses the Pup level 2 bytestream protocol or a special bytestream implementation, based on the the level 1 socket interface, that is significantly faster than Pup bytestreams. A high performance version that does not use the Pup protocol was also implemented.

The stub implementation, which uses Pup level 2 (bytestreams) on the Dolphin, requires from 24 to 28 ms. to execute a remote call with no arguments between two Dolphins on an unloaded network, depending on whether software checksums and byte or word operations are used. The optimized version using Pup level 1 (datagram service) requires from 10 to 11 ms. for the same call. The high performance version is able to execute a remote procedure call with no arguments in 0.8 ms. This last result is significant as it offers hope for tremendous improvements in the performance of all other remote operations. The considerable difference in performance within the family of mechanisms is primarily attributed to the overhead associated with the Pup protocol and the Mesa process machinery; the most efficient version uses neither. A measure of the best case performance that can be expected of an RPC implementation is the microcoded version that executes on the Dorado; it is capable of performing a remote call with no arguments in 145 us.

It is difficult to compare our implementation with Nelson's because both the environment and the implementations are considerably different. Nevertheless, the optimized version that uses the Pup level 1 protocol is most comparable to our "bare" machine implementation. In comparing the two implementations, we find that Nelson's is twice as fast. This is to be expected since the Dolphin processor is 2-3 times faster than a PDP 11/23 and the communications medium used by Nelson has 3 times as much bandwidth as our network.

It is worth noting that the significant difference in performance between Nelson's best case (without microcode) implementation and our implementation is misleading. The highly optimized versions of Nelson's implementation access the network device directly, similar to the type of operation we explore in Chapter 5. These implementations are based on the assumption that certain shortcuts are acceptable if the call itself will execute very quickly. Our RPC implementation did not take advantage of those shortcuts, primarily because there is no reasonably small upper bound on the amount of time required to execute a remote call. Had we implemented our RPC mechanism using similar techniques, we would have significantly reduced the execution time of a remote procedure call in our implementation.

One interesting result of Nelson's work is the quantification of the overhead of a typical bytestream protocol. Each level in the Pup protocol introduces about 10 ms. of overhead. It is precisely this type of overhead that a specialized language kernel is designed to alleviate.

4.3.3.4. Improving Performance

As with the other communication primitives we have considered, one of the most important factors in the performance of an RPC implementation is the frequency of data copy operations. One disadvantage of communication based on messages is that a message must be buffered between the sending and receiving processes. This is true because, in general, the destination process is not known in advance. Nonetheless, communication based solely on remote procedures can take advantage of the knowledge that each incoming message is a procedure invocation that will create a new instance of a process to execute the called procedure. (Other types of messages exist, for example, acknowledgements and replies to previous remote procedure calls; however, we assume that those messages are processed immediately and their buffer space is reclaimed.) Thus, we can read network messages directly into the stack space associated with the next process to be created, saving a copy operation. Figure 4-6 illustrates the effect of this performance improvement.

Recall that a similar technique was suggested to avoid a copy operation on receiving a message for a port. The limitation which applied, namely that the memory organization must allow a user process to reference memory allocated by the communications kernel, does not apply in this case since the buffer memory used is that of the user process's stack. The only requirement is that the available memory for process stack allocation exceeds the size of the largest possible network packet.

4.3.4. Remote Process Activation

Given the duality between procedure calls and messages, it is logical to consider a procedure-oriented analog of sending messages asynchronously. In single processor systems, procedure and process invocation create synchronous and asynchronous instantiations, respectively. In the same vein, we can extend

RPC Performance and Bandwidth Utilization
Without Argument Copy

Size of Argument List (in bytes)	Ms. Per Call (null body)	Percent Improvement	% Bandwidth Utilization
2	20.06	0.35	1.75
4	20.08	0.45	1.83
8	20.13	0.45	1.99
16	20.22	0.69	2.29
32	20.41	0.92	2.90
64	20.80	1.38	4.08
128	21.54	2.45	6.31
256	23.04	4.20	10.35
512	26.03	7.04	17.03
1024	32.02	10.76	26.63

Figure 4-6

the concept of RPC, a synchronous transfer of control between programs on different machines, to include remote process activation (RPA), the asynchronous transfer of control between programs on different machines.

Just as RPC inherits syntax and semantics from local procedures, RPA is derived from local processes. Creation of a process via a process call is an instance of RPA if the process is declared in a different logical processor. The original process may proceed asynchronously with respect to the newly created process, after the parameters have been buffered (at the discretion of the implementation).

RPA can be simulated using RPC. That is, each remote process can be invoked by first invoking an associated remote procedure that creates an instance of the remote process and returns. Such an approach would impose some overhead on the system performance, as well as force the programmer to write extra program code. Another, more compelling, rationalization, motivated by a desire for transparent distributed systems, is that we want to allow all forms of communication that normally execute on a single processor to be extended to the distributed environment. RPA is a logical extension of local process activation.

4.4. Broadcast Messages

Many of the local area networks in use are broadcast networks; each message directed onto the network may be received by any station connected to the network. This organization has the advantage of trivial routing algorithms (messages need only be routed between networks, as in inter-network mail delivery systems) and is more efficient than an organization in which messages must be passed from machine to machine until the final destination is reached.

We should note that not all local-area networks are broadcast networks. Broadcasting that is not directly supported by the hardware can be achieved by routing algorithms. Improved performance is attained by improved routing

[13,53,54].

Message broadcasting is the sending of a single message to all processors in a logical network, which may be a subset of a physical network. Usually, this feature is simulated by simply sending the message to each processor individually. This approach forces the programmer to define a different message handler in each processor and to send multiple copies of the same message onto the network, consuming considerable network bandwidth and local processing time for large messages.

It is not essential for a software system that supports broadcasting to be implemented on a broadcast network. The kernel may perform all routing of messages, so the complexity of broadcast communication is hidden from the user. The algorithms used to route messages would be no different (possibly) than those used to route individual messages. The only difference is that each processor stores the message before forwarding it.

In this section, we consider additions to the port mechanism of StarMod to support broadcasting. We also present the performance results of an implementation and compare those results with the expected gain in performance versus point-to-point communication.

4.4.1. Broadcast Port Declarations

Recall that, in StarMod, a network module encapsulates network-wide objects for use by all processor modules. Declarations within a network module provide consistency and standardization for global objects. To provide broadcast communication, we extend the notion of a local port, declared within a processor module, to that of a network port, declared within a network module. Network ports are queuing points within each processor for broadcast messages. A message sent to a network port is broadcast to every other processor module in the distributed program and may be retrieved on each processor from the local copy of the network port. The port need only be declared once in a distributed program; different types of broadcast ports can be created using a single declaration for each. Just as a broadcast network is the most efficient architecture for delivering packets to all stations on a network, a broadcast port is a much more efficient mechanism for delivering system-wide messages than many local ports. An empirical measure of the performance improvement attained using the broadcast capability will be given in a later section.

4.4.2. Broadcast Semantics

Broadcast port calls are similar to local port calls, differing only in the number of destination machines and the number of reply values expected. These differences affect the manipulation of reply values and the choice of checkpoints.

4.4.2.1. Broadcast Replies

A broadcast port that specifies a return value expects one from each processor module in the distributed program. Therefore, a broadcast port with a return value of type T actually returns an array of type T as a result. The index type of the array is an enumeration type containing the names of all processor modules in the program. This type is implicitly declared in a network module, wherein each processor module is declared, along with all other system-wide definitions. Thus, the value returned by processor X may be referenced using the array subscript notation with index X.

It is possible for communication or processor failures to prevent some processors from returning a value within the timeout interval; eventually, the port call will return with an exception. Some elements of the result array will contain valid responses, other elements will be undefined. The programmer should initialize the result array in an application-specific way, so that valid replies can be recognized and partial results processed.

4.4.2.2. Broadcast Completion Semantics

The points at which a broadcast can be said to have completed are the same as for local port calls. The difference is that many different processors may have to arrive at that point before completion. Hence, the checkpoint specification introduced for local ports may also be applied to broadcast ports. The trade-offs are the same in each case; however, wait time may be much greater for a broadcast and is dependent on the number of destination machines. Therefore, message size may not be a primary factor in choosing a checkpoint. For example, in normal situations, there is little difference between checkpoint 1 and 2. This is not true for broadcasts, since the time to receive acknowledgements from all processors is considerably greater than the time to receive one acknowledgement. Even large messages can be buffered in less time than that required to receive multiple acknowledgements.

4.4.2.3. Multicast Messages

Multicasting differs from broadcasting in that only a subset of the virtual network receives the message. Until now, we have assumed that the entire virtual network, which is a subset of the physical network, receives messages directed to network ports. In reality, we would prefer that the network port mechanism provide multicasting, since broadcasts are a special case of multicasting. We will modify the semantics of network ports to support this capability.

A network port declaration specifys a queuing point for messages that is made available to each virtual processor. Processor modules that import the network port name receive messages directed to it; broadcasts to a non-imported port are transparent to that processor module. The result type of a multicast is the same as before, an array indexed by processor module; however, not all components will be defined for all multicasts.

To determine when a multicast has received all its replies, each kernel must know how many virtual processors imported the network port. This value is a load-time integer constant created by the distributed binder. As the internal tables are modified to reflect the load-time processor mapping, a port mapping is also maintained. A load-time constant is created and stored in each port table entry indicating the number of replies expected for each multicast port.

4.4.3. Implementation

4.4.3.1. Implementation Description

StarMod programs are distributed onto a network partition, a subset of all available processors on the network. The **processor mapping table**, a compiler-generated table that is configured at load time, describes the partition of the network. In particular, the table describes the recipient processors of a broadcast. The megalink network hardware allows a processor to specify a secondary network address, an alias for the hard-wired network address. The secondary address is used for broadcasts; each processor in the partition sets the secondary address to the partition number. Only processors in the partition receive packets sent to this secondary address.

Each pair of processor modules in a distributed program are logically connected by a virtual channel. To broadcast a message, a processor must make available all its virtual channels. Once a processor receives a broadcast request from a user, only currently outstanding port calls are processed until all virtual channels are idle, that is, all outstanding requests have been sent and acknowledged. Then, the broadcast message is transmitted.

As the preceding suggests, it is possible for a port call to be suspended during execution, independent of the checkpoint specified in the declaration. A more complete implementation could take into account the urgency of a port request, as implied by the checkpoint specification, to determine whether certain port calls take precedence over broadcasts. Such a facility would be especially important for real-time systems in which processes execute at different priorities; a low priority broadcast would not prevent an urgent port call from executing. This discussion only pertains to the delay that occurs before the broadcast is actually transmitted, while all outstanding messages are acknowledged. Once the broadcast is transmitted, no communication can take place over any virtual channel until an acknowledgement has arrived on that channel.

Acknowledgements for a broadcast are sequenced to prevent the numerous collisions that otherwise would occur when all processors in a partition attempt to acknowledge the broadcast at the same time. Every processor delays about 1 ms. for each processor that precedes it in the partition, sufficient time to allow the preceding processor to acknowledge the broadcast.

The timeout interval before retransmitting a broadcast is dependent on the number of processors in the partition. The message is resent, using point-to-point communication, to each processor that does not acknowledge the broadcast within the timeout interval. Ideally, the message would be rebroadcast if enough processors failed to receive the initial broadcast. We did not implement this capability.

4.4.3.2. Implementation Performance

Figure 4-7 shows the performance characteristics of our asynchronous broadcast port call implementation. The failure rate was between 1% and 2% for most of the test runs. That is, from 1% to 2% of all broadcast messages were retransmitted because either the original message or an acknowledgement was lost. The message was then retransmitted to the particular machine(s) that did not acknowledge the message until an acknowledgement was successfully received by the processor issuing the broadcast.

Execution Time (in ms.) for Asynchronous Broadcast Port Call To Multiple Destinations

Message Size (in bytes)	Number of Broadcast Destination Machines 1	2	3	4	5	6
2	17.24	20.21	22.98	26.42	28.29	32.22
4	17.32	20.56	23.04	26.48	28.36	32.36
8	17.41	20.72	23.04	26.62	28.62	32.46
16	17.85	20.75	23.09	26.77	28.80	32.71
32	18.04	21.00	23.32	26.91	29.06	32.94
64	18.35	21.23	23.57	27.02	29.31	33.19
128	19.16	22.24	24.48	27.51	29.83	33.47
256	20.76	23.66	25.90	28.64	31.02	34.35
512	23.72	26.71	29.00	31.53	34.21	37.48
1024	29.62	32.78	35.13	37.50	41.25	44.02

Figure 4-7

As the first column of Figure 4-7 demonstrates, there is a fixed amount of overhead, about 6 ms., associated with broadcast communication beyond the amount needed for single-destination communication. This overhead can be attributed to increased computational requirements on the broadcasting processor. Extrapolation suggests that each additional destination machine requires 2-3 ms. to process, significantly less time than a point-to-point port call.

Figure 4-8 shows the speedup achieved by using broadcasting versus point-to-point communication. In reality, we would expect the improvement to be greater than that shown because the comparison was made with idealized point-to-point performance results; the improvement factor does not include the effect of network contention that arises in realistic systems. Contention is inherent in the broadcast capability (multiple destination machines may attempt to acknowledge a broadcast simultaneously) and is included in the performance figures presented.

As expected, there is a decrease in efficiency when the broadcast capability is used to communicate with only one machine. The overhead associated with broadcasting is not offset by any savings in transmission time. The speedup factor

Speedup Factor of Asynchronous Broadcast Communication vs. Point-to-Point Communication

Message Size (in bytes)	Number of Broadcast Destination Machines 1	2	3	4	5	6
2	0.64	1.10	1.45	1.68	1.96	2.07
4	0.65	1.09	1.46	1.70	1.98	2.08
8	0.65	1.09	1.47	1.69	1.97	2.09
16	0.64	1.10	1.48	1.70	1.98	2.09
32	0.64	1.10	1.49	1.72	1.99	2.11
64	0.65	1.12	1.52	1.77	2.04	2.16
128	0.66	1.14	1.56	1.85	2.13	2.27
256	0.68	1.20	1.64	1.98	2.29	2.48
512	0.72	1.29	1.78	2.18	2.51	2.75
1024	0.78	1.41	1.98	2.47	2.81	3.16

Figure 4-8

becomes significant when large messages are transmitted or more than four destination machines are involved. Not surprisingly, the theoretical speedup factor of N for a broadcast to N machines is not attained. In fact, the potential speedup appears to be severely limited. The following analysis describes the expected speedup for an arbitrary number of destination machines and also explains why the theoretical speedup is not achieved.

The time required to execute N point-to-point port calls (assuming no retransmissions) is

$$TPC(N) = N * (Pr + Tr + Ack_1 + Ack_2)$$

where Pr is the local processing time, Tr is the transmission time (including the time required to push the port arguments onto the run-time stack), Ack_1 is the time required to compose and send an acknowledgement on the destination machine, and Ack_2 is the time to receive and process an acknowledgement. A single broadcast to N destinations (assuming no retransmissions) requires time

$$TBR(N) = Pr + Opr + Tr + Ack_1 + N * Oack + N * Ack_2$$

where Pr, Tr, Ack_1, and Ack_2 are as before, Opr is the overhead of local processing associated with broadcasting and Oack is the overhead associated with sequencing multiple acknowledgements. Note that since the composition of acknowledgements is overlapped in time on the multiple destinations, time Ack_1 is required to compose and send N acknowledgements, excluding sequencing to avoid collisions, assuming transmission time is an insignificant factor in Ack_1. In our implementation, we estimate Pr = 7 ms., Ack_1 = Ack_2 = 2 ms., Opr = 5 ms., Oack = 1 ms., and Tr varies from 0.128 ms. to 8.304 ms.

Figure 4-9 shows projected performance results, using TBR(N), for large networks. As the number of destination machines increases, the speedup factor approaches the maximum, for this implementation, of about 3.7 for small messages

TBR(N) Projected Time (in ms.) and Speedup Factor vs. Point-to-Point Communication for Asynchronous Broadcast Port Call To Multiple Destinations

Destination Machines	2 Byte Message	Speedup Factor	1024 Byte Message	Speedup Factor
2	20.14	1.10	32.4	1.43
5	29.14	1.91	41.4	2.80
10	44.14	2.52	56.4	4.11
20	74.14	3.00	86.4	5.36
40	134.14	3.31	146.4	6.33
60	194.14	3.43	206.4	6.74
80	254.14	3.50	266.4	6.96
100	314.14	3.54	326.4	7.10
200	614.14	3.62	626.4	7.40
300	914.14	3.65	926.4	7.50
400	1214.14	3.66	1226.4	7.56
500	1514.14	3.67	1526.4	7.59
1000	3014.14	3.69	3026.4	7.66
2000	6014.14	3.69	6026.4	7.69
3000	9014.14	3.70	9026.4	7.70
4000	12014.14	3.70	12026.4	7.71
5000	15014.14	3.70	15026.4	7.71
10000	30014.14	3.70	30026.4	7.72

Figure 4-9

and 7.8 for large messages, representing the ratio between the time required to execute a point-to-point port call and the time required to process an additional broadcast destination. Using TBR(N), we estimate that a speedup factor of 3 is attained for small messages when 20 destination machines are involved; 5340 machines are required to reach a speedup factor of 3.7. (Figure 4-9 is slightly misleading because entries were rounded to a single decimal place.) Similarly, a speedup factor of 5.36 is possible for large messages with 20 destination machines; 3000 destination machines increase the speedup factor to 7.7. Additional destination machines have little effect on performance improvement.

The figures obtained for TPC(N) and TBR(N) using the estimated values are very close to the observed performance results for N = 1 to 6. The variance is about 1% in all cases. We conclude that TBR(N) is an accurate measure that can be used to estimate the performance of our broadcast port implementation as the number of destination machines increases beyond the number of machines for which we have empirical data.

We have shown that, in our implementation, the speedup achieved by broadcasting versus point-to-point transmission is limited by the time required to send an acknowledgement to the originating machine from each additional machine in the broadcast. By decreasing the acknowledgement time, we can effectively increase the speedup. One possible solution would be to use a token ring architecture that allows the communications adapter to acknowledge a message by setting a bit in the trailing portion of the message as it passes by on the ring. In such a case, acknowledgement time would be zero, allowing the speedup to approach the theoretical maximum.

Figures 4-10 and 4-11 show the performance and speedup for broadcast port calls with a reply value. The amount of traffic on the network was greater than that necessary because message acknowledgements were not piggybacked with

Execution Time (in ms.) for Synchronous Broadcast Port Call To Multiple Destinations

Message Size (in bytes)	Number of Broadcast Destination Machines					
	1	2	3	4	5	6
2	24.42	31.67	40.02	45.99	52.84	59.17
4	24.48	31.72	40.10	46.11	52.91	59.24
8	24.55	31.90	40.23	46.21	53.18	59.33
16	24.68	32.21	40.44	46.32	53.32	59.48
32	24.95	32.76	40.62	46.55	53.64	59.77
64	25.40	33.14	41.13	46.90	54.02	60.31
128	26.40	34.29	42.09	47.89	55.13	61.46
256	28.37	36.37	44.26	49.97	57.21	63.41
512	32.33	41.00	47.91	54.65	60.93	67.10
1024	40.32	48.38	54.33	61.12	67.81	74.25

Figure 4-10

return values. In a realistic environment, the acknowledgements would be delayed until a reply is ready, greatly decreasing network contention. The kernel does not support a delay to piggyback acknowledgements because, in nearly all our test cases, the overhead associated with waiting for a carrier message was too great and the probability of an out-going message arrival in the near future was too small to justify waiting for an out-going message. (Acknowledgements are piggybacked if a carrier message is waiting. Such was not the case in our test programs.) In particular, an asynchronous port call with a 2 byte argument would have required more than twice as much time as is currently necessary because the minimum delay is too long and the logical separation between the port mechanism and communication protocol prevents the protocol from knowing that a reply is expected. Thus, in this instance, the performance of the broadcast capability is adversely affected by a design decision that supports efficient point-to-point communication.

Speedup Factor of Synchronous Broadcast Communication vs. Point-to-Point Communication

Message Size (in bytes)	Number of Broadcast Destination Machines 1	2	3	4	5	6
2	0.85	1.31	1.55	1.80	1.96	2.10
4	0.85	1.32	1.56	1.81	1.97	2.11
8	0.85	1.32	1.57	1.82	1.97	2.12
16	0.86	1.31	1.57	1.82	1.98	2.13
32	0.86	1.30	1.58	1.84	1.99	2.14
64	0.86	1.32	1.59	1.86	2.02	2.17
128	0.87	1.33	1.63	1.91	2.07	2.23
256	0.87	1.36	1.68	1.99	2.17	2.35
512	0.89	1.40	1.80	2.11	2.36	2.57
1024	0.91	1.52	2.02	2.40	2.70	2.96

Figure 4-11

4.5. Summary

Message- and procedure-oriented remote communication, including broadcasts, have been incorporated into StarMod while maintaining the philosophy and structure of the language, including modularity, transparency, strong typing, and a virtual representation of networks.

Our implementation results are consistent with other reported experiences [46] in that the time required to send a null message and receive a null reply is 20 ms. or so. To place our performance results into proper perspective, one must take into account the following factors:

(1) Our implementation uses a microprocessor and communication medium that do not take advantage of state-of-the-art technology. We believe the figures we have presented could be easily improved by a factor of 2 to 5 with sophisticated hardware.

(2) We provide asynchronous, as well as synchronous, communication. Asynchronous message communication is more efficient, by a factor of two, than synchronous message communication. Users may choose the primitive most appropriate for the application, with an improvement in performance if the less sophisticated, asynchronous primitive is applicable.

(3) The execution time of remote message communication is within a factor of 3 of local message communication. With most other primitives, the difference

between local and remote operations is an order of magnitude slowdown in execution speed.

(4) Remote communication to multiple machines using broadcast ports can improve performance by a factor of 1.1 to 7.8 depending on the message size and number of broadcast destinations. The performance improvement saves both communication bandwidth and processor execution time, benefiting other network users as well as other local processes.

Both message- and procedure-oriented communication should be supported by the underlying software in a distributed system. An implementation that supports one form of communication can be easily modified to allow the other; the result should be comparable performance for both forms of communication. Message communication, as we have defined it, allows a process to communicate with others via ports without naming individual processes. Remote procedure calls allow the user to write programs using the same familiar constructs employed in single processor systems. By allowing both forms, the user is free to choose the most appropriate form for the problem at hand, rather than being forced into an uncomfortable mode of communication.

Message-based communication should also include broadcasting, particularly if the network is a broadcast network. While the performance improvement was not as dramatic as expected, primarily because of the need to sequence acknowledgements, a significant improvement in the speedup factor of broadcasts could be readily attained with the addition of a sophisticated network interface. In particular, either collision detection or a token ring would make the sequencing of acknowledgements unnecessary, decreasing the time required for each machine to send its acknowledgement.

Chapter 5

REMOTE MEMORY REFERENCES

5.1. Introduction

A remote memory (variable) reference is a memory (variable) reference issued by a processor that is resolved by another processor's memory. We will use the terms **remote memory reference** and **remote variable reference** interchangeably, recognizing that the two operations are not equivalent. A remote memory reference is a reference to a machine-specific quantity, a memory location, while a remote variable reference is a reference to a language-specific quantity, a variable of any size, type, or structure.

The primary advantage of remote memory references over remote procedure calls or message communication is, as we shall demonstrate, that a remote memory reference can be reasonably executed as a **processor-synchronous** operation. That is, the local processor issuing the request waits for the request to be satisfied or for a timeout to occur, rather than attempting to do other useful work. This way, multiple context switches are avoided, resulting in extremely high performance, an order of magnitude higher than other communication primitives.

In his thesis [44], Nelson strongly recommends that neither local nor remote global variables be used, instead a (remote) procedure should be used to encapsulate and provide access to data. In this chapter, we offer a different point of view on remote memory references. First, we address two important concerns associated with remote memory references, the conflict with data abstraction and the lack of synchronization. Then, we discuss the incorporation of remote variable references into a high-level programming language and present the results of an efficient implementation.

5.2. The Data Abstraction Conflict

Modern data abstraction techniques attempt to isolate data dependencies within local modules. Two issues of concern are *locality of control* (only the defining module may modify its local data structures) and *locality of effect* (only the defining module need be modified to implement changes to local data structures). The main criticism of remote variable references is that such references violate the integrity of data abstraction. In this regard, there is a similarity between remote variable references and inter-module variable references. The lack of uniform references [21] in nearly all programming languages prevents inter-module variable references from supporting locality of effect. To maintain the spirit of data abstraction without uniform references, programmers compensate using one of two techniques. One approach requires that the programmer use a procedure call to reference data from outside the module of definition. This way, the structure of a reference is isolated within a procedure in the defining module. The language implementation may provide a form of macro-substitution, so that little or no run-time overhead is incurred. In those cases where the language implementation does not support the in-line substitution of code, the programmer pays a severe penalty, usually an order of magnitude, for using this abstraction technique.

Another approach is to allow all references to data from outside the defining module to explicitly reference the data, but all such references are read-only. Since variables are only modified in the defining module, locality of control is enforced. Still, references from outside the defining module require explicit knowledge of the representation of the data; if the structure of the data changes, the effect is not localized. This approach, used in Modula [56], has the advantage that inter-module variable references are as efficient as local references, while maintaining locality of control.

Remote variable references are simply inter-module references in which the enclosing modules are processor modules. Either of these two techniques can be applied to remote variable references; most distributed programming language proposals incorporate only the first option. Nevertheless, by allowing a program to explicitly read, but not write, remote memory, locality of control is still preserved, with a significant improvement in performance.

5.3. Synchronization Issues

One advantage of message communication and remote procedure call is that synchronization between distributed processes is inherent in the communication primitive. The remote procedure or message handler executes in the context of a user process on the remote machine, respecting local synchronization. Remote memory references, however, have no inherent synchronization, nor is an explicit context for execution defined. This is an important point that must be addressed before remote memory references can become practical.

We should note that not all variable references require synchronization. Mutual exclusion is often used to coordinate readers and writers of data; if the control flow dictates that readers and writers never conflict, there is no need for this form of synchronization. Variable references that do not need to be synchronized should not have to pay the overhead of synchronization for each reference. In particular, references to static tables that, once initialized, are never modified, do not require synchronization after the initialization phase of execution.

We can provide synchronization for those remote variable references that must be synchronized using either of two approaches. One approach is to provide synchronization for logical groups of references using some other primitive, possibly messages, to define critical regions or rendezvous points. For example, consider a database application in which there are reader processes and writer processes, with priority given to readers. A control process monitors the state of the database, which may be either read-enabled or write-enabled. A process that wants to read (write) sends a request message to the control process asking permission to read (write) the database. A response to a read (write) request message signals permission to proceed; all incoming writer process requests are excluded by the control process. Subsequent data retrieval operations may be performed by the reader processes using remote memory reference since mutual exclusion is ensured. A final message exchange signals the control process that all operations by the reader processes are completed, at which time the writer process is sent a message allowing it to continue.

This approach has the advantage of being programmer-defined; no modifications or extensions to the communications kernel or compiler are necessary. The disadvantage is that, just as with semaphores, programmer-defined synchronization of critical regions is error-prone. Remote memory references are not inhibited, even if the control process is bypassed. In addition, failure to send a completion message may cause other processes to needlessly wait for the database to become free.

Alternatively, we can attempt to ensure that each remote memory reference is individually synchronized with respect to any process that might modify the data, just as an interface module ensures that each interface procedure call is individually synchronized with respect to other interface procedures within the module. A synchronization scheme for this purpose is detailed in a later section.

We conclude that synchronization is not incompatible with remote memory references. Just as different levels of synchronization, from no synchronization to mutual exclusion, are used to coordinate references to local data, we can devise similar synchronization primitives for remote memory references.

5.4. Remote Variable References

The primary issues that need be addressed when incorporating remote variable references into a high-level language are transparency, synchronization, address binding, and efficiency of execution. In the following sections, we present a design for remote variable references within StarMod and the performance characteristics of an implementation.

5.4.1. Transparency

A transparent remote variable reference mechanism for StarMod must allow a user to import and export variables of any type across processor module boundaries. Note that we distinguish between *static transparency* and *dynamic transparency*. Static transparency is a qualitative measure observed at compile-time; dynamic transparency is a quantitative measure observed at run-time. If a remote variable reference is indistinguishable from a local variable reference at compile-time, the reference exhibits static transparency. Unfortunately, it is unlikely that any remote reference implementation will be as efficient as a local reference. This significant difference in execution speed, observed at run-time, makes dynamic transparency an unrealistic goal. Nevertheless, if inter-module references are infrequent and efficient, reasonable dynamic transparency can be attained.

We assume throughout this discussion that the static processor boundaries delineated by processor module declarations provide enough information to the compiler to determine whether an operation should be treated as a remote operation. Some references, compiled as remote operations, may actually be local references at run-time if processor modules are mapped to processors many-to-one.

5.4.1.1. Transmitting Complex Types

To provide static transparency, it must be possible to reference a remote variable of any type. Using the same technique applied to port parameters, complex variables, especially those containing pointer fields, are marshalled before being transmitted.

The basic data types of StarMod require no marshalling and are retrieved by kernel requests of the form:

```
RemoteVar (machine, address, 1);
```

This request is for 1 word of memory from the machine and address specified. Structured variables created by type constructors (e.g., array, record) are retrieved by kernel requests of the form:

```
RemoteVar (machine, address, n);
```

where n is the size of the variable, a compile-time constant unless dynamic structures are allowed.

5.4.1.2. Qualified References

One disadvantage of remote variable references is that arbitrarily large data structures might be marshalled and transmitted unnecessarily. In particular, if a reference to a pointer variable causes its referent to be transmitted, an entire list structure is marshalled and transmitted, even if the pointer is only used in a dereferencing operation for a simple object. To optimize remote variable references, we will perform type qualification on the remote instance of the variable and transmit the resultant, fully-qualified object as the value of the remote reference. We will couch the discussion in terms of the StarMod type qualification operations that are available in most modern programming languages, but the concepts extend naturally to handle other type constructors.

The three qualification operations of interest are pointer dereferencing, array element selection, and record qualification. These three operations can be expressed using two qualification operators, indirection and indexing. Remote variable references for qualified objects will be constructed using a syntax similar to compiled machine code. We do not propose actual machine code so as not to preclude heterogeneous networks.

The format of a kernel request for a remote variable reference of any type is:

RemoteVar (machine, address, op_1, ... op_n, typetag, size);

where **machine** is the processor where the remote reference is to be satisfied, **address** is the remote address of the variable, each operator, **op_1** to **op_n**, is one of {INDEX, INDIRECT} followed by a (potentially null) argument, **typetag** is a base type specification, and **size** is the size of the qualified object, if known.

The operators INDEX and INDIRECT are used to represent the machine language constructs typically employed in compiling qualified references. The operator INDEX represents an indexing operation used for referencing fields of records and subscripted array elements. All index operators are followed by the value of the index as determined by the local machine; computation of the index may itself require a remote variable reference. The operator INDIRECT represents indirect addressing and is used in pointer dereferencing.

More specifically, let M_x be the machine identifier for the machine that exports variable x, A_x is the address of x, T_x is the type tag for x, and S_x is the size of x. Variable x is referenced by the following:

RemoteVar (M_x, A_x, op_1,...op_n, T_x, S_x);

If x is of type pointer to y then, x^ is referenced by:

RemoteVar (M_x, A_x, op_1,...op_n, INDIRECT, T_y, S_y);

If x is a record type with field y, x.y is referenced by:

RemoteVar (M_x, A_x, op_1,...op_n, INDEX, O_y, T_y, S_y);

where O_y is the offset of field y within record x. Finally, if x is an array of type y then, x[z] is referenced by:

RemoteVar (M_x, A_x, op_1,...op_n, INDEX, z', T_y, S_y);

where z' is the value of the index z modified to reflect the virtual origin and element size of the array type (if known).

The compiler generates these encodings of qualified references each time a remote variable is referenced. Rather than generate machine code that performs indirection and indexing, a remote request that represents the qualification is constructed on the run-time stack. Then, a call to RemoteVar is generated. The correct number of words are transmitted starting at the address calculated by interpreting the arguments of RemoteVar.

Each type or variable that requires marshalling that is exported by a processor module has an associated routine to marshall instances of the type. The routine may be programmer specified, as in [22], or compiler specified. If the result of a qualified reference contains pointer fields or if different data representations are used by different processors, the result must be marshalled. In either case, the need for marshalling can be determined at compile-time. The **typetag** field conveys the necessary information. It may contain the address of a marshalling routine or a static type identifier used to calculate the address of a marshalling routine.

By performing type qualification on the remote machine, we may drastically reduce the amount of marshalling and transmissions that must occur. A static analysis of Pascal programs [11] has shown that more than half of all pointer references contain dereference qualification; more than 82% of all array references contain subscripts. We conclude, therefore, that the overhead of performing remote qualification would be more than offset by the savings in transmission time, local processing time to marshall large structures, and local data space to buffer copies of remote variables.

5.4.1.3. Pointer Types

A serious problem with transparent remote references involving pointer types is that a program segment may produce different results depending on how processor modules are mapped to processors. For example, consider the following code segment:

```
(* q is of type ^ BaseType *)
var p : ^ BaseType;
:
p := q;
:
if p = q then ... else ... endif;
```

If q is a local (non-remote) variable, the conditional expression will be true. On the other hand, if q is a remote variable, assignment to p will cause the referent of q to be copied into the local address space; the value of p would be the address of the local copy of the referent, while q contains the address of the remote copy of the referent. Thus, the conditional expression will only be true if the semantics of pointer comparisons are changed to compare the pointers' referents, not the addresses contained in the pointer locations.

There is no transparent solution to this problem that does not change the well-known semantics of pointer equality. But, rather than prohibit all remote variable operations involving pointers, we will subscribe to the caveat used for floating point arithmetic. Just as equality comparisons between floating point quantities may not be meaningful, equality comparisons between pointers on different processors may not be meaningful.

5.4.1.4. Type-Based Synchrony

Not all remote variable references should be treated equally. The basic motivation for these remote references is the potential for extremely high performance attained by issuing the references as processor-synchronous operations. There is an important difference, however, in the expected performance between a remote variable reference that reads a single integer and one that retrieves a large array structure. Similarly, remote references to variables that must be marshalled before being transmitted are likely to take significantly longer than references to simple variables, so both type and size are important factors. Transparency dictates that remote references to large structures look like references to simple variables; reason dictates that some references are executed processor-synchronous and others are not.

In many circumstances it is possible for the compiler to determine whether a reference will delay the processor for an unacceptable length of time. Such references can be automatically issued as **process-synchronous** operations, that is, only the requesting process is delayed. This ensures that no processor is idle while a time-consuming, complex remote variable reference is in progress.

Of course, there are many situations in which the compiler cannot know at compile-time how long a reference will take. Dynamic structures may be very small or very large and no compile-time decision can be made as to whether the reference should be issued as a processor- or process-synchronous operation. In that case, the reference is issued as a processor-synchronous operation with a very short timeout interval. If a reply is not received within the specified period of time, the reference is aborted and reissued as a process-synchronous operation.

5.4.1.5. Failure and Retry

Certain circumstances may cause a processor-synchronous reference to automatically retry some operation that has failed. For example, if the communications line is busy when a processor-synchronous request is issued, the request may wait a short period of time, depending on the mean packet length, and try again. If requests are sequenced, the remote request may be retransmitted when a reply is not received within the expected time limit. A processor-synchronous reference that ultimately fails to complete within an acceptable time frame is reissued as a process-synchronous reference.

A remote reference is unable to succeed as a process-synchronous reference only if the remote processor has failed or if communication between the two machines is disabled. Both of these events are well-defined exceptions. Thus, remote reference retry and failure is masked from the user. Exceptions are used to pass unusual conditions that occur during a remote variable reference to the user, including processor and communication failure.

5.4.2. Synchronization

We have already discussed situations that use remote memory references without synchronization or with programmer-defined synchronization. In this section we consider how to provide system-defined synchronization for remote variable references.

In StarMod, local processes are synchronized for mutual exclusion purposes using interface modules (monitors). One approach to remote variable reference synchronization is to provide the effect of a monitor for data accessible by remote processors. That is, a remote variable reference is executed as if it were a call to an interface procedure within the module. The remote reference contains, as an argument, the address of the semaphore associated with the interface module. The semaphore is examined to see if the interface is occupied. This is a special operation, not a P operation, as the process cannot be allowed to wait if the interface is occupied. Most of the time we expect that the reference will find the module unoccupied and will be able to return a result immediately. In those cases where the interface module is occupied, the remote variable reference is aborted and reissued as a process-synchronous reference. It is not possible for the reference to read an inconsistent copy of the data because no process can enter the interface module while the reference is executing. This is true because the reference is executed as a processor-synchronous operation with respect to both the local and remote processors.

If the time spent by any one process within an interface module is short, as should be the case, why can't a remote variable reference still be executed as a processor-synchronous reference, even if forced to wait for the interface module? A request cannot be executed as a processor-synchronous operation on the remote machine and then be forced to wait for an interface module, as no other process, including the current owner of the module, will be allowed to continue. A new process representative for the reference has to be created and queued for the module. The overhead usually associated with process creation, deletion, and switching makes it impractical, in most cases, to wait for an interface module during a processor-synchronous reference.

If a processor-synchronous reference is reissued as a process-synchronous reference, the remote processor creates a process representative for the reference. The reference process will perform P and V operations on the semaphore and is allowed to wait for the interface if necessary. Thus, the time required to copy the data onto the network or into a network packet is a protected critical section.

Note that we have only extended local synchronization to include remote references executed locally. References to global variables not defined within an interface module are not included; by definition, the programmer has implicitly declared that no synchronization is necessary.

In all cases where synchronization is required, the compiled form of remote variable references must be modified to include synchronization information. In particular, the reference must contain a synchronization flag and the address of a semaphore. Alternatively, remote references can be mapped into a compiler-generated, interface module description table on the remote processor that contains the address of the semaphore for the interface module, as well as the address limits of its data structures. This approach may be preferable if processor modules are separately compiled as semaphore addresses may not be available. In either case, the appropriate information can be made available at compile-time and is statically generated.

5.4.3. Address Binding

As previously discussed, remote variable references take the form:

RemoteVar (machine, address, op_1, ... op_n, typetag, size);

These requests, generated at compile-time, contain compile-time quantities (machine identifier, nonrelocated address, etc.) that must be transformed to run-time quantities. A **distributed loader** loads a distributed program consisting of processor modules onto a network and modifies the processor modules' internal tables to reflect the program distribution.

Each processor module is assigned a static identifier at compile-time. A **processor mapping table**, produced by the compiler and modified by the distributed loader, specifies the mapping from static processor identifiers to actual machine identifiers. When processor modules are distributed among physical processors by the distributed loader, the processor mapping table is updated to reflect the run-time distribution of modules.

Remote variable addresses are resolved using a **segment address table.** Each segment, corresponding to a compiler-generated relocation counter, has an entry in the segment table with its base address. Typically there are two segments, text and data. When a processor module is loaded, the distributed loader updates the segment table with the appropriate base addresses. Variable addresses are maintained as offsets from the start of the segment in which they reside. Remote variable requests can then be resolved by adding the segment table entry to the offset given in the request.

5.4.4. Implementation

In this section we describe the design and performance of an implementation of StarMod remote variable references. We also compare our performance results with those obtained by Spector [49] and offer suggestions for further performance improvement.

5.4.4.1. Implementation Description

As we have said, remote variable references are executed as processor-synchronous operations. The reasons for this approach include:

(1) Remote memory references that retrieve small data structures are sufficiently simple and efficient that the processor is not idle for very long.

(2) The time overhead associated with the additional context switching required for process-synchronous references is comparable to the wait time required to process a remote variable reference (2-4 bytes in length).

(3) The time required to use the more general communication mechanism, including additional context switches (at least four), would greatly increase the time for a remote memory reference.

(4) The minimum process time-slice interval, as determined by the hardware clock, is much greater (by a factor of 20) than the time to execute a remote memory reference. Hence, most remote variable references from a process will finish executing within that process's time slice.

A remote memory reference is compiled into a call to procedure RemoteVar. This procedure assembles a request packet containing the arguments to RemoteVar and attempts to transmit the packet to the destination machine. Any outstanding read request that has not been serviced (the network device is always reading when not writing) is interrupted. If the communications line is busy, RemoteVar will delay for at most 100 microseconds and try again. This is sufficient time to allow any small packet to pass, including acknowledgement packets and requests by other machines. (If the mean packet length is considerably larger than the 12 bytes needed for acknowledgements and remote memory requests, the delay should be increased accordingly.) When the request has been transmitted, the local processor continuously tests the network device to see if a reply has arrived. (Allowing the interrupt handler to perform this function would significantly increase the total time required for a reference.) When the reply arrives, it is pushed onto the stack, the network device is reset to read, and RemoteVar returns to the user process.

The remote processor handles remote memory requests at the device level. Incoming messages are checked for validity, then, remote memory requests are processed immediately. A preallocated network packet is filled with the reply information. Data is copied into the network packet, which is then transmitted. The requesting processor will ignore the reply if it had already aborted the request.

Only simple remote variable references were implemented, including all simple types, arrays (without pointers), and records (without pointers). Qualification and marshalling were not implemented.

5.4.4.2. Implementation Performance

Figure 5-1 shows the time required to execute a remote variable reference for data structures of varying sizes and the network bandwidth utilization for each reference. The results were obtained by using the following timing program:

```
StartTime := TIME();
for i := 1 to 32000 do
   LocalVar := RemoteVar;
end for;
AverageTime := (TIME() - StartTime) div 32000;
```

Loop overhead was calculated by executing the timing program without the remote variable reference; loop overhead was not included in our timing results. In addition, for complex assignments (i.e., large arrays), the time required to move the values from buffer storage into the local variable was not included. Note that all timing results presented are based on successful execution of a processor-synchronous

Remote Variable Reference Performance and Network Bandwidth Utilization

Size of Remote Variable (in bytes)	Microseconds Per Reference	% Bandwidth Utilization
2	880	14.55
4	977	14.74
8	1028	17.12
16	1122	21.39
32	1324	27.79
64	1730	36.07
128	2547	44.60
256	4178	51.70
512	7439	56.57
1024	13963	59.47

Figure 5-1

remote reference. If the communication medium is unavailable or a response is not received in a short period of time, the remote variable reference is reexecuted as a process-synchronous reference. This requires significantly more time to execute since the more complex protocol mechanism used to implement synchronous ports is brought into play. Depending on the average packet length, which determines the amount of time a remote variable reference will wait for a free communications line, a remote memory reference reexecuted as a process-synchronous reference can take from 18 to 20 ms. to retrieve one word, slightly less than the time required to execute a synchronous port call. This small improvement in performance is due to the fact that remote memory references require less processing time than a port call and a user process is not invoked to reply to the remote reference.

Two processors communicating via remote variable references 2 bytes in size can impose a 14.55% load on the one megabit/second network; the effective data rate is 18 kilobits/second. Remote variable references 1024 bytes in size can impose a 59.47% load on the network for an effective data rate of 586 kilobits/second.

The data should not be interpreted to mean that we advocate using a processor-synchronous implementation for operations requiring over 10 ms. to execute. Instead, the data allows the system designer to determine which remote variable references should be executed as processor-synchronous references and which references should be executed as process-synchronous references. The data suggests that variables up to 8 bytes in size, more than 98% of all references [11], can be retrieved by a processor-synchronous reference without undue delay. The compiler could generate calls to a processor-synchronous operation for remote references up to 8 bytes in size, requiring about 1 ms. to execute, and a process-synchronous operation for remote references to large data structures requiring significantly more than 1 ms. to execute.

Figure 5-2 shows the breakdown of the instructions executed for each remote variable reference. Since the kernel implementation is written in a high-level language (StarMod), some instructions could be saved by coding in assembly language. In particular, the procedure setup and exit routines are more complicated than necessary for this application. Also, a generalized block move routine, written in assembly language for efficiency, was required to construct a packet containing the data. (In a later section that considers performance improvements, the copy code is rendered unnecessary.) We estimate that coding the entire operation in assembly language would yield an improvement of about 8% in performance.

5.4.4.3. Comparisons with Spector's Remote References

In Spector's thesis [49], an experiment performed on Alto computers connected by a 2.94 megabits/second Ethernet is described. The Alto has an internal cycle time of 180 nanoseconds, executes approximately 330,000 instructions per second, and has a memory bandwidth of 29 megabits/second. Remote operations, including remote memory reference, were implemented in microcode. It was shown that on an unloaded Ethernet, microcoded remote memory

Instruction Breakdown for Remote Variable Reference

Requesting Machine	
Push arguments to RemoteVar	4
Procedure setup for RemoteVar	10
Setup request	30
Await completion loop	3
Setup read request	7
Await completion loop	3
Remote Machine	
Handle interrupt	35
Process switch (if necessary)	36
Setup response	32
Copy reply data to buffer	10 + 2 * size
Send reply	33
Requesting Machine	
Cleanup	16
Procedure exit for RemoteVar	10
Pop arguments to RemoteVar	1

Figure 5-2

references could be executed in 155 microseconds; a single processor is capable of issuing 5000 remote memory references per second. A software version, using the raw datagram facilities of PUP Level 0 for packet transport [6], executes the same remote reference in 4.8 ms.

The corresponding remote reference in our StarMod implementation requires 844 microseconds on a one megabit/second network; a single processor is capable of issuing 1090 remote memory references per second. (The implementation was "special-cased" for 1 word requests for comparison purposes, saving 36 microseconds over the general implementation). A 2.94 megabits/second network would decrease the total transmission time per reference from 112 microseconds to 38 microseconds, enabling a remote reference to execute in 770 microseconds. Since the Alto is approximately 2 times faster than an PDP 11/23, the execution time could be reduced to 404 microseconds on an Alto. Assuming a speedup factor of 2 to 5 when microcoding software, our implementation would require from 81 to 202 microseconds if coded in microcode on an Alto. This estimation is consistent with Spector's results.

A comparison of Spector's software version with our implementation shows the advantage of specialized language kernels. Our remote memory reference executes almost 6 times faster than Spector's BCPL implementation, using hardware that is half as fast. Clearly, the overhead associated with the raw datagram facilities of PUP Level 0 protocol is prohibitive for remote memory references. Our implementation demonstrates the efficiency advantages of specialized high-level language kernel operations over more general layered protocols.

While it is difficult to draw conclusions from a comparison of such drastically different experiments, we believe the following claims are supported:

(1) The StarMod implementation of remote memory references is extremely efficient given the available hardware.

(2) The lower levels of standard network protocols are too general to efficiently support remote memory references.

(3) Results from work on efficient low-level communication primitives implemented in microcode or with specialized hardware may be incorporated directly into a high-level language implementation with little or no loss in performance.

(4) Remote memory references executed as processor-synchronous operations within a high-level language implementation are an order of magnitude more efficient than most other communication mechanisms and, assuming a reasonable level of transparency, are a valuable primitive for communication in a distributed program.

5.4.4.4. Effects of Qualification on Performance

The implementation of remote variable references has been made very efficient by limiting the amount of processing needed to satisfy a reference. Remote qualification adds complexity to the reference that increases both transmission and processing time. Each level of qualification adds at most two words to the argument list; approximately 45 microseconds are needed to push these arguments onto the run-time stack and transmit them to the remote processor. The qualification operations must be interpreted by the remote site, requiring up to 40 microseconds per qualification. Thus, we estimate that each qualification operation requires on the order of 85 microseconds, or 10% of a simple reference. This is an extremely small price to pay to avoid marshalling and transmitting large structures.

5.4.4.5. Effects of Synchronization on Performance

The synchronization method presented earlier has little effect on the performance of remote variable references. The semaphore implementation allows a process to determine whether an interface module is occupied in one machine instruction. Synchronization information in the request packet adds 16 microseconds to the transmission time. We estimate the total time overhead associated with synchronization to be 50 microseconds, or about 6% of a remote reference. This does not include the unfortunate case in which the interface module is occupied.

5.4.4.6. Improving Performance

As the size of the data structure retrieved using a remote memory reference increases, the time required to perform the operation is dominated by the copy costs of assembling a network packet (assuming a fixed communications bandwidth). The copy is unavoidable in most circumstances because the packet header information must immediately precede the data. There are two solutions to this problem; one involves special hardware and the other requires modifications to the compiler.

One way to avoid assembling a special network packet is to build a network interface that accepts two addresses; one for the packet header and one for the data or packet body. There are two advantages to such an approach. In a system that uses memory management, the packet header can be composed in kernel address space and the data can remain in user address space. This avoids the problem of copying data from user space to kernel space and vice versa. Additionally, data would not have to be copied into packet buffers, as the data and packet header may now be assembled independently. Network interfaces that implement this *scatter-gather* technique are currently available.

An alternative solution is to allocate space for packet header information in memory before each data structure. The compiler can determine whether the time/space trade-off is reasonable for a particular data structure and, for large structures, simply allocate additional storage preceding the structure. The internal copy costs would then be zero. Figure 5-3 shows the time required for remote variable references using this technique. Note that network utilization for large structures has risen from 59.47% to 89.28%. Our experiments show that this approach improves performance by 8% to 20% for small data structures of 4 to 64 bytes and up to 33% for large data structures of 1024 bytes. The expected improvement on a processor with a hardware copy instruction would not be quite as high. This approach can be used by both processors involved in the communication

Remote Variable Reference Performance Without Copy

Size (in bytes)	Microseconds Per Reference	Percent Improvement	% Bandwidth Utilization
2	880	0.0	14.55
4	899	8.0	16.02
8	931	9.3	18.90
16	995	11.3	24.12
32	1128	14.8	32.62
64	1391	19.6	44.86
128	1915	24.8	59.32
256	2966	29.0	72.83
512	5066	31.9	83.06
1024	9301	33.4	89.28

Figure 5-3

if suitable compiler modifications are made. That is, the remote machine could transmit a response packet containing the actual data structure, not a buffer copy. The local processor could read large data structures directly into the location desired, for example, the run-time stack for parameters or local data space for complex assignments.

Careful examination of Figure 5-3 reveals that the execution time is still a function of the length of the remote reference, even though the copy operation has been eliminated. We attribute the excess time to interrupt routines; the longer an operation requires to execute, the more likely it is to be interrupted by some other process. In particular, the clock handler and process scheduler are inherent to the kernel and must execute periodically, interrupting all other processes except processor-synchronous operations, which complete before handling the interrupt. Therefore, time-consuming, processor-synchronous operations have a high probability of being immediately followed by an interrupt routine. The effect is to slightly distort the timing results for long references by approximately 2%.

Another option to greatly improve performance is the installation of a communications medium with higher bandwidth. The one megabit/second transmission rate of our experimental network is clearly not state-of-the-art. Ethernet networks [14] with a bandwidth of 10 megabits/second are currently available and technology advances suggest that 100 megabits/second networks will be available within the next few years. Figure 5-4 summarizes the effect this technology would have on the timing results we have demonstrated (ignoring for the moment the issue of PDP 11/23 memory hardware support for 100 megabits/second DMA). The projected estimates of Figure 5-4 were derived using 755 us. as the constant processing time of a reference, excluding transmission time. The transmission time for different length references and bandwidths was then added to the basic processing time. This way, we are able to factor out the

**Projected Timing Results (in Microseconds)
for Higher Bandwidth Networks**

Number of Bytes	1 Mb	10 Mb	100 Mb
2	880	767.8	756.28
4	899	769.4	756.44
8	931	772.6	756.76
16	995	779.0	757.40
32	1128	791.8	758.68
64	1391	817.4	761.24
128	1915	868.6	766.36
256	2966	970.0	776.50
512	5066	1174.8	796.98
1024	9301	1584.4	837.94

Figure 5-4

discrepancy caused by interrupt routines that was previously mentioned.

Again, the data should not be interpreted to mean that we advocate building networks with relatively slow microcomputers and a 100 megabits/second network. Instead, the data demonstrates that even inexpensive microcomputers can communicate very efficiently using remote variable references, if the communications medium is powerful enough to support those references.

5.4.5. Protection and Autonomy

Two issues associated with remote memory references not yet discussed are remote memory protection and processor autonomy. Mechanisms must exist that prevent rogue processors from examining remote memory locations where access is restricted. Also, a local processor should reserve the right to prohibit remote memory references from executing.

One approach, the one favored by StarMod, is to use a uniform address space within a distributed program. In StarMod, a processor module encapsulates a logical processor and defines the interface with other processors. Variables exported by a processor module may be read using a remote variable reference; all other memory locations are protected against outside inspection by the compiler. This protection is extended to qualified references by requiring that the type be exported before qualification is permitted. Run-time checking can be used to ensure that qualification isn't used to access otherwise inaccessible memory.

In addition, all StarMod network packets carry a program identifier, preventing other programs from interacting maliciously with a running StarMod program. Processor autonomy is enforced at compile-time and is reflected by the export list associated with a processor module.

An alternative approach is to base remote operations on some form of capability [18]. The primary advantage of the uniform address space in StarMod is that protection is enforced at compile-time, not run-time.

5.5. Summary

In this chapter we described how to extend the variable importation and exportation philosophy of StarMod across processor boundaries in a transparent fashion. In addition, we presented the results of an implementation that suggest that remote variable references can be made reasonably efficient without constructing special hardware or resorting to microcode. Other studies [49,50] have shown that such references can be made extremely efficient if special hardware or microcode is employed.

A comparison between the performance of memory references, both local and remote, and message communication may be criticized on the grounds that memory references require synchronization techniques that greatly increase the performance of the reference. Without this synchronization, the references are deemed impractical. This criticism is only partially valid. Even though such a comparison is not between primitives of equal functionality, it is logical to compare the performance of various methods of communication in those circumstances where any one of a number of alternatives is reasonable; certainly performance would be an important criterion in primitive selection. We have shown that remote variable references can be synchronized without introducing significant overhead (in the general case). In the worst case, the performance is slightly less than message communication, since a reference that fails is immediately reissued using message-based communication.

One of Spector's goals [48] was to have a processor issue 1% of its non-instruction memory references to a remote processor, yet suffer no more than a 50% speed degradation. Spector was able to achieve this goal within the context of a special-purpose implementation in microcode. Using our implementation, a StarMod program in which references to remote variables are 1% of all variable references will degrade in speed by approximately 59.3%. (Assuming an average instruction time of 6 us. for the PDP 11/23, a program executing a remote reference 1% of the time would execute 67,843 instructions/second, instead of 166,666 instructions/second.) Even though the goal was not met, and probably could not be met within the restrictions of our environment, we believe the result is a promising one.

Chapter 6

A FAMILY OF COMMUNICATION PRIMITIVES

6.1. Introduction

Having designed a programming language that supports multiple models of communication and analyzed an implementation of the run-time kernel, we draw on these experiences to compare and contrast the communication primitives we have considered. In this chapter we compare the various models of communication that have been discussed, using both qualitative and quantitative criteria, summarize our experiences in constructing the kernel implementation, and consider architectural support for a distributed programming language kernel.

6.2. Comparing Models of Communication

There are numerous factors that influence the choice of communication primitive; the relative importance of these factors dictates which model of communication best meets the circumstances. In this section, we will reexamine the communication primitives in light of these factors.

6.2.1. Performance

From a performance standpoint, shared memory is the most efficient form of communication between processes. Where shared memory does not exist, remote memory references are a viable alternative. We have shown that a remote memory reference issued as a processor-synchronous operation can be executed very quickly; however, higher-level forms of communication are often desired. Asynchronous port calls require 10 times more execution time because they cannot be reasonably executed as processor-synchronous operations. Synchronized communication (e.g., RPC) requires twice again as much time to execute because of the need to process a reply message.

Aside from these absolute measures, there are other factors that influence the performance of a distributed program. We will examine some of those factors to determine how they affect the performance of the individual communication models.

6.2.1.1. Network Contention

Network contention affects the performance of the various communication primitives in different ways. Remote memory references are most severely affected because a reference that is unable to execute quickly must be reissued as a process-synchronous reference. In our implementation, that results in a factor of 23 degradation in execution speed. A processor that issues 1% of its non-instruction memory references to remote memory and fails to execute a remote memory reference processor-synchronously 1% of the time, will slow from executing 166,666 instructions/second to 60,060 instructions/second, a 64% degradation. The same processor issuing all remote memory references as processor-synchronous references will execute 67,843 instructions/second. To avoid the overhead introduced by network contention, a small maximum packet length ensures that a remote variable reference will be able to use the communications line when needed, allowing the reference to execute processor-synchronously.

Broadcasts are also greatly affected by contention. Depending on the implementation, a broadcast that is not acknowledged by all intended recipients may be rebroadcast to all nodes or sent to each individual node that did not receive the initial broadcast. Ideally, the issue of how to retransmit the message would be based on the percentage of nodes that received the original message. In our implementation, a broadcast message is retransmitted individually to each node that did not acknowledge the message. Because a broadcast message requires some computation by every node, each rebroadcast caused by contention influences performance, however slightly, on every node.

The performance of port calls and remote procedure calls is not as severely affected by contention. Each time the device driver attempts to send a message, but is unable to do so because the communications line is busy, a timeout interval is scheduled and the device driver process is suspended. An exponential back-off algorithm helps avoid congestion of the line immediately after a long packet has passed. The only unnecessary delays in our implementation introduced by contention are, once again, caused by the long minimum timeout interval.

6.2.1.2. Interaction Between Communication Primitives

Since the StarMod kernel supports multiple models of communication, it is possible for one primitive to interfere with the efficient execution of another. In particular, each of the communication primitives can affect the performance of a broadcast because broadcast communication requires that each virtual channel between the local processor and every other processor be free. A broadcast request is queued until all virtual channels are free; subsequent point-to-point communication requests are also queued in the interest of fairness.

The execution time of a remote memory reference can be severely affected by other communication primitives because such references are sensitive to network delays. The arrival of a port call at a processor awaiting a response to a remote memory reference may cause the response to be lost. For higher-level primitives, this is not a serious drawback, since those primitives are not as sensitive to transmission errors. However, remote memory references executed as processor-synchronous references are not retransmitted until successful. Instead, the reference is reissued as a process-synchronous reference, at a high cost in performance.

6.2.1.3. Local Vs. Remote Communication

Before distributing a concurrent program to improve performance, one must consider the type of communication used by the concurrent program to determine how to best distribute the program and also how to estimate the expected improvement in performance. If processes in the distributed version of the program are to communicate using the same form of communication as the concurrent program, the ratio between the execution times of the remote and local form of communication will, in part, dictate the amount of speedup that can be achieved by distributing the program. Primitives with a high ratio of remote to local execution time may not achieve expected speedup because the increased parallelism will be partially offset by the increase in communication costs.

Figure 6-1 shows the execution times and ratios of the communication models as derived from our implementation. Not surprisingly, the highest ratios occur for those communication primitives whose local execution is directly supported by the hardware, procedures and memory references. The complexity of the port call implementation causes the execution time to be quite high for a local operation, significantly lowering the ratio between remote and local execution. The result: distributing a concurrent program whose processes communicate via shared variables will not improve performance as much as distributing a concurrent program

Ratio Between Remote and Local Execution Time
for Various Communication Models with 2 Byte Arguments

Communication Type	Remote	Local	Ratio
Asynchronous Port Call	11.11 ms.	3.60 ms.	3.1:1
Synchronous Port Call	20.73 ms.	4.76 ms.	4.4:1
Memory Reference	880.0 *us.*	4.27 *us.*	206:1
Procedure Call	20.13 ms.	84.0 *us.*	240:1

Figure 6-1

whose processes communicate via ports (assuming the same increase in computational parallelism for both). Therefore, higher performance is attained by distributing those processes that communicate via ports to different physical processors and grouping processes that use shared variables on the same processor. This is a general rule that should be used when distributing a concurrent program to improve performance.

6.2.1.4. Overhead of Marshalling Parameters

The need to marshall arguments is not an important point for comparison because all communication models require marshalling; in fact, we expect that all communication primitives would use the same marshalling routines (if marshalling in-line is not appropriate). The ratio of overhead attributed to marshalling versus the total time required to execute a primitive may be unacceptable, however. That is, it might be considered unreasonable for a remote memory reference to request a data structure that would require considerable effort to marshall and transmit. To ensure that no processor is needlessly idle, it would be good programming practice to limit the amount of marshalling associated with a remote memory reference.

6.2.2. Ease of Implementation

Our experience has shown that each of the models of communication we considered can be implemented with a reasonable amount of effort. Asynchronous message passing is the easiest to implement, broadcasting the most difficult. While it is not possible to rank the relative difficulty of implementation for each communication mechanism, primarily because we did not undertake a complete implementation of each primitive, we can draw some conclusions based on our experience.

The type translation mechanism is the same for each communication model because all support full functionality; each primitive permits arguments of any type. Once a type translation scheme is implemented, a nontrivial task by itself, it ceases to be an issue for comparison among the various primitives.

Messages require more code to implement than RPC because of the nondeterministic nature of the ports/region concept. On the other hand, the machinery necessary to guarantee the semantics for remote procedure call in the presence of crashes was not implemented. Since the orphan algorithms represent a considerable amount of work, we conclude that RPC is significantly more difficult to implement than port calls; the latter do not require orphan algorithms at all.

The processor-synchronous memory reference is easy to implement, but only if some other form of process-synchronous communication already exists. The first attempt to execute a remote memory reference does not require the more complex, secure protocol used for messages. If a remote memory reference fails to succeed, however, it must resort to a higher-level primitive, preferably one that is also available to the user. Thus, an implementation that supports only remote memory references as the model of communication must build, internally, much of the implementation used for message passing, even though user processes are not allowed to communicate via messages.

6.2.3. Functionality

The type of support for communication that a primitive provides is an important factor in determining appropriate situations for its use. For example, remote procedure call has been criticized as "inappropriate" for distributed database implementations; the master/slave relationship imposed by RPC is artificial in some situations, particularly in the cohort-sponsored recovery of a transaction that occurs when a coordinator fails [17]. An IPC facility based on a peer-peer relationship is preferable.

A primary reason for supporting multiple forms of communication is to allow the user to choose a communication model appropriate for the application. The primitives we have considered offer different levels of functionality for different program environments.

6.2.4. Familiarity of the User Interface

One advantage of RPC is that its syntax and semantics are inherited from local procedures. Programmer familiarity makes RPC easy to use and less error-prone. In addition, concurrent programs based on procedure calls may be easily distributed to improve performance without significant code modification. Unfortunately, a familiar user interface may be a disadvantage in that the overhead associated with remote communication may be hidden by innocuous appearances. This is particularly true about remote variable references. Nevertheless, given a choice, many programmers will choose a model of communication with which they are familiar, often without due consideration of the performance trade-offs involved.

6.2.5. Formal Tractability

The inherent complexity of distributed programs has led many researchers to investigate proof techniques for communication primitives [2,3,27,28,35,29,47]. In an environment in which program correctness proofs are deemed necessary, a model of communication might be chosen based on its formal tractability. For example, Schlichting and Schneider [47] suggest that the ports/region communication mechanism, based on static references to a communication channel, is preferable to a communication mechanism in which arbitrary processes may communicate (e.g., dynamic communication links in Arachne [20]) because the reduced number of pairs of processes that may communicate require fewer satisfaction formulas to be constructed. Synchronous forms of communication, such as remote procedure call and port calls with reply, are shown to have much simpler proof rules than asynchronous communication primitives; the increased parallelism introduces added complexity, suggesting that the proof rules for broadcasting are likely to be even more complex.

A formal argument can also be made for inclusion of remote memory references. Proofs for distributed programs typically view each process in isolation during the sequential part of the proof, but eventually must consider the entire system's state. If a remote memory reference is considered to be an atomic operation (our implementation suggests that, to some degree, this is a reasonable assumption), it may be used by a process to interrogate the state of other distributed processes. Thus, the programmer is no longer limited to local memory references while constructing a program whose proof requires global information.

We do not believe that the formal properties exhibited by any of these models of communication should be used to justify their prohibition. Factors such as performance, implementability, and ease of use are more likely to be of primary concern in the implementation of real systems. It is frequently the case, however, that formal arguments clarify practical experience; the harder it is to reason formally about a program, the harder it is to understand the program. So, formal tractability is one of many factors to consider when comparing communication primitives.

6.3. Lessons in Kernel Construction

During the implementation of the StarMod kernel, whose performance has been previously detailed, we attempted many different approaches to various problems to determine the effect each approach had on the complexity and performance of the kernel. While performance may be used as an objective measure of the success of a particular technique, the complexity of each approach was measured subjectively, based on the amount of code required for its implementation, the amount of debugging required, and the ease with which corrections were applied. The result of these different approaches to the implementation of the kernel is a series of lessons or general principles that apply to the construction of communication systems in general and distributed programming language kernels in particular, lessons that are especially important when performance is a primary criterion.

6.3.1. Balance Simplicity and Performance

A delicate balance between simplicity and performance must be maintained to avoid unnecessary complexity or extremely poor performance. For simplicity, an original version of the kernel was constructed in which each incoming message was handled by a process created for just that purpose. The performance, per message, was not considered acceptable; process creation, deletion, and context switching were simply too time-consuming compared to the amount of time the process handler actually executed. As an experiment, the kernel was recoded so that only one process acted as both a device driver and message handler. This approach soon became overly complicated and error-prone. In addition, performance suffered because the network device was inactive while messages were processed, causing the next incoming message to be lost. Our eventual solution was a compromise between these two extremes. A simple device driver process, dedicated to initializing the device and processing its interrupts, minimizes the time during which the device is neither reading nor writing. A second process serves as a virtual channel multiplexor/demultiplexor. This process implements the communication protocol and calls routines to quickly handle incoming messages. The resulting organization suggests a good compromise between program complexity and performance.

6.3.2. Simple Protocols Are Not Simple

Simple communication protocols are not simple to program. We implemented a stop-and-wait protocol, using a 1-bit sliding window, because the complexity of allowing multiple outstanding packets in a sequence was not justified, particularly in a local-area network where mean transmission time is small. Modifications to the basic protocol, as presented in Tanenbaum [51], were required to allow the protocol to reach an idle state in which no extraneous transmissions are made between kernel processes; even more changes were required to implement broadcasting. A general pattern emerged in which seemingly minor changes often had undesirable effects on performance.

6.3.3. Don't Increase Traffic to Improve Performance

A distributed control protocol, such as SDAM [36,37], can be used to improve throughput in the face of network contention. SDAM assumes that network nodes can eavesdrop, receiving the source, destination, and direction of each packet on a bus-oriented network. An implicit token is contained in the delay between transmissions, in effect, sequencing each node's access to the communications line. End-nodes, residing at each end of the bus, are used to generate control tokens for directing token flow.

Alternatively, explicit tokens can be exchanged to sequence access to the network. This may be appropriate for networks that are (a) unable to peek at each packet on the network, invalidating the assumptions of SDAM and (b) not heavily congested with communication traffic. Our experiences suggest that these mechanisms designed to improve performance, at a cost of increased network traffic, frequently succeed only at the latter, to the detriment of the former. For example, an acknowledgement sequencing scheme for broadcast messages was attempted in which each processor in the sequence would wait for a continuation message from the previous member of the sequence, acknowledge the broadcast, and then send a continuation message to the next processor in the sequence. The rationale for this was (1) the acknowledgements must be sequenced to avoid numerous collisions and the resultant long delays because the hardware did not provide collision detection, (2) the message would allow the next processor in the sequence to acknowledge the broadcast almost immediately after the previous one, since the communications line is certainly free and the messages are short, and (3) even loosely synchronized clocks would have been too inefficient because the clock resolution was not high enough. Unfortunately, if a continue message is lost, it must be resent after timeout, again introducing the clock. The added complexity did not improve the performance; on the contrary, performance degraded because many continuation messages were lost. Instead, acknowledgements were loosely synchronized using busy-wait loops of about 800 us. in duration. The average performance was much better than that attained using the previous scheme.

6.3.4. Use Simple Packet Structures

The fundamental object of concern in a communication kernel is the packet. Experience suggests that packet organizations that contain more than the absolute minimum amount of information should be avoided. In our implementation, the lack of dynamic allocation caused us to make some decisions on data structure design that, in retrospect, we believe were a major source of errors. Our basic packet format included numerous special fields for system use, including link fields for packet queues. Thus, only one packet allocation routine was needed to allocate storage for the packet, and all information typically associated with packets. Unfortunately, we also decided to use argument lists composed on the user's stack as packets, avoiding an extra copy for each message. These two types of packets were inherently incompatible and discerning between the two added unnecessary complexity. A simpler packet format, containing only information mandated by the hardware and a message followed by an indirect link to other information, would have greatly simplified all packet handling in the kernel and probably would have avoided our most difficult bugs.

6.3.5. Avoid Expensive Instructions

Be wary of expensive instructions, principally when programming in a high-level language. On the PDP 11/23, the multiply, divide, and shift instructions are extremely time-consuming relative to the other instructions, making bit operations expensive. Assembly language listings were used to determine where and why such instructions were used; where possible, statements were recoded to avoid use of these instructions. We found that attempts to reduce packet size by packing multiple fields in a single word often resulted in a considerable increase in processing time to decode the fields, a factor of two or three more than the time required to transmit the information in a separate word.

6.3.6. Avoid Mutual Exclusion and Synchronization

Mutual exclusion and synchronization are expensive operations and should not be used in situations where they are not required. General abstractions protected by monitors and accessed only by protected procedure calls are extremely attractive from a programmer's point of view; however, their overuse can be detrimental to performance. In our implementation, each call to a protected module requires about 50 instructions of overhead, regardless of the size of the called procedure. Procedures that simply return without modifying data within the monitor should be coded as non-protected procedures. The representation of some abstractions may require exclusion simply to examine the abstraction; in practice, we have found that many abstractions do not have this property. For example, our implementation contains numerous interface modules that implement message queues; procedures to add and retrieve messages from the queue are exported as part of the module. Each call from outside the monitor to retrieve a message from a queue is preceded by a test to ensure that the queue is non-empty, avoiding an interface call if, as is frequently the case, the queue is empty. This is not equivalent to using a protected procedure since some other process may be in the act of adding a message to the queue; testing the queue from outside the module could suggest no message is in the queue, while an interface call would block until the process in the module completed, after which a message could be retrieved. In this instance, no errors result as the message will be retrieved the next time the queue is examined. The disadvantage is that the queue representation must be visible outside the implementation module, a small inconvenience for increased performance.

6.3.7. Use a High Resolution Line Clock

A high-resolution line clock in each processor is a vital ingredient for a high-performance implementation. Much of the communications protocol is time dependent; message retransmission, broadcast acknowledgement sequencing, and piggybacked acknowledgements are all driven by the clock. A high-resolution clock allows timeout intervals to be finely tuned for each communication primitive and to be used in more situations. An acceptable resolution is about twice the time required for a context switch, since timeout intervals significantly less than that are insufficient to allow other work to be done by another process. The lack of a high-resolution clock influenced many of our implementation decisions; for example, acknowledgements were not delayed in order to be piggybacked since the minimum delay was too long and would have had a serious effect on the performance. This decision resulted in decreased performance for synchronous broadcast ports because each machine sent separate acknowledgements and replies.

6.4. An Architecture for a Family of Primitives

During the construction of the language kernel, we frequently found our attempts to improve performance were obstructed by the hardware. In this section, we examine the various components in the architecture of a local-area network and suggest architectural support for distributed programming language kernels.

6.4.1. Processor

Our experiments show that the processor can easily become a bottleneck in communication, especially if a high-bandwidth network under light load is employed. The lowest level primitive we implemented, remote memory reference, consumes only

14.55% of the available bandwidth, an effective transfer rate of 18 kilobits/second. Spector was able to generate a load of 42% on a 2.94 megabits/second Ethernet by using microcode and a faster processor (approximately 0.33 MIPS), resulting in an effective transfer rate of 1.28 megabits/second. Spector suggests that each processor in a local network of up to 50 processors connected by a 100 megabits/second network needs to execute at a rate of 1 MIPS to be effective; our results suggest that a 0.17 MIPS processor such as ours is incapable of effectively utilizing current technology, even if a small number of processors is involved and the local network has only 10 megabits/second bandwidth.

If, as we assume, the network has been constructed to increase performance, it is reasonable to assume that higher-performance processors would be employed. Small networks of up to 10 processors can efficiently utilize a 1 to 10 megabits/second communication line if the processors execute instructions at a rate of 0.33 MIPS to 0.5 MIPS. Processors that execute less than 0.33 MIPS do not efficiently utilize current communications technology. The instruction rate of the processor is not as important an issue in communication performance, however, if a separate communications controller (adapter) is used to transmit and receive packets on the network.

6.4.2. Communications Adapter

Spector's remote reference/remote operation model differentiates between two types of operations, primary and secondary. A primary reference is executed by special low-level processes, not user-level processes. Secondary operations are more complex and tend to require process switching and request queuing on the main processor. These operations are distinguished by their complexity in relation to the capabilities of the communications adapter. In particular, with the appropriate hardware, a primary reference could be executed directly by the communications adapter. The sophistication of the communications adapter is likely to be the single most important factor in the efficiency of extremely optimized, primary operations such as remote memory reference.

The performance of our implementation of remote memory references was seriously affected by the lack of a communications processor to process the references. Nearly one third of the instructions required to perform a reference were used to field the interrupt and switch contexts. An intelligent controller could field and process the request without requiring a context switch on the main processor. Evidence is provided by Spector's microcoded version of the remote memory reference, which required 280 microcode instructions, that higher performance is possible. His implementation avoided the user process machinery and, even though the reference still executes nearly 100 times slower than the same reference executed locally, most of the time was spent on transmission. Increased bandwidth would greatly improve the performance of Spector's implementation, whereas greater bandwidth would not have as important an effect on the performance of our implementation because a large percentage of the time required to process a reference is consumed by the process machinery.

A communications adapter composed of a bit-sliced processor with local memory would greatly improve performance. Such a processor, dedicated to communications, can have a significant effect on performance as communication traffic increases. Unfortunately, it may not be a cost effective alternative, notably so when the resulting communications adapter is more sophisticated, hence more expensive, than the main processor. Without some form of front-end processor, however, the performance of communication is limited by the process switching machinery on the main processor. As long as process switching is involved, primitive remote operations, such as remote memory reference, will continue to require two orders of magnitude more time to execute than a comparable local operation;

composition of a request, transmission time, request processing, and a context switch will continue to require on the order of 100 instruction times.

We would also like the communications adapter to implement *scatter-gather*, that is, multiple addresses may be used to specify the source or destination address of a packet. This avoids some packet assembly operations in the main processor, reducing copy costs and improving performance. If multiple addresses are allowed, copy costs can be eliminated from the cost of communication. For example, the address of each parameter in a message might be transmitted to the communications adapter, along with the address of the packet header. The result is an implementation similar to call-by-reference in that only addresses are necessary (although the communications adapter would also be given the length of each field), rather than the copy operation used for call-by-value parameters.

A scatter-gather communications adapter can also help avoid copying data between different segments in virtual memory into a common segment. A kernel residing in a virtual memory organization that assigns a different set of mapping registers to each user may have to use more than one set of registers to map all the addresses specified in a transmission request. For example, the packet header may permanently reside in kernel address space, while the message body is always transmitted from the user's address space. (Typically, no message is transmitted that contains fields in more than one user's space.) The kernel maps virtual addresses to physical addresses, that are then sent, as a transmission request, to the communication adapter.

6.4.3. Memory

Memory requirements are minimal and easily met. If the network is asynchronous, the bandwidth of the memory is unlikely to be a limiting factor. Direct reception of messages on a synchronous network with 100 megabits/second bandwidth is possible if the memory has a cycle time of about 300 ns.

The organization of memory can have an impact on the performance of the kernel. For example, on the PDP 11/23, there are two modes, kernel and user mode. Kernel programs run in kernel address space, while user programs are mapped to memory using one of a set of four mapping registers. Operations requiring a change in the current set of registers will take longer than operations that do not. The Intel 8086 [26] uses segment registers to refer to a user's data, but provides a *segment override prefix*, a one byte instruction that specifys the segment register to be used. The overhead, in this case, of referencing a user's data from within the kernel would be quite small, assuming the user's segment registers are readily available.

6.4.4. The Network

When we consider the network as a whole, other issues arise that have considerable impact on the kernel implementation. These include network topology, network bandwidth, processor homogeneity, and language homogeneity. The topology of the network will have an impact on the basic communication protocol within the kernel, as well as the need for routing algorithms. The available bandwidth is obviously an important factor in the system performance. Also, even though we have limited our discussion to local-area networks, we still must address the issue of heterogeneous networks. It may be desirable to construct a high-performance, local-area network using different types of processors. In addition, there may be compelling reasons to allow multiple language implementations on the network.

6.4.4.1. Network Topology

A token ring architecture that allows a message to be acknowledged at little or no cost (an acknowledgement bit is set in the message as it passes through the destination machine) would have a significant effect on the performance of all higher-level communication primitives, particularly broadcasts. The speedup attained by broadcasting would not be limited by the time required to send an acknowledgement from each additional machine in a broadcast since acknowledgements would be free. Such an architecture would permit the theoretical speedup of a factor of N to be achieved when broadcasting to N machines.

Alternatively, a packet-switched network would require the communications kernel to route messages, an unnecessary complication in a broadcast network. Such an organization would clearly have an adverse affect on performance; however, other circumstances may dominate.

6.4.4.2. Network Bandwidth

Current technology readily supports 10 megabits/second communication in a local-area network; 100 megabits/second bandwidth will be commonplace in the near future. Figure 6-2 summarizes the execution costs of the various models of communication we have implemented in light of increased bandwidth and processor speed. (The figure assumes 2 byte arguments.)

One observation of note is that, since our communication primitives are processor bound, an increase in bandwidth from 10 megabits/second to 100 megabits/second has little effect on the performance of any of the communication primitives (assuming reasonably small messages from 2 to 64 bytes in size). On the other hand, an increase in processor speed improves performance significantly. However, the increased bandwidth would have a more dramatic effect on system performance as the number of processors in the network increased.

6.4.4.3. Processor Homogeneity

Frequently, local-area networks are composed of heterogeneous processors. Although StarMod programs consist of homogeneous logical processors, there is no requirement that these logical processors be mapped to a set of homogeneous

Projected Execution Time (in ms.) for Higher Bandwidth Networks and Higher Performance Processors

Processor MIPS	Network Bandwidth	Async. Port Call	Sync. Port Call	RPC	Remote Mem Ref
.17	1 Mb	11.11	20.73	20.13	0.880
.17	10 Mb	10.95	20.41	19.81	0.765
.17	100 Mb	10.94	20.38	19.78	0.753
.33	1 Mb	5.64	10.54	10.24	0.504
.33	10 Mb	5.48	10.22	9.92	0.389
.33	100 Mb	5.47	10.19	9.89	0.377
.5	1 Mb	3.82	7.14	6.94	0.379
.5	10 Mb	3.66	6.83	6.63	0.263
.5	100 Mb	3.65	6.80	6.60	0.252
1	1 Mb	2.00	3.75	3.65	0.253
1	10 Mb	1.84	3.43	3.33	0.138
1	100 Mb	1.82	3.40	3.30	0.127

Figure 6-2

physical processors. In particular, we examined one form of delayed binding, qualified references, in which a form of quadruples was used to encode the reference for interpretation by the remote processor. This approach was taken for those cases where the processor on which the reference is to be executed is a different architecture from the requesting processor.

An additional problem that arises in heterogeneous networks is the issue of type translation. Just as local addresses are not considered meaningful to a remote site, a local interpretation of a bit string as some basic data type may not be a meaningful interpretation for a different architecture. In other words, even basic types may require marshalling when transmitted between different architectures. In [22], Herlihy and Liskov address the problem of type translation in heterogeneous networks.

6.4.4.4. Language Homogeneity

StarMod programs encapsulate logical, distributed processors that combine to form a logical network, homogeneous with respect to the implementation language. It is conceivable, however, for logical StarMod networks to communicate with other logical processors or networks, implemented in some other language, within the realm of a single physical network. Crossing the gap between logical networks is similar to the problem of communication between different physical networks; a uniform interface must be defined through which the networks communicate.

A StarMod program can declare templates for objects (e.g., processes, procedures, ports, variables) that are defined by a logical network implemented in some other language. The template declaration provides all the information necessary for StarMod programs to communicate with processes residing in different logical networks in a type-safe manner (limited, of course, by the type checking capability of the other logical networks). The same approach is often used to present a uniform interface between different programming languages on the same processor.

Certain properties must be consistently maintained by the kernel on each physical processor, independent of the particular implementation language. Communication protocols and address binding are two issues of concern. Message formats must be the same for all communicating logical networks or each processor must translate incoming messages into an internal representation, significantly degrading performance.

One advantage of a specialized language kernel is that the implementation can be tailored to the types of communication supported by the language. This applies especially to the protocol used in communication. A special-purpose kernel is unlikely to implement a general communications protocol similar to those used for most local-area networks. To communicate with processes that run above such a protocol, the StarMod kernel would have to communicate in terms of that protocol. A certain loss in performance is traded for the added flexibility of communicating with those processes.

Additionally, address binding must be consistent if remote memory references or remote procedure call are to be allowed between logical networks. Techniques for address binding are often dictated by architectural concerns; thus, a network composed of homogeneous processors is likely to have uniform address binding on all processors. This is especially true for microprocessor-based systems without virtual memory. In any case, static addresses contained in remote references must be mapped to run-time addresses. The mapping function need not be the same on each processor, even though the domain of static addresses is the same for all machines.

6.5. Summary

In this chapter we evaluated the communication primitives using various criteria, including performance, discussed our experiences in developing the language kernel, and considered architectural support for a distributed programming language kernel, including processor, memory, communications adapter, and bandwidth requirements.

Chapter 7
CONCLUSION

7.1. Summary

In this dissertation we have demonstrated that multiple forms of communication, spanning a spectrum of complexity and performance, can be consistently integrated into a single programming language and its kernel. Message-based communication, remote procedure call, broadcasting, and remote variable references were incorporated into StarMod in a manner consistent with the modular philosophy and virtual-network orientation of the language.

Analysis of the implementation's performance shows that we have achieved good communication performance without resorting to special hardware or microcode. Comparisons with other work show that the performance results we observed are at least comparable to, and often better than, the results of others. We attribute these performance results to the specialization of the language kernel, which tailors the implementation to the communication primitives supported.

An important aspect of our work is that we have quantified the costs and benefits associated with various models of communication. In particular, we have shown that low-level operations executed processor-synchronously are an order of magnitude more efficient than higher-level communication primitives executed as process-synchronous operations.

Finally, we have developed a distributed programming language testbed environment that can serve as a vehicle for future study of programming in distributed systems.

7.2. Future Research

In this section we suggest problems for future research that are natural extensions of the work presented in this dissertation or that are related to our work.

7.2.1. Delayed Bindings

One way to minimize communication costs is to minimize communication. We considered a form of delayed binding that allowed qualification to take place on a remote processor, reducing marshalling and transmission costs. This is just one example of delayed binding, in which interpretive code is produced by the compiler and transmitted to a remote processor under the assumption that the code could be transmitted quickly and the data could not. We can extend the concept to expressions (expression evaluation takes place at the source of the operands and the result is transmitted back to the origin of the expression) and even statements (procedures migrate to data if the data is large and the procedure is small).

7.2.2. High-Performance, Processor-Synchronous Operations

We investigated the performance of low-level, processor-synchronous operations, in particular remote variable references, and found that such operations are much more efficient than process-synchronous operations. Also, Nelson's work demonstrated that remote procedures could be executed very quickly if allowed to execute as processor-synchronous operations. Further experimentation should be undertaken to determine what other types of operations might reasonably be executed processor-synchronously and mechanisms developed to specify and determine synchrony.

7.2.3. Network Contention

The observed performance of network communication protocols in the presence of contention is frequently much worse than expected. We have presented performance results in best-case situations, with no network contention except that introduced by the particular communication primitive of interest. Practical experience using real-world, local-area networks with average network load is necessary to assess the genuine value of our performance results.

7.2.4. Network Architectures

The topology of the network is an important factor in the performance of distributed programs. Our results are based on a bus-structured network without collision detection. Further experimentation is needed to determine the effect on performance of other network architectures, including token ring, packet-switched, and circuit-switched networks.

7.2.5. Dynamic Mapping

For the most part, we have presented a static view of the world. Truly distributed systems tend to exhibit dynamic properties that are not supported by the programming language as we have defined it; in particular, we have not considered processor failures, network partitioning, the addition of new processors, or process migration. Further extensions are needed to provide these dynamic aspects to our otherwise static approach to programming distributed systems. A reasonable balance between the advantages of static binding inherent in most programming languages and the dynamic binding found in most operating systems is needed to ensure maximum programmer flexibility while maintaining high performance.

7.2.6. User Experience

An important hurdle that must be overcome before distributed systems can reach their full potential is the problem of distributing existing concurrent programs. One way to greatly ease the effort necessary to transform a concurrent program into a distributed program is to incorporate transparent primitives for distributed programming into a high-level, concurrent programming language. If remote memory references and nonlocal variable references are indistinguishable, processes that communicate using shared variables may be distributed across many machines without extensive code modifications. Similarly, a message-based communication facility that inherits its syntax and most of its semantics from the procedure model allows a concurrent program executing procedures to be transformed into a distributed program sending messages without much effort.

User experience is the most important factor in assessing the ultimate value of language-level primitives for communication. The implementation described in this dissertation is a major step towards gaining the necessary experience; it lays a foundation upon which future work in high-performance, language-level primitives for communication in local-area networks can be based.

Appendix

PDP 11/23 Operation Times[*]

Increment register	1.72 *us.*
Move x to y (both in memory)	7.23 *us.*
Move x to y (both in registers)	1.72 *us.*
Jump to address	4.27 *us.*
StarMod procedure call	83.60 *us.*
Interface procedure call	247.70 *us.*
Process creation/termination	274.70 *us.*
Process switch	220.40 *us.*
Send a signal	190.00 *us.*
Wait for a signal	236.00 *us.*

[*] Timing figures for machine instructions are from [15]. Other figures were derived from the implementation.

Bibliography

[1] Andrews, G.R., "Synchronizing Resources," *TOPLAS* 3, 4, pp. 405-430 (Oct 1981).

[2] Apt, K.R., Francez, N., and de Roever, W.P., "A Proof System for Communicating Sequential Processes," *TOPLAS* 2, 3, pp. 359-385 (July 1980).

[3] Ashcroft, E.A., "Proving Assertions About Parallel Programs," *JCSS 10*, pp. 110-135 (Jan 1975).

[4] Balzer, R.M., "PORTS -- A Method for Dynamic Interprogram Communication and Job Control," *Proc. of AFIPS SJCC Computer Conf*. 39, (1971).

[5] Baskett, F., Howard, J.H., and Montague, J.T., "Task Communication in Demos," *Operating Systems Review 11*, 5, pp. 23-31 (Nov 1977).

[6] Boggs, D.R., Shoch, J.F., Taft, E.A., and Metcalfe, R.M., "Pup: An Internetwork Architecture," Report CSL-79-10, Xerox Palo Alto Research Center (1979).

[7] Brinch Hansen, P., "The Programming Language Concurrent Pascal," *IEEE Trans. on Software Eng. 1*, 2, pp. 199-207 (June 1975).

[8] Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept," *Comm. ACM 21*, 11, pp. 934-941 (Nov 1978).

[9] Cheriton, D.R., Malcolm, M.A., Melen, L.S., and Sager, G.R., "Thoth, a Portable Real-Time Operating System," *Comm. ACM* 22, 2, pp. 105-115 (Feb 1979).

[10] Clark, D.D., "Modularity and Efficiency in Protocol Implementation," Technical Report, Computer Systems and Communications Group, MIT Laboratory for Computer Science (July 1982).

[11] Cook, R. and Lee, I., "A Contextual Analysis of Pascal Programs," *Software - Practice and Experience 12*, 2, pp. 195-203 (Feb 1982).

[12] Cook, R.P., "*MOD - a Language for Distributed Computing," *IEEE Trans. on Software Eng*. 6, 6, pp. 563-571 (Nov 1980).

[13] Dalal, Y.K., "Broadcast Protocols in Packet Switched Computer Networks," Ph.D. Thesis, Stanford University (Apr 1977).

[14] Digital, et al., The Ethernet Local Area Network, Data Link Layer and Physical Specifications Version 1.0. (Sept 1980).

[15] Digital Equipment Corporation,, Microcomputers and Memories. (1981).

[16] Dijkstra, E.W., "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs," *Comm. ACM 18*, 8, pp. 453-457 (Aug 1975).

[17] DuBourdieu, D.J., "Implementation of Distributed Transactions," *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 81-94 (Feb 1982).

[18] Fabry, R.S., "Capability-Based Addressing," *Comm. ACM 7*, 7, pp. 403-412 (July 1974).

[19] Feldman, J.A., "High Level Programming for Distributed Computing," *Comm. ACM* 22, 6, pp. 353-368 (June 1979).

[20] Finkel, R.A. and Solomon, M.H., "The Arachne Distributed Operating System," TR #439, University of Wisconsin - Madison Computer Sciences (July 1981).

[21] Geschke, C. and Mitchell, J., "On the Problem of Uniform References to Data Structures," *Proc. Int. Conf. on Reliable Software*, pp. 31-42 (June 1975) Also in SIGPLAN Notices 10, 6.

[22] Herlihy, M. and Liskov, B., "A Value Transmission Method for Abstract Data Types," *TOPLAS 4*, 4, pp. 527-551 (Oct 1982).

[23] Herlihy, M.P., "Transmitting Abstract Values in Messages," Master's Thesis, MIT Laboratory for Computer Science (Apr 1980).

[24] Hoare, C.A.R., "Monitors : An Operating System Structuring Concept," *Comm. ACM 17*, 10, pp. 549-556 (Oct 1974).

[25] Hoare, C.A.R., "Communicating Sequential Processes," *Comm. ACM 21*, 8, pp. 666-677 (Aug 1978).

[26] Intel Corporation,, The 8086 Family User's Manual. (Oct 1979).

[27] Keller, R.M., "Formal Verification of Parallel Programs," *Comm. ACM 19*, 7, pp. 371-384 (July 1976).

[28] Lamport, L., "Proving Correctness of Multiprocess Programs," *IEEE Trans. Software Eng. SE*-3, 2, pp. 125-143 (Mar 1977).

[29] Lamport, L. and Schneider, F.B., "The 'Hoare Logic' of CSP, and All That," TR 82-490, Cornell University (May 1982).

[30] Lampson, B.W. and Sturgis, H.K., "Crash Recovery in a Distributed System," unpublished, Xerox Palo Alto Research Center (Apr 1979).

[31] Lampson, B.W., "Applications and Protocols," in *Distributed Systems - Architecture and Implementation*, ed. B.W. Lampson et al.,Springer-Verlag, Lecture Notes in Computer Science 105 (1981).

[32] Lantz, K.A., "Uniform Interfaces for Distributed Systems," Ph.D. Thesis, University of Rochester (May 1980).

[33] Lauer, H.C. and Needham, R.M., "On the Duality of Operating System Structures," *Operating Systems Review 13*, 2, pp. 3-19 (Apr 1979).

[34] LeLann, G., "Motivations, Objectives, and Characterization of Distributed Systems," in *Distributed Systems - Architecture and Implementation*, ed. B.W. Lampson et al.,Springer-Verlag, Lecture Notes in Computer Science 105 (1981).

[35] Levin, G.M. and Gries, D., "A Proof Technique for Communicating Sequential Processes," *Acta Informatica 15*, pp. 281-302 (1981).

[36] Li, L. and Hughes, H.D., "Definition and Analysis of a New Protocol," *Proc. 6th Conf. on Local Computer Networks*, (Oct 1981).

[37] Li, L., Hughes, H.D., and Greenberg, L.H., "Performance Analysis of a Shortest-Delay Protocol," *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 259-273 (Feb 1982).

[38] Liskov, B., "Primitives for Distributed Computing," *Proc. 7th Symp. on Operating Systems Principles*, (Dec 1979).

[39] Liskov, B., "Linguistic Support for Distributed Programs: A Status Report," Computation Structures Group Memo 201, MIT Laboratory for Computer Science (Oct 1980).

[40] Liskov, B., "On Linguistic Support for Distributed Programs," *IEEE Trans. on Software Eng. 8*, 3, pp. 203-210 (May 1982).

[41] Liskov, B. and Scheifler, R., "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *Proc. 9th POPL Conf.*, pp. 7-19 (1982).

[42] Michael, A., Private Communication. (Oct 1982).

[43] Mitchell, J.G., Maybury, W., and Sweet, R., "Mesa Language Manual," TR CSL-79-3, Xerox Palo Alto Research Center (Apr 1979).

[44] Nelson, B., "Remote Procedure Call," Ph.D. Thesis, Carnegie-Mellon University (May 1981).

[45] Ousterhout, J.K., Scęlza, D.A., and Sindhu, P.S., "Medusa: An Experiment in Distributed Operating System Structure," *Comm. ACM 23*, 2, pp. 92-105 (Feb 1980).

[46] Peterson, J.L., "Notes on a Workshop on Distributed Computing," *Operating Systems Review 13*, 3, pp. 18-27 (July 1979).

[47] Schlichting, R.D. and Schneider, F.B., "Using Message Passing For Distributed Programming: Proof Rules and Disciplines," TR 82-491, Cornell University (May 1982).

[48] Spector, A., "Extending Local Network Interfaces to Provide More Efficient Interprocess Communication Facilities," *Proc. ACM Pacific 80*, (Nov 1980).

[49] Spector, A., "Multiprocessing Architectures for Local Computer Networks," Ph D. Thesis, Stanford University (Aug 1981).

[50] Swan, R.J., Fuller, S.H., and Siewiorek, D.P., "Cm* - A Modular, Multi-microprocessor," *Proc. AFIPS 1977 NCC 46*, pp. 637-644 (1977).

[51] Tanenbaum, A., *Computer Networks*, Prentice-Hall, Inc. (1981).

[52] United States Department of Defense, Ada Programming Language, (Military Standard) (Dec 1980).

[53] Wall, D.W., "Mechanisms for Broadcast and Selective Broadcast," Ph.D. Thesis, Stanford University (June 1980).

[54] Wall, D.W., "Selective Broadcast in Packet-Switched Networks," *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 239-258 (Feb 1982).

[55] Wirth, N., "The Design and Implementation of Modula," *Software - Practice and Experience 7*, 1, pp. 67-84 (Jan 1977).

[56] Wirth, N., "Modula - A Language for Modular Multiprogramming," *Software - Practice and Experience 7*, 1, pp. 3-35 (Jan 1977).